adaptSTM - An Online Fine-grained Adaptive STM System

Mathias Payer
ETH Zurich, Switzerland
mathias.payer@inf.ethz.ch

Thomas R. Gross
ETH Zurich, Switzerland
trg@inf.ethz.ch

Abstract

Transactional memory (TM) is an attractive platform for parallel programs, and several software transactional memory (STM) designs have been presented. Here we explore several optimization opportunities to adapt to the running program and to adapt parameters that are optimized for the average case. Depending on the program the transactional load can vary per thread (e.g., client/server threads), or the program uses multiple phases of computation with different transactional loads. Therefore it is important that the STM adapts to the current situation, and that the adaptation process is short, efficient, and thread-local.

We present *adaptSTM*, a competitive, word-based STM library that is based on a global clock and an array of combined global versions (timestamps) and locks. To keep track of transactional data *adaptSTM* implements a multi-level buffer and uses read-set extension to achieve competitive performance.

The fine-grained adaptation system measures important runtime parameters like read- and write-locations, commit-, and abort-rate, and is able to adapt important parameters like write-set hash-size, hash function, and write strategy based on runtime statistics on a per-thread basis. Using the STAMP benchmarks, adaptSTM is compared against two other STM libraries, TL2 and tinySTM.

adaptSTM outperforms TL2 for all benchmarks except SSCA2, and offers performance that is competitive with tinySTM for low-contention benchmarks; for high-contention benchmarks adaptSTM outperforms tinySTM. Comparing *adaptSTM* to TL2 results in an average speedup of 43% for 8 threads and 137% for 16 threads. The speedup results from the read-set extension, several non-adaptive optimizations and the fine-grained adaptation system. Adaptation alone increases performance on average by 4.33% for 16 threads, and up to 10% for individual benchmarks, compared to *adaptSTM* without active adaptation.

1. Introduction

Transactional memory attempts to replace locks by executing critical sections in a transactional manner. Transactions are well known from databases and can be used to group several instructions into one atomic block that is atomically committed to memory.

Transactional memory promises the speed of fine grained locking without requiring a low-level view of the system. The programmer specifies *atomic* sections and the transactional memory accesses. The runtime system ensures mutual exclusion.

Transactional memory can be implemented in software (STM), in hardware (HTM) or as a hybrid approach (HyTM). The use of an array of locks is the reason why STM systems can reach performance as good as a well written non-STM program that uses fine grained locking without the added complexity and code to handle these locks efficiently.

Most current transactional memory systems are implemented in software. The advantage of STM systems is that different algorithms and parameters can be tested without hardware development costs. With careful design the overhead of STM systems can be as low as about 40 instructions per transactional read or write on average (and could be even lower in a low-level assembler implementation).

Current STM systems are the result of many engineering decisions. There is no single decision which is responsible for good performance. Only a careful selection of different parameters results in a competitive baseline.

The contributions of this paper are:

- *adaptSTM*, a fast adaptive STM library that collects runtime information and adapts to the given workload as well as to phase changes.
- A fine-grained, thread-local, low overhead adaptation mechanism that handles phase changes and different thread workloads concurrently.
- A detailed analysis of the presented thread-local adaptive parameters using the STAMP [3] benchmark suite as well as benchmarks comparing the *adaptSTM* library to the current versions of TL2 [6] and tinySTM [10].

The rest of the paper is organized as follows: Section 2 presents an overview over general STM parameters, Section 3 shows some implementation details, followed by Section 4 showing the advantages and possibilities of a fine-grained thread-local adaptive STM system. Section 5 presents benchmarks relative to current STM implementations and offers an evaluation of the different adaptive parameters. At last Section 6 concludes.

2. Design decisions and related work

Early STM systems [11, 12, 14, 15, 20, 26] evolved out of limited hardware TM systems. STM systems are either word-based or object-based and work on word or object granularity. Most of the current word based STM implementations agree on general design decisions. They use a global locking table [5, 7, 8, 13, 22, 25, 27] and a hash function to distribute the available locks over the complete memory region. All STM systems keep per transaction information about accessed locations. Local read-sets and write-sets are used to verify committability of transactions, and write-sets are used to undo transactions upon conflicts or aborts.

One other common criterion is the use of a global clock [4, 23, 24, 28]. The global time is sampled at the start of the transaction and the version of write-locations is set to the current time in the commit phase. This simplifies validation of reads and writes as the STM system only checks if the version (timestamp) of the accessed address is smaller or equal than the start time of the current transaction.

Two current fast lock-based STM systems where the source code is available are TL2 [4, 6] and tinySTM [9, 10]. Both systems follow the general principles in this section and use a combined lock/version-array and a global clock. TinySTM is based on a single-version, word-based variant of the lazy snapshot algorithm [23]. These current systems are used to compare between different design decisions and optimizations.

The book *Transactional Memory* [18] by Larus and Rajwar gives an overview about the history of many TM systems. It shows different STM designs and includes implementation details.

But just like current STM implementations agree on general design decisions they differ in subtle and important details. The following sections will give more details about important design decisions and trade-offs for an efficient STM implementation.

2.1 Locks and versions

Many lock based STM systems use a single combined global array [8, 13, 16, 25] for locks and versions (timestamps) similar to TL2 and tinySTM. Each entry in the global lock array covers some area of memory dependent on the hash function and contains either the version of that memory area, or a pointer to the transaction that holds the lock for this region.

The lowest order bit is used to distinguish between locks and versions. If the bit is set, then the remaining bits contain a version, otherwise the location contains a pointer to a transaction which holds the lock for that memory region.

2.1.1 Hash function

The hash function used to map memory regions to locks must be shared by all transactions running concurrently. A good hash function ensures that concurrent reads and writes to different addresses are mapped to different locks. Hash functions have been studied for a long time [1, 17] but most proposed algorithms are not applicable to STM systems, as hashing is very frequent, must be fast, and perfect or near perfect hashing is not needed.

To limit the number of instructions in the hashing function, most STM systems use a single right shift instruction and a logical *and* to hash memory addresses to entries in the lock/version table. This design is a trade-off between potentially higher contention and speed of the hash function.

Under the assumption that a transaction accesses memory locations close to each other (locality), shift hash functions lead to another advantage. Nearby memory locations are mapped to the same lock, thereby reducing (i) the number of locks a transaction holds and (ii) the number of cache-lines a lookup touches to access data.

2.1.2 Hash-table size

The size of the hash-table decides on the amount of memory which is mapped to a single lock. There are some limitations to the size of the hash-table: (i) the overall memory consumption of the STM library must be considered, especially for a 64bit word-size, (ii) the complete hash-table must be initialized during start-up, and whenever the global clock is reset, and (iii) there is a trade-off between the potential lock contention and the overall number of locks a transaction holds.

As the hash-table gets larger, the locking of the STM system will be more fine grained with the advantage of less false lock contention and the disadvantage of additional overhead for an excessive amount of locks.

2.1.3 Locality and granularity

Locality and lock granularity can be tuned by the hash function. If we assume a 64bit pointer, and a hash function like `addr>>5`, then the lock granularity is 2^5 resulting in a 32 byte wide stride. Depending on the size of the hash-table, this region will be locked for all masked higher-order bits.

2.1.4 Lock acquisition strategies

Global locks can be acquired in different fashions. The main design decision is to use either *eager, encounter-time* locking, or to use *lazy, commit-time* locking.

If the transaction uses *eager locking* it will acquire the locks as soon as the first write to a memory location covered by that lock is encountered. *Eager locking* signals concurrent

transactions that a specific location is currently locked. This scheme makes it possible to abort early on conflicts, but might lead to cascading aborts where, e.g., T1 aborts due to T2 holding a lock and T2 aborts due to other conflicts. *Eager locking* can be used for read locations as well, this is called *visible reads*. Transaction-local book-keeping of read entries is called *invisible reads*.

Lazy locking uses local book-keeping to keep track of locks and only acquires the locks at commit-time. This avoids the problem of cascading aborts, but conflicts are detected late.

Of the two reference systems TL2 uses lazy locking to reduce the overall time a lock is held, tinySTM uses eager locking to reduce contention. These two different strategies give us a range to compare against.

2.2 Read-set, lock-set, and write-set

Every transaction needs some data structures to keep track of transactional reads and writes, see Figure 1: (i) the *read-set* saves tuples of version and read addresses that have been read but not yet written, (ii) the *write-set* contains tuples of addresses and old/new values that have been written during the current transaction, and (iii) the *lock-set* contains tuples of version and lock addresses that have been taken during this transaction.

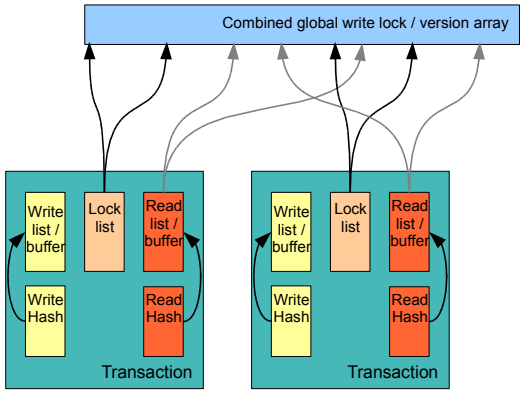

Figure 1. Overview of transaction-local data, including read-set, lock-set and write-set.

The sets are transaction-local and are used to log the transaction's progress. Different data-structures can be used to implement these sets. Most STM systems use arrays that are expanded if the capacity is reached and combine *lock-set* and *write-set*. This is a simple approach that scales up to a few entries.

An alternative to arrays are hash-tables. Entries in the *read-set* and *write-set* can be hashed into different hash-tables for fast lookups. This strategy is beneficial if there is a high number of addresses that is read or written.

The *lock-set* is only needed to scan trough taken locks at a commit or an abort, and random lookup is not needed, therefore an array suffices. The *lock-set* contains versions and lock addresses for writes as well, so there is no need to include the version in the *write-set*. This makes the write-set smaller and increases cache locality of the written locations.

2.2.1 Bloom filter

A Bloom filter [2] is a probabilistic data structure, which can add elements and test set membership of an element in constant time. A Bloom filter consists of an array of length m, initially set to 0. To add an element to the data structure it is fed to k hash functions. The resulting hash value and the value of the Bloom array are combined with a logical *or*. To test for set membership the element's hash value is calculated. If the hash value contains one position with a logic 1 where the Bloom filter contains a logic 0, then the element cannot be part of the set.

In an STM system Bloom filters can be used for a faster check if a location already exists in a read-set or write-set. If the Bloom filter matches, then the element can be part of the set, and the set must be checked. The probability of false positives increases with the number of elements that were added to the Bloom filter.

2.3 Write buffering

Any STM library must buffer transactional writes to ensure correctness. There are two different orthogonal strategies to buffer writes. A transaction can buffer the written locations locally and write the data back as soon as the transaction is in the commit phase (*write-back* or lazy-update). Alternatively the transaction can write the data directly to memory and cache the original value transaction locally (*write-through* or eager-update). The use of *write-through* limits the choice of the locking strategy to *eager*-locking as the location must only be accessed by one transaction exclusively; *write-back* can use both eager-locking and lazy-locking as data is only written to main memory during the commit phase.

Both buffering strategies have their advantages and drawbacks. A *write-back* strategy offers cheap abort possibilities, but the commit phase takes longer as all data is written to memory. With a *write-through* strategy the commit phase is cheap, but an abort is more expensive as all write locations need to be restored, if the transaction is aborted.

If the current program phase suffers under high contention and a high abort rate, then *write-back* is beneficial, otherwise *write-through* is faster.

2.4 Global clock

The system samples the global clock at the start of a transaction and compares all future read and write locations with the given start time. If the time of any write location's lock is later, the transaction must be (i) validated and extended or (ii) aborted.

If a transaction wants to commit it must validate its read-set. If the validation succeeds, the global clock is increased and the write location's locks are updated to the current time and the data is committed to main memory.

The system of a global clock as used in *adaptSTM* and other STM libraries like TL2 and tinySTM is only valid for shared memory systems and might be a source of memory contention. Especially if there exists no shared cache between cores this can lead to bus contention.

To avoid clock overrun on a 32bit system a simple check can be used whenever a transaction starts to synchronize all running transactional threads and to reset the clock. This scheme adds no overhead to the frequent transactional read and write operations and a single comparison to the start of a new transaction. Before the clock overruns all threads are synchronized and the locks and clock are reset. On 64bit systems this clock overrun check is not needed as the 63bit counter will not overrun for reasonable programs.

2.5 Memory allocation

To make transactional memory allocation possible, the STM system must keep track of all memory allocations and deallocations. On a transaction abort deallocations must be undone and allocations must be freed. On a transaction commit deallocations must be freed and allocations must be made visible to all other threads. Hudson et al. [16] studied the effects of concurrent transactional memory allocation and proposed an alternative memory allocator for STM systems in more detail.

2.6 Adaptivity in current systems

Different forms of adaptivity already exist in current systems. Marathe et al. developed ASTM [19], an object-based STM algorithm that extends DSTM [14] and adapts between (i) lazy and eager lock acquire strategies and (ii) two forms of meta-data for transactional objects. TinySTM showed that an eager acquire strategy is better for lock based STMs because of the earlier conflict detection.

Yoo and Lee present a TM system that is extended by an adaptive transaction scheduler [29]. The scheduler detects high contention and throttles the number of concurrent transactions.

TinySTM [10] is the first lock-based STM implementation which includes adaptive dynamic tuning. The implementation can adapt three parameters, (i) the size of the global locking table, (ii) the number of shifts for the global locking table's hash function, and (iii) the size of the hierarchical array. All three parameters must be tuned globally, which potentially limits scalability since all transactions must reach a synchronization point and wait for each other.

3. Engineering of the non-adaptive STM baseline

To implement a fast, competitive, and fine-grained adaptive STM system it is crucial to select a good baseline implementation. The non-adaptive baseline implementation of *adaptSTM* follows Section 2. This section describes trade-offs and design decisions that enable competitive performance compared to TL2 and tinySTM without adaptation.

3.1 Locks and versions

adaptSTM uses a single static global array for versions and locks. Experiments with the STAMP [3] benchmarks showed that a good average size for the lock hash-table is 2^{22}`*sizeof(word)` and a shift by 5 bit `((addr>>5) & (HASH_PATTERN))` offers a good tradeoff between fine-grained locking and additional overhead for handling too many locks. The variance for the used numbers of the hash-table are tolerable throughout the STAMP benchmarks. The runtime difference between a reasonable amount of shifts (4 to 6 bits), and different hash-table (2^{20}, 2^{22}, 2^{24}) sizes is not dominant, as long as the number of hash-table entries is large enough to support enough locks for all parallel transactions.

adaptSTM implements both *eager locking*, and *lazy locking* for write locations, but prefers eager locking to support *write-back* and *write-through* write strategies. The write strategy is selected in the Makefile. An invisible read strategy is used for read locations (other transactions are unaware of other threads' read locations).

3.2 Read-set, lock-set, and write-set

adaptSTM uses arrays for read-sets and lock-sets. For write-sets we use a lazy multi-level approach that starts with a linear list. If the number of locations written reaches a specific mark, the hash-table is built on the fly and future lookups no longer need a sequential scan but use the hash-table for fast access. Hash collisions are enqueued in a linear list.

To reduce the number of additional cache misses *adaptSTM* takes special care to allocate write-sets next to each other and aligned to cache lines. The use of *lock-sets*, the multi-level sets, and the alignment strategy is new in *adaptSTM*.

3.2.1 Optimizations for read-sets and write-sets

adaptSTM proposes different hash functions for transaction local access to the write-set and read-set hash-tables. In the STAMP benchmarks the number of locations read and written is small in most cases. Hence a smaller hash-table and a different hash function can be used compared to the global lock table.

The correct choice of the size of the transaction-local hash-tables is crucial for good performance. Whenever a transaction starts (or reaches a certain number of read locations or write locations) the hash-tables must be initialized. This initialization overhead is significant if the size of the hash-table is not chosen properly. There is a trade-off between larger hash-tables that provide a faster lookup and a higher initialization cost paid for every transaction.

Our experiments with the STAMP benchmarks showed that a hash-table is not beneficial for read entries. For write entries a hash-table for 32 entries is best on average, but the optimal size varies greatly between benchmarks.

Bloom filters of 64bit length are added to the read-set and write-set. But the overhead of using a Bloom filter for the read-set was still larger than only adding the elements to the array. The speed gain of using a Bloom filter before the hash lookup in the write-set is minimal.

3.3 Global clock and read-set extension

adaptSTM uses a global clock that is incremented whenever a transaction commits.

If a transaction reads or writes a location, or tries to commit, the version of an allotted lock can be larger than the transaction's version. A naive approach would abort whenever such a higher version is encountered. To reduce the amount of retried transactions *adaptSTM* first tries to extend the version of the transaction to the current global time. Versions of all untaken locks of the current transaction are validated (for eager locking the locks corresponding to all read locations are validated, for lazy locking all write locks are validated as well). If no location has been written by another transaction since the start of the current transaction (e.g., no old value that is no longer existent was read in the current transaction), then the version can be extended to the current global time and the transaction can continue.

3.4 Contention management

There are various actions an STM can carry out if a lock is currently taken by another transaction: (i) abort the current transaction and retry, (ii) signal the other transaction to abort and continue, or (iii) wait some time and retry to take the lock. An adaptive contention management uses runtime data to dynamically select the best approach.

adaptSTM implements a wait and retry strategy. The current transaction is yielded a configurable amount of times. The yield operations give the other threads the opportunity to release the lock before the current transaction has to abort itself.

4. Fine-grained adaptive STM

Current STM systems are optimized for the average case. They cannot adapt to changing workloads and phase changes. The advantage of an adaptive STM design is that the adaptation mechanism can tailor the STM parameters to current phases and workloads at runtime.

The adaptation system collects important information at runtime to justify a deliberate decision. Important metrics include the transaction frequency, the number of unique read and write locations, the number of hash-table collisions per hash-table (e.g., for the global lock, read-set, and write-set), the number of aborts compared to the number of successful commits, and the quality of hash functions. The metrics are used to select between different write strategies, to adapt the local write-hash function, and to tune the locality of the write-set's hash-table.

4.1 Local vs. global adaptivity

There are two approaches to adaptivity. One is global adaptivity, which changes parameters for all running transactions, and the other is local adaptivity, which changes parameters locally, on a per-thread basis.

The advantage of local adaptivity over global adaptivity is that every thread has its local settings, e.g., a reader-thread will optimize the transactional parameters for best read performance and a writer-thread can optimize for write throughput. *Global adaptivity* can be a bottleneck for scalability as it requires global synchronization and barriers for all threads that make frequent changes of the adaptive parameters expensive. Each thread on the other hand can change the *local* transactional settings whenever a transaction is (re-)started without synchronization overhead.

The disadvantage of local adaptivity is that some changes are not covered in this scheme, e.g., the global lock hash function or the size of the global lock table cannot be changed at runtime without synchronization, but must be preset to a reasonable value. Some global changes like adaptation of the contention manager can be done without synchronization if designed carefully. Even a switch between eager and lazy locking can be implemented without synchronization. Both locking schemes can be used in parallel transactions, although fairness is not guaranteed as the probability for a conflict is higher in a lazy locking scheme.

4.2 Write-back vs. write-through

In a contented environment with a high abort rate it is beneficial to use write-back instead of write-through to commit write changes to memory.

adaptSTM samples the abort rate and decides to switch between write-back and write-through, if the abort rate reaches a threshold. The system uses the average of the last 64 transactions to calculate the abort rate.

4.3 Adapting the size of hash-tables

The size of the write-set hash-table is crucial for good performance. If the hash-table is too large, the overhead of resetting the table every time a transaction starts is high. On the other hand, if the table is too small, then the lookup will be slow due to many hash collisions. In the current implementation hash collisions are queued in a linked list in the same hash-table slot.

The adaptive system samples the moving average of unique write locations per transaction. If the load of the hash-table is more than 33% then the size of the table is doubled. On the other hand, if the load is below 10% then the size of the table is halved. Details about the adaptation policy can be seen in Figure 2. The data were obtained using the STAMP benchmarks and offer a good tradeoff between hash-table size and hash-table collisions.

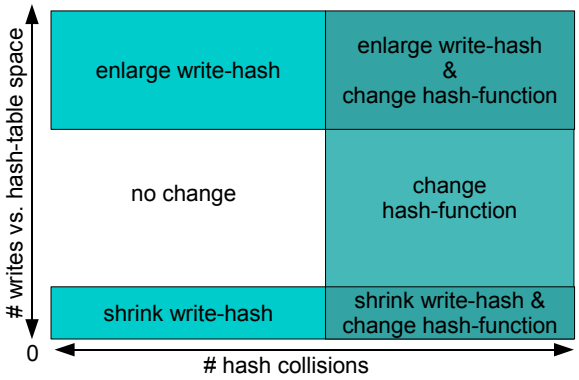

Figure 2. Summary of the hash-table adaptation mechanism, depending on the number of hash collisions and the load in the hash-table.

4.4 Locality tuning for hash-tables

One criterion for low-level hash functions is speed. They should not use more than a couple of instructions, otherwise the cost of the hash function prevails over the better distribution.

An adaptive system can sample the number of hash collisions as well as the quality of the hash function by measuring hotness of individual bits, similar to Bloom filters. Depending on these measurements it is possible to switch to a better hash function. A simple approach is to iterate through a predefined set of hash functions, whenever the number of collisions is above a specific mark.

4.5 Adaptive contention management

An extension of the basic contention management is to scale the number of yield operations according to the overall contention in the system. The current transaction is yielded an amount of times relative to the number of retries for this transaction.

This adaptive contention strategy implements a backoff strategy which retries immediatly if the contention is low, or yields an increasing amount of times in contented situations.

4.6 Adaptive statistics and overhead

An adaptive system needs to collect statistics about the program and running transactions. Collecting performance numbers and additional flexibility to adapt individual parameters incur some overhead, e.g., additional counters and *if*-statements to select the correct setting.

Our non-adaptive system collects information about the number of unique write locations, as well as the overall number of write and read locations. It is cheap to add additional counters for the number of started transactions, committed transactions, and aborted transactions, as these events are relatively rare. More frequent events like lock collisions and hash collisions are more expensive to count, but do not incur a significant overhead.

5. Benchmarks and evaluation

This section presents an evaluation of different optimizations, adaptive parameters, and performance compared to other STM systems based on the STAMP [3] benchmark suite.

The STAMP benchmark suite is designed for transactional memory research and contains a collection of parallel C programs. The parallel code in the benchmarks uses coarse-grained transactions. Inside a transaction all transactional read and write operations are instrumented and redirected to the STM library. The given programs are:

1. **Bayes:** A Bayesian network learning benchmark, using a hill-climbing strategy with both global, and local search.
2. **Genome:** Gene sequencing (and checking) benchmark that reconstructs a gene sequence from segments of a larger gene.
3. **Intruder:** Signature based network intrusion detection benchmark, processing network packets in parallel, and matching them against intrusion signatures.
4. **kmeans:** K-means is a partition based clustering algorithm producing 'hard' clusters.
5. **Labyrinth:** This benchmark finds shortest-paths between pairs of points in a given maze using Lee's algorithm.
6. **SSCA2:** A benchmark consisting of four graph kernels that operate on a large, directed, weighted multi-graph.
7. **Vacation:** The Vacation benchmark implements a travel reservation system consisting of several clients that issue requests and a server that processes the requests.
8. **YADA:** The YADA benchmark implements Delaunay mesh refinement using Ruppert's algorithm.

All benchmarks were measured on two Intel 4-core Xeon E5520 CPUs resulting in a total of 8 cores with 2.27GHz and 12GB main memory. The system uses Ubuntu version 9.04 Jaunty Jackalope for 64bit systems, glibc version 2.9.1, and gcc version 4.3.3-5. Average speedups are calculated by comparing overall execution time for all programs for different configurations. adaptSTM supports both 32bit and 64bit mode. The measurements presented here use the 64bit version.

The evaluation covers results for 1, 2, 4, 8, and 16 concurrent threads. For 16 concurrent threads two threads share one core, and the scheduler decides on the interleaving of the threads. This can lead to additional contention. The 16 thread case shows the non-optimal case where a transactional thread does not have exclusive access to a core but shares it with another concurrent thread.

The next sections represent an analysis of different STAMP characteristics, an evaluation of non-adaptive optimizations like the read-set extension, implications of adaptive param-

eters, and competitive performance compared to tinySTM and TL2.

5.1 STAMP characteristics

The STAMP benchmarks cover a wide range of programs. From few to many transactions, from a low number of reads to a high number of reads, and from a low number of writes to a high number of writes. All these parameters are combined in different programs, and some of them even go through different phases. We assume that the STAMP suite captures reasonable performance characteristics of transactional programs and we are aware of the limitations of using a finite benchmark suite for our measurements.

Bench.	#c	#l	#r		#w	
Bayes[a]	33	1	11	1/26	1	1/2
	1513	2	23	1/423	2	0/20
Genome	2489218	0	36	1/4154	0	0/24
Intruder	23428126	1	23	3/875	1	0/47
kmeans[b]	87382	24	24	1/33	24	1/33
Labyrinth	1026	177	180	3/844	177	0/844
SSCA2	22362279	1	1	1/1	1	1/2
Vacation	4194304	7	394	14/1807	7	0/79
YADA	2415298	13	58	0/1320	16	0/331

[a] Bayes executes two different sequential STM runs.

[b] kmeans has 720 equal sequential runs.

Table 1. STAMP characteristics showing commits (#c), locks (#l), avg., min/max reads (#r), and avg., min/max writes (#w).

As can be seen in Table 1 the STAMP benchmarks are challenging for an STM system, the total number of transactions ranges from a low number of about 1546 for the Bayes benchmark to up to 23428126 for the intruder benchmark in the average case. Therefore the STM library must have a low initialization cost as well as a low constant overhead for additional transactions to keep the overall runtime spent in the STM system low. The average transactional footprint varies as well as the number of transactions. There are benchmarks with a high number of transactional reads and writes (Vacation, Labyrinth, and YADA) and there are benchmarks with a low number of transactional reads, and writes (SSCA2, kmeans, and Bayes). Another challenge of the STAMP benchmarks is the great variance of the transactional load for some benchmarks. For genome the number of read locations can vary between one and 4154, with an average of only 36 read locations. For the Vacation, YADA, and Labyrinth benchmarks the transactional load varies greatly in the number of read and write locations. The varying load poses challenges to the adaptive system, it must adapt to changing situations fast, and cover phases as well. If the adaptive system is not able to adapt fast enough, the configuration is not optimal.

5.2 Evaluation of the read-set extension

Another factor that is important for competitive performance is the read-set extension. Contention and read version failures rise with the number of concurrently running transactions. As contention increases other threads will increase versions for locks, conflicting with the current transaction. It is important to reduce the amount of unnecessary retries. Instead of aborting, *adaptSTM* tries to extend the read-set by validating all previous reads. Re-validation of the read-set, whenever a read version fails is beneficial for the overall performance and reduces the amount of retries.

Bench.	2 threads	4 threads	8 threads	16 threads
Bayes	17	23	27	27
	47%	57%	41%	52%
Genome	1346	3350	8945	10611
	17%	53%	52%	38%
Intruder	536451	1436177	8527748	6656061
	96%	84%	75%	80%
kmeans	175525	569984	1682853	1535232
	100%	100%	100%	100%
Labyrinth	19	46	110	237
	100%	100%	100%	100%
SSCA2	33	106	57	149
	100%	100%	100%	100%
Vacation	1997	4685	8962	6923
	92%	90%	89%	82%
YADA	138791	280942	417134	408544
	95%	94%	91%	79%

Table 2. Number of read-set validations per benchmark for a specific number of threads and percentage of successful validations and read-set extensions.

Table 2 shows that most benchmarks exhibit a high success rate for read-set validation and extension. This optimization reduces contention between threads and increases the commit rate. The benchmarks show that re-validation and read-set extension is successful in the majority of the cases.

5.3 Commit and retry rates

An important performance parameter to measure contention is the retry rate as a function of the number of successful commits. The contention will increase with an increasing number of threads which results in more retries.

As can be seen in Table 3, this assumption holds for the Intruder, kmeans, Labyrinth, and YADA benchmarks. Genome, and Vacation exhibit the same behavior, but the number of retries is very low. However the Bayes, and SSCA2 benchmarks do not follow this pattern. The number of retries is almost constant. An analysis using Valgrind [21] and the callgrind tool showed for the Bayes benchmark that *malloc* and *free* together use more than 40% of the time, the Intruder benchmark uses more than 20% of the time for the *strstr* glibc function, and the Genome benchmark spends more than 37% of the time in the *strcmp* glibc function.

Bench.	2 threads	4 threads	8 threads	16 threads
Bayes	1476	1392	1415	1323
	1%	1%	1%	2%
Genome	2489218	2489220	2489220	2489228
	0%	0%	0%	2%
Intruder	23428127	23428129	23428133	23428141
	1%	5%	24%	18%
kmeans	3844852	3844940	3932505	3932865
	1%	3%	10%	9%
Labyrinth	19	1032	1040	1056
	2%	4%	10%	20%
SSCA2	22362283	22362287	22362295	22362315
	0%	0%	0%	0%
Vacation	4194304	4194304	4194304	4194304
	0%	0%	0%	0%
YADA	2495029	2544550	2581206	2574275
	8%	12%	21%	383%

Table 3. Number of commits per benchmark for a specific number of threads and percentage of retries.

An external limitation is the memory allocation system. The glibc memory allocator uses locks to ensure mutual exclusions. As the number of threads increases the calls to the memory allocator can lead to an unwanted synchronization and linearization. This limitation can be removed if a lock-free allocator or an STM-aware memory allocator is used.

5.4 Design decisions for the global hashtable

Two parameters can be changed for the global hashtable: (i) the size of the hashtable, and (ii) the hash function. It is important that these parameters are selected carefully as they are crucial for good performance.

The most important factor for the size of the hashtable is that the table must be large enough to support enough locks for all concurrent transactions. But one must keep in mind that the size of the hashtable is partly responsible for the static initialization overhead because it must be initialized with zero.

The hash function represents an important tradeoff between data locality and over-locking. If shifting hash functions of the form `(addr<<X) & HASH_SIZE` are used then 2^X bytes are mapped to a single lock. Depending on the data access pattern of an individual transaction and the interaction of the access patterns of concurrent transactions either too many locks are used if concurrency is low and data locality is high, or there is contention with concurrent transactions. Table 4 shows the different options.

The expected result for different configurations of the hashtable size and the numbers of shift bits is that the performance will vary greatly depending on the data locality and data parallelism of individual benchmarks.

In our experiments we used adaptSTM in the default configuration with a fixed number of shifts for the global hash function and a fixed number of entries for the hashtable. We then used STAMP to evaluate different combinations

# Shift bits	Data locality	Result
low	low	Good mapping between lock distribution and locality
low	high	Missed potential for lock optimization
high	low	Possible contention for concurrent threads
high	high	Good mapping between lock distribution and locality

Table 4. Different configurations for hash functions

of number of shift bits and hashtable sizes to reason about the variance in these parameters. Tables 7 through 14 in Appendix A show all combinations of hashtable sizes and number of shifts for 4 threads and all STAMP benchmarks. The figures show averages of 5 runs, standard deviation was low for most runs, except some corner cases with a large number of shifts. The figures compare hashtable sizes of 2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24} , and 2^{26} entries, and number of shifts for 0, 2, 4, 6, 8, and 10 shift bits.

Except for the Genome benchmark, the size of the hashtable has no influence on the runtime of the benchmark. Only for unrealistically small tables and a small number of shifts (0, or 2) the hashtable size has a noticeable effect because of the larger amount of locks a transaction must hold. Larger hashtables lead to better performance in the Genome benchmark with diminishing returns after 2^{20} entries. The results in Tables 7 through 14 in Appendix A show that any hashtable size of 2^{20} to 2^{24} entries is reasonable, and there is no need to adapt the size of the hashtable at runtime.

The number of shift bits in the hash function represents the tradeoff between data locality and overlocking. A higher amount of shift bits increases the stride of continuous memory that a single lock covers. If the transaction already holds the lock to a part of the data structure then no other concurrent transaction can interfere with other parts of the data structure covered by the same lock, potentially reducing the number of retries due to conflicts on the same data structure. On the other hand, if the number of shift bits is too large then a single lock covers more data than the transaction uses and hinders concurrent transactions to access nearby data. Tables 11 and 13 in Appendix A show that some benchmarks like Labyrinth and SSCA2 are immune to changes of the number of shift bits. Benchmarks like Vacation, Genome, Intruder, and YADA have data locality and profit from a minimum number of 2 shift bits. Other Benchmarks like Genome, kmeans, Intruder, and YADA have concurrent transactions that work on nearby data structures. As soon as the number of shift bits is larger than 8 bits, overlocking takes place and reduces the amount of parallelism that can be achieved, resulting in decreased performance. The results in Tables 7 through 14 in Appendix A show that

all benchmarks perform well with a number of shift bits from 2 to 8.

Table 8 shows detailed statistics for the Genome benchmark. The runtime performance is consistent throughout the different configurations for hashtable sizes and number of bits for the hash function. Standard deviation is low for all results. If both the number of shifts for the hashtable and the hashtable size are reasonable then the variation between different configurations is low.

Reasonable choices of the hashtable size and a reasonable number of shift bits result in low changes of the program behavior and runtime differences are not dominant. Therefore adaptSTM uses a fixed hashtable size of 2^{22} entries and a fixed number of 5 hash bits.

5.5 Implications of adaptive parameters

Depending on the workload of the program each adaptive parameter helps to get optimal throughput. This section analyzes the different adaptive parameters and their contribution to the overall result.

The adaptation process and online optimization is stable, leading to a low runtime standard deviation. Table 5 shows the following *adaptSTM* configurations using the average of 5 runs (standard deviation is low for all benchmarks):

naWB: Baseline configuration with write-back methodology and without adaptation.

aWBT: Configuration with activated adaptation, offering dynamic configuration of the write strategy (write-back or write-through), and an exponential drop-off in the waiting time for contented transactions.

aWWH: Adds automatic configuration of the size of the write hash array for fast lookup of write entries to aWBT.

aWHH: Extends aWWH with different hash lookup functions to tune locality in the write hash array.

aALL: Uses all adaptive parameters. The aWHH configuration is extended by a selective Bloom filter to speedup the lookup of write entries.

Bench.	Config.	1 thr.	2 thr.	4 thr.	8 thr.	16 thr.
Bayes	naWB	27.18	26.27	20.71	20.84	20.71
	aWBT	27.28	26.70	21.10	21.47	20.34
	aWWH	27.18	26.46	20.70	20.90	20.64
	aWHH	27.07	26.66	20.71	20.58	20.62
	aALL	27.06	26.68	20.59	20.76	20.39
Genome	naWB	12.53	6.25	3.17	1.91	2.35
	aWBT	12.21	6.31	3.14	1.86	2.24
	aWWH	12.41	6.34	3.14	1.87	2.31
	aWHH	12.24	6.35	3.08	1.85	2.34
	aALL	12.16	6.23	3.02	1.83	2.24
Intru.	naWB	42.41	24.63	14.06	11.18	11.15
	aWBT	38.37	23.09	13.03	10.30	10.41
	aWWH	37.56	23.16	13.19	10.64	10.31
	aWHH	38.20	23.03	13.09	10.35	10.45
	aALL	38.35	23.13	13.06	10.35	10.40
kmeans	naWB	112.71	83.65	52.41	36.07	35.49
	aWBT	125.25	89.18	56.88	36.88	36.77
	aWWH	126.88	88.75	58.27	39.08	36.95
	aWHH	127.11	85.04	56.22	38.35	37.37
	aALL	123.70	89.64	50.63	34.74	31.95
Labyr.	naWB	82.91	43.30	23.66	13.67	16.59
	aWBT	83.20	43.14	24.15	13.57	16.62
	aWWH	83.16	43.57	23.36	13.48	16.52
	aWHH	83.22	43.18	23.20	14.10	16.44
	aALL	83.56	43.13	23.52	13.54	16.42
SSCA2	naWB	26.11	21.90	15.90	19.36	19.57
	aWBT	26.52	22.14	15.68	19.17	19.39
	aWWH	26.61	22.24	15.79	19.18	19.15
	aWHH	26.47	21.98	15.74	19.36	19.32
	aALL	26.00	22.08	15.29	19.30	19.46
Vacat.	naWB	43.70	25.42	13.13	7.04	11.75
	aWBT	43.91	24.77	12.82	6.94	11.45
	aWWH	43.21	24.77	12.78	6.83	11.63
	aWHH	44.25	24.44	12.62	6.73	11.35
	aALL	42.57	24.55	12.70	6.76	11.39
YADA	naWB	16.18	13.09	8.96	8.57	11.30
	aWBT	15.61	12.84	8.63	8.69	10.46
	aWWH	15.36	12.79	8.65	8.51	10.27
	aWHH	15.70	12.89	8.78	8.55	10.18
	aALL	16.00	12.92	8.84	8.53	10.27

Table 5. Effect of different STM parameters, with and without adaptation, runtime in seconds.

The benchmarks in Table 5 and Figure 3 show that fine-grained thread-local adaptive tuning is able to increase performance of a competitive STM library by 4.33% on average for 16 concurrent threads, 3.39% for 8, and 3.77% 4 concurrent threads over the non-adaptive configuration. Individual benchmarks are able to increase performance by 10% over the hand-optimized non-adaptive baseline.

The adaptive configuration starts with the best mean configuration for the STAMP benchmarks and tries to improve from there. The fine-grained adaptation system checks and adapts the parameters every 64 times a thread starts a new transaction, or retries a conflicting transaction.

More threads lead to higher contention, which degrades performance for statically tuned STM libraries. An adaptive mechanism is able to adapt runtime parameters to a changing workload, reacting as soon as contention rises.

The adaptive system adds some overhead to the total processing time. So it is not surprising that the runtime for a single thread can be higher than for a system without adaptation. But as soon as the environment gets less stable (e.g., if the number of thread increases or there is some background activity), the adaptive system is able to increase performance by tuning the correct parameters.

A potential limitation of the STAMP benchmarks is that all transactions except the Labyrinth benchmark have a low average number of written locations below 30, for most benchmarks the maximum number of written locations is

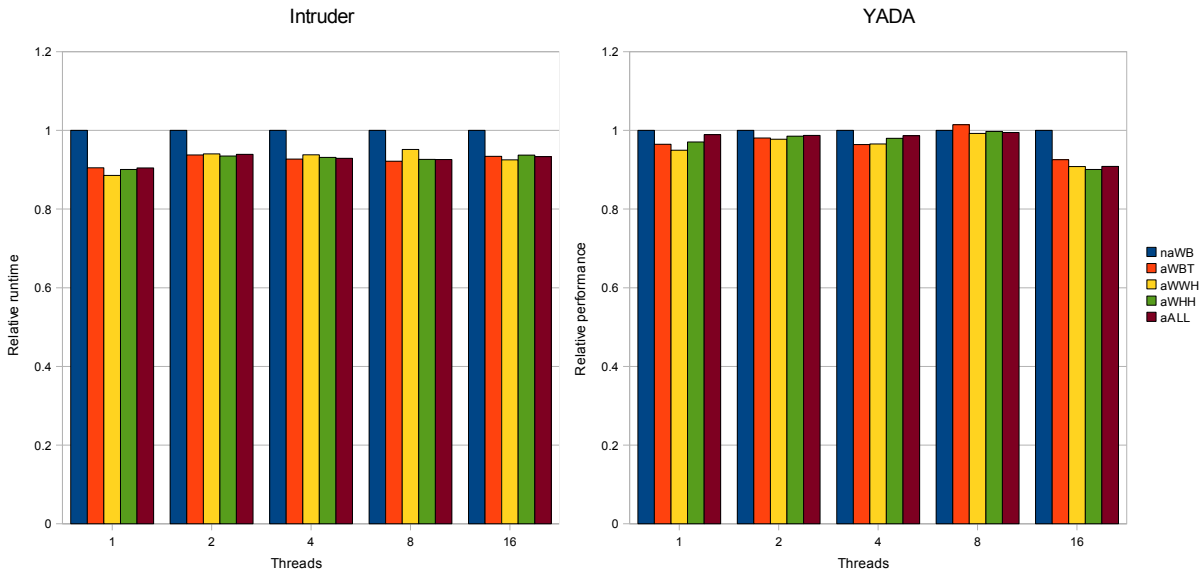

Figure 3. Effects of different STM parameters showing different adaptive configurations, runtime is relative to the non-adaptive case (lower is faster).

below 50 locations. For the STAMP benchmarks the transactional read and write accesses were instrumented manually. A compiler might emit a higher amount of transactional reads and writes due to limited context information. With the adaptation of the hash-table for write entries *adaptSTM* is able to handle a much higher number of locations that future workloads can use.

5.6 Comparison with other STM libraries

This section compares the performance of *adaptSTM* against the two fast STM systems TL2 [6], and tinySTM [10]. The comparison shows numbers for TL2 version 0.9.6, tinySTM 0.7.3, and tinySTM 0.9.9. Two versions of tinySTM were used, because the newer version is not faster than tinySTM 0.7.3 in all benchmarks.

Table 6 and Figure 4 show the competitive performance of the *adaptSTM* system using the average of 5 runs. Standard deviation is low for all benchmarks.

TL2 exited with assertion failures for some contented runs with 8 and 16 threads of the YADA benchmark, and exited with segmentation faults for all configurations of the Bayes benchmark.

If contention is high, as can be seen in Table 3 and Table 6 for the Intruder, YADA, and Vacation benchmarks then *adaptSTM* outperforms the other STM libraries. The adaptive system switches from a write-through to a write-back strategy, and the contention manager increases the time a transaction spins for a taken lock before it aborts.

As can be seen in Table 6 the adaptive system outperforms TL2 for all benchmarks except SSCA2. This suggests that the default write-through approach combined with eager-locking is faster than the lazy-locking write-back approach of TL2. Comparing *adaptSTM* to TL2 results in an average speedup of 43% for 8 threads and 137% for 16 threads.

adaptSTM is able to show better performance for both versions of tinySTM for most of the benchmarks. Especially in highly contented environments *adaptSTM* is able to adapt to the given situation. *adaptSTM* is able to outperform tinySTM 0.9.9 by 390% on average for 16 threads, or if we exclude the yada benchmark by 123% for 16 threads.

An interesting figure is the runtime for 16 threads. These numbers show the case with higher contention through concurrent programs. *adaptSTM* is able to cope with the additional contention and is still able to deliver good performance. The Intruder, kmeans, Vacation, and YADA benchmarks show how *adaptSTM* is able to handle higher contention compared to TL2 and tinySTM.

In a heavily contented environment other factors can influence the result of STM benchmarks. We attribute the performance edge of adaptSTM in contented situations to a large extend to the design decision to use an (adaptive) back-off strategy instead of retrying immediatly.

6. Concluding remarks

STM libraries must offer good performance for long-running and short-running transactions with varying transactional workloads, keeping initialization costs as well as constant overhead low. An STM system that adapts important parameters like write-methodology, size of local hash-tables, and hash functions according to runtime statistics is able to react to phase changes in the program and to speed up overall execution.

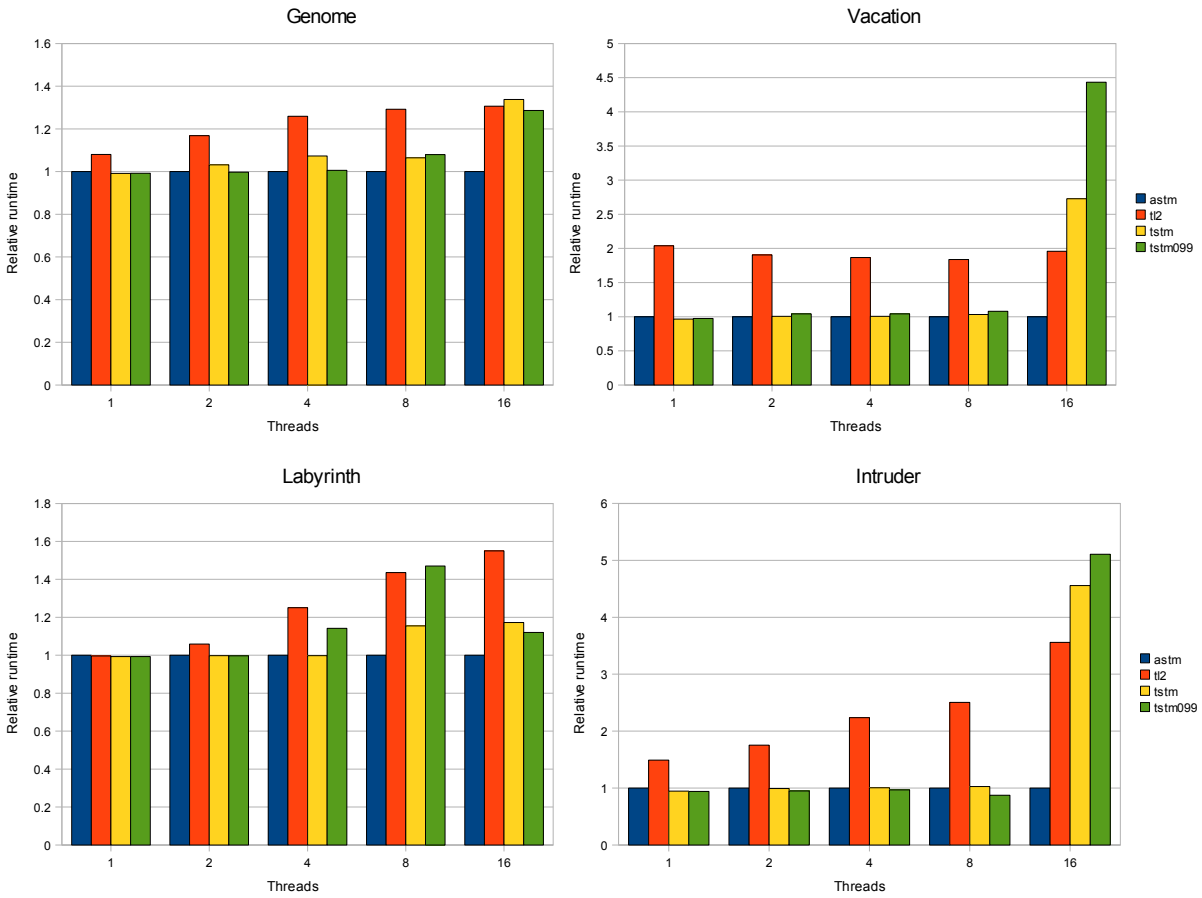

Figure 4. Comparison of adaptSTM (astm) with TL2, tinySTM version 0.7.3 (tstm), and tinySTM version 0.9.9 (tstm099). Runtime relative to adaptSTM baseline (lower is faster).

This paper presents a *fine-grained*, *thread-local adaptive*, lock-, and word-based STM library called *adaptSTM* that delivers good performance for varying workloads and transaction lengths. To our knowledge *adaptSTM* is the first STM library that uses *thread-local* adaptiveness to adapt to the behavior of the running program.

Thread-local adaptation has several advantages over global adaptation: (i) there is no need for inter-thread synchronization and barriers, (ii) threads with different workloads will adapt to their specific workloads and not to a global average, and (iii) local adaptiveness has less overhead and can therefore happen more frequently, resulting in a faster adaptation to phase changes.

For the benchmark programs, adaptSTM performs better than TL2 in all cases except SSCA2, and slightly better or slightly worse than tinySTM for low-contention scenarios; for high contention scenarios, adaptSTM performs significantly better. *adaptSTM*'s thread-local adaptation is able to improve performance by an average of 4.33% for 16 threads, and 3.39% for 8, and 3.77% for 8 threads over the hand-optimized non-adaptive case, and up to 10% for individual benchmarks. Compared to TL2 an average improvement of 43% for 8 threads and 137% for 16 threads is achieved.

Transactional Memory is an attractive platform for parallel programs, and Software Transactional Memory has attracted considerable attention. This paper demonstrates that an STM system provides numerous opportunities for optimizations and that adaptivity is an important feature of a high-performance TM system. The designers of hardware support for TM are well-advised to pay attention to lessons learned by STM users and implementors.

References

[1] AHO, A. V., LAM, M. S., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Reading, MA, 2006.

[2] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (1970), 422–426.

[3] CAO MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The*

Source code: *http://people.inf.ethz.ch/payerm/adaptSTM/*

Bench.	STM	1 thr.	2 thr.	4 thr.	8 thr.	16 thr.
Bayes	astm	27.06	26.68	20.59	20.76	20.39
	tl2					
	tstm	27.04	26.21	20.78	20.37	20.2
	t099	32.14	23.9	22.17	21.64	22.81
Genome	astm	12.16	6.23	3.02	1.83	2.24
	tl2	13.13	7.27	3.8	2.36	2.93
	tstm	12.06	6.42	3.24	1.95	3
	t099	12.06	6.21	3.04	1.97	2.88
Intru.	astm	38.35	23.13	13.06	10.35	10.4
	tl2	57.14	40.55	29.22	25.92	37.03
	tstm	36.33	22.95	13.12	10.64	47.41
	t099	36.06	21.96	12.63	9.04	53.13
kmeans	astm	123.7	89.64	50.63	34.74	31.95
	tl2	196.92	128.54	67.76	43.37	113.93
	tstm	97.11	69.71	39.44	26.86	42.24
	t099	109.29	77.83	44.35	33.08	88.24
Labyr.	astm	83.56	43.13	23.52	13.54	16.42
	tl2	83.26	45.7	29.42	19.44	25.44
	tstm	82.97	43.05	23.48	15.64	19.25
	t099	83.01	43.01	26.85	19.91	18.4
SSCA2	astm	26	22.08	15.29	19.3	19.46
	tl2	25.25	22.42	16.42	19.43	19.09
	tstm	23.88	19.56	15.07	19.57	19.73
	t099	22.76	19.25	14.89	13.18	14.38
Vacat.	astm	42.57	24.55	12.7	6.76	11.39
	tl2	86.83	46.74	23.72	12.42	22.29
	tstm	41.13	24.69	12.76	6.99	31.06
	t099	41.47	25.54	13.23	7.29	50.49
YADA	astm	16	12.92	8.84	8.53	10.27
	tl2	36.92	25.75	16.24	12.66	20.96
	tstm	14.88	12.13	8.39	9.11	296.65
	t099	14.62	12.47	9.2	9.62	350.34

Table 6. Comparison of adaptSTM (astm) with TL2, tinySTM version 0.7.3 (tstm), and tinySTM version 0.9.9 (t099). Runtime in seconds, lower is better.

IEEE International Symposium on Workload Characterization (September 2008).

[4] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking ii. In *DISC'06* (2006), pp. 194–208.

[5] DICE, D., AND SHAVIT, N. What really makes transactions faster? In *TRANSACT'06* (2006).

[6] DICE, D., AND SHAVIT, N. Understanding tradeoffs in software transactional memory. In *CGO '07* (Washington, DC, USA, 2007), pp. 21–33.

[7] DRAGOJEVIĆ, A., GUERRAOUI, R., AND KAPALKA, M. Stretching transactional memory. In *PLDI'09* (New York, NY, USA, 2009), ACM, pp. 155–165.

[8] ENNALS, R. Efficient software transactional memory. Tech. Rep. IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan 2005.

[9] FELBER, P., FETZER, C., MÜLLER, U., RIEGEL, T., SÜSSKRAUT, M., AND STURZREHM, H. Transactifying applications using an open compiler framework. In *TRANSACT'07* (August 2007).

[10] FELBER, P., FETZER, C., AND RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08* (New York, NY, USA, 2008), ACM, pp. 237–246.

[11] FRASER, K. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[12] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. *SIGPLAN Not. 38* (2003), 388–402.

[13] HARRIS, T., PLESKO, M., SHINNAR, A., AND TARDITI, D. Optimizing memory transactions. *SIGPLAN Not. 41*, 6 (2006), 14–25.

[14] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND WILLIAM N. SCHERER, I. Software transactional memory for dynamic-sized data structures. In *PODC'03* (Jul 2003), pp. 92–101.

[15] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *ISCA'93* (1993).

[16] HUDSON, R. L., SAHA, B., ADL-TABATABAI, A.-R., AND HERTZBERG, B. C. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM '06: Proc. of the 5th int. symp. on Memory management* (New York, NY, USA, 2006), ACM, pp. 74–83.

[17] KNUTH, D. E. *The Art of Computer Programming, Sorting and Searching*, vol. 3. Addison-Wesley, pub-AW:adr, 1973.

[18] LARUS, J. R., AND RAJWAR, R. *Transactional Memory*. Morgan & Claypool, 2006.

[19] MARATHE, V. J., III, W. N. S., AND SCOTT, M. L. Adaptive software transactional memory. In *DISC'05* (2005), pp. 354–368.

[20] MARATHE, V. J., AND SCOTT, M. L. A qualitative survey of modern software transactional memory systems. Tech. Rep. TR 839, University of Rochester Computer Science Dept., Jun 2004.

[21] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07* (New York, NY, USA, 2007), pp. 89–100.

[22] RAMADAN, H. E., ROY, I., HERLIHY, M., AND WITCHEL, E. Committing conflicting transactions in an stm. In *PPoPP '09* (New York, NY, USA, 2009), ACM, pp. 163–172.

[23] RIEGEL, T., FELBER, P., AND FETZER, C. A lazy snapshot algorithm with eager validation. In *DISC'06* (Sep 2006), Springer, pp. 284–298.

[24] RIEGEL, T., FETZER, C., AND FELBER, P. Snapshot isolation for software transactional memory. In *TRANSACT06* (Jun 2006).

[25] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., MINH, C. C., AND HERTZBERG, B. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06* (New York, NY, USA, 2006), ACM, pp. 187–197.

[26] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *PODC'95* (New York, NY, USA, 1995), ACM, pp. 204–213.

[27] SPEAR, M. F., DALESSANDRO, L., MARATHE, V. J., AND SCOTT, M. L. A comprehensive strategy for contention

management in software transactional memory. In *PPoPP '09* (New York, NY, USA, 2009), ACM, pp. 141–150.

[28] SPEAR, M. F., MICHAEL, M. M., AND VON PRAUN, C. Ringstm: scalable transactions with a single atomic instruction. In *SPAA'08* (New York, NY, USA, 2008), ACM, pp. 275–284.

[29] YOO, R. M., AND LEE, H.-H. S. Adaptive transaction scheduling for transactional memory systems. In *SPAA'08* (New York, NY, USA, 2008), ACM, pp. 169–178.

A. Tables and figures

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	32.49	32.1	32.07	32.89	32	32.22
2	31.98	32.01	32.09	32.12	31.99	32.03
4	31.97	32.04	32.04	32.24	32.2	32.19
6	32.13	32.04	32.15	32.05	32.04	32.2
8	32.06	32.21	32.03	32.03	32.03	32.03
10	32.13	32.22	32.32	32.11	32.01	32.12
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	24.07	25.26	24.2	24.3	24.4	25.13
2	24.92	24.37	24.59	23.24	24.48	24.2
4	24.96	24.23	25.12	25.51	23.75	25.06
6	26.08	24.19	24.8	24.45	24.95	25.08
8	26.4	27	26.44	25.22	24.02	25.7
10	27.29	27.31	25.71	26.55	28.68	26.98
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	23.25	22.02	22.81	22.44	22.01	21.68
2	21.15	22.98	22.56	21.86	22.05	22.85
4	23.08	21.39	23.2	22.99	22.63	22.02
6	22.43	21.92	23.51	22.23	22.11	22.51
8	23.65	23.66	24.17	23.82	23.85	23.26
10	24.08	23.29	24.99	24.09	23.37	25.22
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	23.07	23.23	22.74	23.24	22.83	23.46
2	23.17	25.22	21.77	23.61	22.75	21.85
4	23.52	24.76	23.4	22.57	24.29	22.16
6	25.34	24.11	23.29	24.21	26.2	23.49
8	26.41	24.1	25.23	23.66	25.91	25.9
10	26.48	26.16	26.7	25.29	25.57	25.69
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	22.1	23.38	24.45	24.09	25.53	24.05
2	22.41	21.39	22.57	23.91	22.84	23.18
4	22.81	23.36	25.94	21.88	24.02	23.17
6	24.05	24.2	24.96	23.57	23.6	22.26
8	24.75	23.68	22.97	25.93	25.55	23.46
10	24.99	24.87	26.18	23.56	24.11	25.27

Table 7. Detailed statistics for the **Bayes** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	11.5	11.71	11.81	12.18	12.28	12.26
2	11.54	11.78	11.76	12.02	12.06	11.98
4	11.5	11.7	11.61	11.72	11.67	11.79
6	11.58	11.63	11.61	11.75	11.69	11.72
8	11.64	11.6	11.68	11.63	11.7	11.61
10	11.51	11.48	11.53	11.42	11.5	11.48
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	6.55	6.92	7	7.11	7.17	7.17
2	6.67	6.65	6.8	6.72	6.69	6.76
4	6.23	6.24	6.51	6.36	6.6	6.54
6	6.65	6.31	6.33	6.27	6.47	6.44
8	6.28	6.11	6.28	6.35	6.35	6.46
10	6.39	6.47	6.31	6.51	6.47	6.37
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	3.85	3.78	3.77	3.65	3.6	3.5
2	3.67	3.48	3.42	3.28	3.28	3.2
4	3.44	3.22	3.11	3.06	3.06	3.1
6	3.34	3.19	3.19	3.07	3.01	2.98
8	3.41	3.24	3.15	3.11	3.06	3.06
10	3.48	3.4	3.44	3.42	3.45	3.46
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	2.73	2.72	2.54	2.28	2.08	2.04
2	2.46	2.22	2.12	1.99	1.93	1.9
4	2.12	2.02	1.87	1.85	1.81	1.85
6	2.09	1.99	1.93	1.8	1.78	1.8
8	2.22	2.11	1.98	1.99	2.06	1.93
10	2.24	2.14	2.12	2.26	2.23	2.2
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	2.9	2.78	2.81	2.79	2.62	2.69
2	2.8	2.74	2.57	2.56	2.51	2.46
4	2.46	2.5	2.48	2.26	2.38	2.53
6	2.49	2.42	2.39	2.22	2.2	2.2
8	2.56	2.63	2.46	2.43	2.39	2.35
10	2.84	2.71	2.57	2.63	2.61	2.52

Table 8. Detailed statistics for the **Genome** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	36.86	38.05	41.55	43.74	44.69	46.35
2	36.32	37.44	39.53	40.71	41.77	42.67
4	36.05	36.84	38.34	39.67	40.2	40.33
6	35.97	36.63	38.73	38.7	38.95	38.97
8	35.63	36.58	37.93	37.59	37.7	37.67
10	35.86	36.63	36.91	37.03	36.92	36.91
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	27.61	27.02	28.25	28.95	29.35	30.28
2	25.21	25.35	26.19	26.42	27.05	27.46
4	24.16	24.5	25	25.54	25.71	25.87
6	23.85	24.19	24.89	25	25.02	25.05
8	23.54	24.07	24.24	24.28	24.28	24.26
10	23.62	23.89	23.92	23.93	23.93	23.96
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	25.16	16.41	14.94	15.07	15.31	15.89
2	15.19	13.64	13.7	13.75	14.03	14.24
4	12.79	12.95	12.98	13.22	13.32	13.39
6	12.47	12.55	12.95	12.94	13.01	13.17
8	12.28	12.48	12.57	12.56	12.56	12.61
10	12.32	12.37	12.41	12.4	12.4	12.51
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	33.51	23.78	8.32	8.22	8.32	8.53
2	50.75	7.49	7.32	7.48	7.54	7.58
4	8.03	6.87	6.94	7.11	7.05	7.16
6	6.85	6.75	7.05	6.92	6.96	7.02
8	6.65	6.66	6.75	6.8	6.8	6.67
10	6.63	6.62	6.75	6.62	6.74	6.64
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	168.37	29.66	12.97	13.63	13.48	12.99
2	32.65	12.32	13.16	12.54	12.02	11.8
4	11.26	11.99	12.47	11.75	11.28	11.45
6	10.98	12.23	11.98	11.32	11.27	11.28
8	11.43	12.09	11.44	11.29	11.15	11.26
10	11.69	11.83	11.67	11.61	11.64	11.64

Table 9. Detailed statistics for the **Vacation** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	130.38	129.88	129.27	130.51	131.72	130.06
2	132.19	130.01	130.69	128.57	130.91	130.86
4	122.35	122.06	123.29	121.79	121.28	121.52
6	116.62	117.14	117.65	116.51	117.37	117.2
8	115.54	114.69	114.88	115.44	115.02	114.78
10	114.62	116.94	118.1	118.13	117.27	117.37
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	95.33	103.34	94.84	95.21	93.61	103.56
2	91.99	100.83	102.05	94.41	95.09	95.4
4	90.8	98.88	93.53	88.56	88.74	89.43
6	88.79	91.81	96.21	97.53	97.03	89.41
8	91.99	93.97	95.86	92.25	93.37	94.76
10	99.13	92.31	97.85	100.35	94.21	97.07
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	51.6	53.92	52.06	57.31	55.29	52.65
2	53.93	54.55	54.65	50.66	55.6	53.99
4	49.68	53.84	53.89	53.11	52	53.82
6	51.89	53.02	56.91	54.9	53.79	56.41
8	53.99	62.01	61.57	56.23	54.68	58.92
10	71.31	74.92	64.95	66.55	68.95	66.89
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	32.45	33.84	33.4	31.55	33.26	35.91
2	34.08	32.05	32.07	33.37	33.24	35.18
4	35.11	34.69	33.85	34.01	33.18	35.83
6	35.15	35.4	35.03	36.33	34.94	35.79
8	41.82	44.03	43.23	44.21	46	43.83
10	68.2	68.31	67.4	68.46	68.42	68.29
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	30.52	30.91	31.86	31.29	30.54	32.42
2	33.29	29.66	31.29	31.37	29.42	31.22
4	30.99	31.39	33.46	31.74	32.31	33.12
6	43.15	45.3	47.1	41.91	41.52	43.62
8	536.68	508.28	520.35	500.45	541.7	462.83
10	216.68	265.15	299.07	279.37	258.94	317.82

Table 10. Detailed statistics for the **kmeans** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	81.31	81.32	81.3	81.38	81.29	81.31
2	82.12	81.34	81.35	81.31	81.3	81.34
4	81.32	81.33	81.32	81.33	81.31	81.3
6	81.35	81.35	81.34	81.35	81.31	81.3
8	81.59	81.34	81.34	81.33	81.3	81.3
10	81.36	81.35	81.33	81.31	81.3	81.32
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	42.85	42.71	42.92	42.39	42.47	42.44
2	42.93	42.57	42.32	42.38	42.45	42.38
4	42.93	42.66	42.64	42.37	42.33	42.38
6	43.01	42.54	42.3	42.31	42.33	42.41
8	42.56	42.42	42.32	42.27	42.36	42.35
10	42.73	42.38	42.49	42.48	42.44	42.36
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	23.29	23.38	23.05	22.82	22.97	23.07
2	23.19	23.08	22.92	23.14	22.92	23.12
4	23.38	23.18	23.15	22.7	22.99	23.14
6	23.08	23	23.07	23.14	22.88	23.11
8	23.23	23.15	22.85	23.02	22.92	23.13
10	23.25	23.42	23.27	23.14	23.21	23.19
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	13.36	13.79	13.95	13.51	14.29	13.41
2	13.36	13.65	13.33	13.26	13.24	13.62
4	14.07	14.2	13.44	13.51	13.37	13.48
6	13.8	13.52	14.07	13.56	13.54	13.94
8	14.2	13.6	13.31	13.78	13.77	13.87
10	13.44	13.91	13.32	13.51	13.47	13.51
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	16.39	16.57	16.23	16.23	16.28	16.3
2	16.45	16.33	15.97	16.23	16.52	16.28
4	16.75	16.37	16.87	16.4	16.75	16.02
6	16.27	16.15	15.87	16.44	16.53	16.37
8	16.21	16.44	16.32	16.53	16.1	16.55
10	16.29	16.28	16.56	16.46	16.21	16.25

Table 11. Detailed statistics for the **Labyrinth** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	33.03	33.27	34.96	36.57	36.64	37.42
2	32.92	33.24	34.12	34.9	34.97	35.43
4	32.83	33.19	33.72	34.15	34.48	34.29
6	32.55	33.1	33.32	33.9	33.66	33.77
8	32.53	32.92	33.12	33.1	33.31	33.16
10	32.68	32.87	32.81	32.88	32.82	32.91
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	24.01	24.09	24.06	24	23.88	24.59
2	23.31	23.53	23.37	23.17	22.94	23.05
4	23.06	23.26	22.79	22.66	22.55	22.9
6	23.17	23.24	23.1	23.02	22.98	23
8	23.2	23.28	23.22	23.22	23.18	23.22
10	23.24	23.37	23.4	23.35	23.39	23.3
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	15.04	14.26	14.15	14.03	14.04	14.13
2	13.61	13.56	13.48	13.19	13.03	13.37
4	13.32	13.41	13.26	13.14	13.17	13.09
6	13.26	13.4	13.27	13.22	13.19	13.25
8	13.54	13.23	13.6	13.56	13.3	13.57
10	18.52	18.58	18.6	18.83	18.64	18.62
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	12	11.65	11.42	11.17	11.09	11.17
2	11	10.81	10.69	10.51	10.34	10.46
4	10.51	10.38	10.38	10.17	10.2	10.22
6	10.3	10.22	9.99	10.09	10.08	10.02
8	11.07	10.79	10.69	10.73	10.89	10.73
10	17.06	17.23	16.95	17.17	17.09	16.81
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	23.47	16.13	11.78	11.08	11.02	10.93
2	14.07	11.27	10.64	10.44	10.56	10.21
4	13.77	11.23	10.76	10.57	10.07	9.98
6	11.49	10.63	10.45	10.25	10.1	10.01
8	12.31	12.37	12.04	11.81	11.95	11.75
10	13.71	13.1	12.59	12.46	12.64	12.71

Table 12. Detailed statistics for the **Intruder** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	22.76	23.06	23.8	24.36	24.63	24.98
2	22.74	23.15	23.63	24.02	24.27	24.45
4	22.87	23.26	23.64	24.07	24.14	24.31
6	22.93	23.31	23.51	23.85	23.9	23.89
8	22.91	23.18	23.43	23.57	23.58	23.61
10	22.92	22.99	23.2	23.06	23.27	23.21
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	21.5	21.42	21.48	21.62	21.93	22.28
2	21.45	21.45	21.6	21.75	21.75	22.16
4	21.43	21.49	21.45	21.58	21.74	21.98
6	21.45	21.44	21.62	21.48	21.61	21.54
8	21.44	21.66	21.36	21.39	21.39	21.4
10	21.56	21.27	21.23	21.21	21.33	21.16
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	14.86	15.01	14.84	14.87	14.97	15.25
2	14.9	14.88	14.8	14.82	14.95	15.14
4	14.84	14.49	14.85	14.98	14.95	15.03
6	14.88	14.84	14.86	14.88	14.9	14.65
8	14.88	14.84	14.89	14.81	14.97	14.88
10	14.82	14.8	14.83	14.76	14.79	14.82
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	19.11	18.82	18.82	18.77	18.87	18.91
2	18.96	18.9	18.94	18.87	18.99	19.05
4	19.1	19.14	18.91	18.99	18.96	18.91
6	19.21	19.07	19.01	19.06	19.17	19.18
8	19.12	18.95	19.05	19.08	18.97	19.11
10	19.22	19.16	19.13	19.13	19.03	19.07
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	19.98	18.97	18.4	18.54	18.83	18.94
2	18.9	18.34	18.59	18.79	19.06	19.03
4	18.71	18.55	18.9	19.08	19.06	19.03
6	18.68	18.71	18.85	18.9	18.97	18.93
8	18.53	18.51	18.48	18.58	18.43	18.42
10	19.37	19	18.93	18.98	18.89	19.07

Table 13. Detailed statistics for the **SSCA2** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.

1 Thread	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	14.93	15.11	15.79	16.18	16.3	16.38
2	14.83	15.11	15.34	15.44	15.52	15.6
4	14.69	14.79	15.03	15.07	15.18	15.2
6	14.67	14.83	14.87	14.89	14.91	15.02
8	14.57	14.66	14.72	14.69	14.7	15.03
10	14.52	14.67	14.67	14.63	14.66	14.67
2 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	17.5	14.69	13.74	13.53	13.55	13.5
2	14.18	13.36	13.02	12.98	12.98	12.96
4	13.17	12.79	12.69	12.68	12.67	12.74
6	13.66	13.61	13.5	13.5	13.54	15.59
8	14	14.14	14.53	13.77	13.89	14.02
10	14.88	16.57	14.67	14.95	16.1	15.85
4 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	19.61	11.21	9.63	9.18	8.99	8.99
2	10.97	9.34	8.84	8.45	8.8	8.71
4	9.37	8.81	8.72	8.59	8.53	8.65
6	9.91	9.62	9.51	9.54	9.51	9.35
8	10.02	10	10.21	10.01	9.91	9.94
10	11.03	11.66	17.05	10.83	46.49	11.3
8 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	23.06	11.64	9.99	8.89	8.47	8.63
2	11.56	9.29	8.44	8.51	8.61	8.28
4	9.96	8.7	8.54	8.55	8.61	8.19
6	10.29	9.35	9.32	9.42	10.23	9.27
8	10.4	10.18	9.79	9.81	9.73	9.53
10	10.75	10.58	20.17	10.66	10.56	10.82
16 Threads	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}	2^{26}
0	N/R	28	15.32	12.23	11.07	10.95
2	27.83	14.64	11.53	11.05	10.69	10.72
4	15.01	11.45	11.15	10.78	10.69	10.39
6	12.3	10.93	10.67	10.57	10.53	10.33
8	13.81	13.08	12.37	12.46	12.34	12.6
10	18.95	18.68	18.41	17.3	19.75	26.67

Table 14. Detailed statistics for the **YADA** benchmark showing different global hashtable parameters. The hashtable size varies from 2^{16} to 2^{26}, and the number of hash bits varies from 0 to 10. The benchmarks were run for 1 to 16 threads.