Performance Evaluation of Adaptivity in Software Transactional Memory

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. We explore and analyze several optimization opportunities to adapt STM parameters to a running program.**

This paper uses adaptSTM, a flexible STM library with a non-adaptive baseline common to current fast STM libraries to evaluate different performance options. The baseline is extended by an online evaluation system that enables the measurement of key runtime parameters like read- and write-locations, or commit- and abort-rate. The performance data is used by a thread-local adaptation system to tune the STM configuration. The system adapts different important parameters like write-set hash-size, hash-function, and write strategy based on runtime statistics on a per-thread basis.

We discuss different self-adapting parameters, especially their performance implications and the resulting trade-offs. Measurements show that local per-thread adaptation outperforms global system-wide adaptation. We position local adaptivity as an extension to existing systems.

Using the STAMP benchmarks, we compare adaptSTM to two other STM libraries, TL2 and tinySTM. Comparing adaptSTM and the adaptation system to TL2 results in an average speedup of 43% for 8 threads and 137% for 16 threads. adaptSTM offers performance that is competitive with tinySTM for low-contention benchmarks; for high-contention benchmarks adaptSTM outperforms tinySTM.

Thread-local adaptation alone increases performance on average by 4.3% for 16 threads, and up to 10% for individual benchmarks, compared to adaptSTM without active adaptation.

I. Introduction

Transactional memory offers an attractive alternative to locks by executing critical sections in a transactional manner. The programmer specifies *atomic* sections and transactional memory accesses. The runtime system ensures mutual exclusion of the atomic sections. Transactional memory can be implemented in software (STM), in hardware (HTM) or as a hybrid approach (HyTM).

Many current transactional memory systems are implemented in software. The advantage of STM systems is that different algorithms and parameters can be evaluated 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 assembly language implementation).

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

This paper uses adaptSTM, a flexible and neutral STM library that implements a non-adaptive baseline common to many fast STM libraries. This non-adaptive baseline is extended by a novel online sampling system that collects different performance data. The performance data is then used in a thread-local adaptation system that tunes important STM parameters for each program phase to the workload of each individual thread.

The contributions of this paper are:

- A fine-grained, thread-local, low overhead adaptation mechanism that adapts STM parameters according to detected phase changes and different thread workloads.
- A detailed analysis of the presented thread-local self-adapting parameters and the trade-offs between collecting performance data and available adaptivity.

The reminder of the paper is organized as follows. Section II presents design decisions for a non-adaptive and competitive STM baseline. Section III extends the non-adaptive baseline STM library by an adaptive optimization system and explains different thread-local optimization strategies. Section IV evaluates different STM alternatives, the different adaptive parameters, explains trade-offs between different optimization systems, offers insights into efficient optimization systems, and justifies the advantages of local adaptivity compared to global adaptivity. Section V lists related work, explains the background of modern STM systems, and highlights design decisions for efficient transactional memory. Section VI concludes the paper.

II. Efficient STM baseline

Every transaction needs local 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.

Most STM systems use arrays that are expanded if the capacity is reached and combine *lock-set* and *write-set*. This

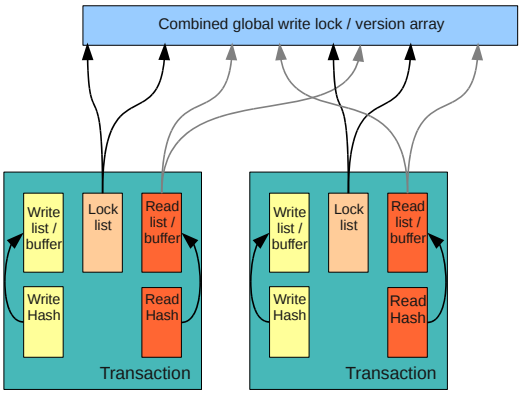

Figure 1. Transaction-local data, including read-set, lock-set and write-set.

simple approach has limited scalability. 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.

A platform to evaluate adaptivity in STM systems must build on a fast, competitive, and non-adaptive baseline implementation. The implementation of the non-adaptive adaptSTM baseline makes the following design decisions:

Global versions: adaptSTM uses a single static global array for versions and locks. Experiments with the STAMP [2] benchmarks show that a good average size for the locking table is 2^{22}`*sizeof(word)`, and a shift by 5 bits (`((addr>>5) & (HASH_PATTERN))`) offers a good tradeoff between fine-grained locking and additional overhead for handling too many locks.

Locking strategy: adaptSTM implements both *eager locking* and *lazy locking* for write locations, but prefers eager locking to enable an adaptive selection between *write-back* and *write-through* write strategies.

Cache locality: transactions use separate arrays for read-sets, lock-sets, and write-sets. If the number of locations exceeds a specific threshold a hash-table is built on-the-fly to enable faster lookup. Our experiments with the STAMP benchmarks show that a hash-table is not beneficial for read-sets and lock-sets. For write-sets a hash-table for 32 entries is best on average, but the optimal size varies greatly between benchmarks.

Read-set revalidation: 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 tries to *update 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 is updated to the current global time and the transaction continues.

Contention management: for contended transactions adaptSTM implements a wait and retry strategy. The current transaction is yielded a configurable number of times. The yield operations give the other threads the opportunity to release the lock before the current transaction has to abort itself.

III. Adaptivity in STM

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

The adaptation system uses the information provided by the online sampling system to justify a deliberate decision. Adaptation can be global (for all threads), or thread-local, whereas both forms of adaptation have their advantages.

Important sampled 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. These thread-local counters are measured using a moving window implementation in the transaction library.

Our adaptation system uses the runtime data to select between different write strategies, to adapt the local write-set hash-function, and to tune the locality of the write-set's hash-table.

A. Adaptivity in current systems

Adaptivity is a challenging property for dynamic systems. It enables an STM system to adapt to different workloads as well as to phases during the runtime of a program. Adaptivity faces two challenges: (1) it must find a good configuration that outperforms the non-adaptive baseline and (2) it must find that configuration fast and with low overhead.

Different forms of adaptivity already exist in current systems. Marathe et al. developed ASTM [17], an object-based STM algorithm that extends DSTM [13] and adapts (i) lazy and eager lock acquire strategies, and (ii) two forms of meta-data for transactional objects. TinySTM [8, 9] showed that an eager acquire strategy is better for lock-based STMs because of the earlier conflict detection. TinySTM [9] is the first lock-based STM that uses *global* optimizations to adapt the size and the hash function of the global locking table.

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

adaptSTM [20] on the other hand uses eager-locking, sampling, and local adaptivity to adapt different *thread local* parameters (e.g., hash-table size for write-sets, locality tuning for thread-local hash-tables, selecting between write-back and write-through strategies, and adaptive contention management). adaptSTM takes the idea from other adaptive STM systems and extends adaptivity to thread-local parameters using a wide selection of different adaption options.

B. Local vs. global adaptivity

There are two approaches to adaptivity. One is global adaptivity, which changes the parameters for all running transactions. The other is local adaptivity, which changes the parameters 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 optimizes the transactional parameters for best read performance and a writer-thread optimizes for write throughput. *Global adaptivity* is 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 without synchronization overhead every time a transaction is started or restarted.

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.

C. Write-back vs. write-through

Any STM library must buffer transactional writes to ensure correctness. A transaction either buffers the written locations locally and writes the data back as soon as the transaction is in the commit phase (*write-back* or lazy-update). Alternatively the transaction writes the data directly to memory and caches the original value locally (*write-through* or eager-update).

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.

In a contended 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 adaptation system uses the average of the last 64 transactions to calculate the abort rate.

D. 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, then 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 of adaptSTM hash collisions are queued in a linked list in the same hash-table slot.

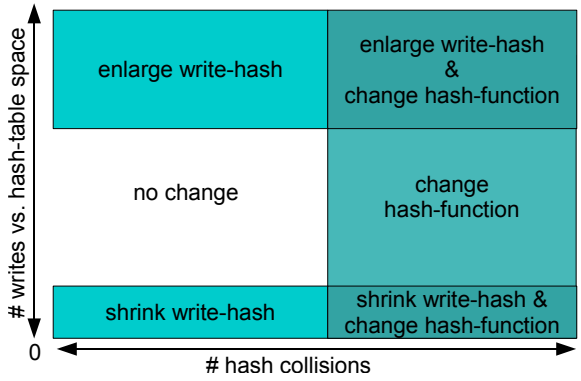

Figure 2. Summary of the hash-table adaptation mechanism, depending on the number of hash collisions and the load in the hash-table.

The adaptation 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 are illustrated in Figure 2. The data are obtained using the STAMP benchmarks and offer a good tradeoff between hash-table size and hash-table collisions.

E. Locality tuning for thread-local 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 is higher than the potential benefit from a better distribution. Simple hash-functions are not able to cover all data access patterns. The adaptation system switches between different simple hash-functions for the thread-local write-set and selects the hash-function with a low number of hash collisions. The list of hash-functions is:

- `(addr<<X) & SIZE`, whereas `X` is 2, 4, 5, 6, or 8.
- `((addr<<16)&SIZE)^((addr<<5)&SIZE)`
- `((addr<<12)&SIZE)^((addr<<2)&SIZE)`

The choice of the hash-function is a trade-off between quality and speed. These seven hash-functions cover a wide range of different data-access patterns and the number is still low enough to provide a quick turn-around between the different hash-functions if the number of hash collisions is high.

F. 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 that retries immediately if the contention is low, or yields an increasing amount of times in contended situations.

G. 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 baseline collects information about the number of unique write locations, as well as the overall number of write and read locations. The overhead to add additional counters for the number of started transactions, committed transactions, and aborted transactions is reasonable, 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.

IV. Benchmarks and evaluation

This section presents an evaluation of different optimizations, adaptive parameters, and performance compared to other STM systems based on the STAMP [2] benchmark suite version 0.9.10.

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 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 analysis shows measurements for 1, 2, 4, 8, and 16 threads. The measurements for 16 threads represent a contended situation where always two threads share one core. The results for 16 threads show how well a system handles contention.

The next sections present an analysis of different STAMP characteristics, an evaluation of non-adaptive optimizations like the read-set extension, implications of adaptive parameters, and evaluate performance compared to tinySTM and TL2 to provide a reference to other published systems.

A. STAMP characteristics

The STAMP benchmarks cover a wide range of transactional programs. The workload configuration for the benchmarks in this paper is available in Table I.

Benchmark	Parameters
Bayes	`-v32 -r4096 -n10 -p40 -i2 -e8 -s1`
Genome	`-g16384 -s64 -n16777216`
kmeans	`-m25 -n15 -t0.00001 -i random-n65536-d32-c16.txt`
Labyrinth	`-i random-x512-y512-z7-n512.txt`
Vacation	`-n4 -q60 -u90 -r1048576 -t4194304`
Intruder	`-a10 -l128 -n262144 -s1`
SSCA2	`-s20 -i1.0 -u1.0 -l3 -p3`
YADA	`-a15 -i ttimeu1000000.2`

Table I
WORKLOAD CONFIGURATIONS FOR THE INDIVIDUAL STAMP BENCHMARKS. THE PARAMETERS ARE IDENTICAL TO THE ++ PARAMETER SET IN THE STAMP EVALUATION [2] EXCEPT FOR KMEANS WHERE WE REPLACED *-m15* BY *-m25* TO INCREASE THE WORKLOAD.

Table II shows that the STAMP benchmarks are challenging for STM systems. The total number of transactions ranges from about 1,546 for Bayes to up to 23,428,126 for Intruder and the average transactional footprint varies as well. 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).

Bench.	commits	locks	reads			writes		
			avg.	min.	max.	avg.	min.	max.
Bayes[a]	33	1	11	1	26	1	1	2
	1,513	2	23	1	423	2	0	20
Genome	2,489,218	0	36	1	4,154	0	0	24
Intruder	23,428,126	1	23	3	875	1	0	47
kmeans[b]	87,382	24	24	1	33	24	1	33
Labyrinth	1,026	177	180	3	844	177	0	844
SSCA2	22,362,279	1	1	1	1	1	1	2
Vacation	4,194,304	7	394	14	1,807	7	0	79
YADA	2,415,298	13	58	0	1,320	16	0	331

[a] Bayes executes two different sequential STM runs.
[b] kmeans has 720 equal sequential runs.

Table II
STAMP CHARACTERISTICS SHOWING COMMITS, LOCKS, AVG., MIN/MAX READS, AND AVG., MIN/MAX WRITES.

The varying load poses two challenges to the adaptation system. First, the adaptation system must find a good configuration. Second, the adaptation system must find the configuration fast and with tolerable overhead.

B. Evaluation of the read-set extension

Another factor that is important for competitive performance is the read-set extension optimization discussed in Section II. Contention and read version failures rise with the number of concurrently running transactions. As contention increases other threads will increase versions of locks, creating conflicts with the current transaction. It is important to reduce the amount of unnecessary retries.

Bench.	2 threads	4 threads	8 threads	16 threads
Bayes	17	23	27	27
	47%	57%	41%	52%
Genome	1,346	3,350	8,945	10,611
	17%	53%	52%	38%
Intruder	536,451	1,436,177	8,527,748	6,656,061
	96%	84%	75%	80%
kmeans	175,525	569,984	1,682,853	1,535,232
	100%	100%	100%	100%
Labyrinth	19	46	110	237
	100%	100%	100%	100%
SSCA2	33	106	57	149
	100%	100%	100%	100%
Vacation	1,997	4,685	8,962	6,923
	92%	90%	89%	82%
YADA	138,791	280,942	417,134	408,544
	95%	94%	91%	79%

Table III
NUMBER OF READ-SET VALIDATIONS PER BENCHMARK FOR A SPECIFIC NUMBER OF THREADS AND PERCENTAGE OF SUCCESSFUL VALIDATIONS AND READ-SET EXTENSIONS.

Bench.	2 threads	4 threads	8 threads	16 threads
Bayes	1,476	1,392	1,415	1,323
	1%	1%	1%	2%
Genome	2,489,218	2,489,220	2,489,220	2,489,228
	0%	0%	0%	2%
Intruder	23,428,127	23,428,129	23,428,133	23,428,141
	1%	5%	24%	18%
kmeans	3,844,852	3,844,940	3,932,505	3,932,865
	1%	3%	10%	9%
Labyrinth	1,028	1,032	1,040	1,056
	2%	4%	10%	20%
SSCA2	22,362,283	22,362,287	22,362,295	22,362,315
	0%	0%	0%	0%
Vacation	4,194,304	4,194,304	4,194,304	4,194,304
	0%	0%	0%	0%
YADA	2,495,029	2,544,550	2,581,206	2,574,275
	8%	12%	21%	383%

Table IV
NUMBER OF COMMITS PER BENCHMARK FOR A SPECIFIC NUMBER OF THREADS AND RELATIVE PERCENTAGE OF RETRIES.

Instead of aborting, adaptSTM tries to extend the read-set by (re-)validating all previous reads.

Table III 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.

C. Commit and retry rates

An important performance parameter to measure contention is the retry rate as a function of the number of successful commits. Contention will increase with the number of threads and result in more retries.

As shown in Table IV, 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 [19] and the callgrind tool shows for the Bayes benchmark that *malloc* and *free* consume more than 40% of the time.

An external limitation is the memory allocation system. The glibc memory allocator uses locks to ensure mutual exclusion. As the number of threads increases the calls to the memory allocator lead to an unwanted synchronization and linearization. This limitation is removed if a lock-free allocator or an STM-aware memory allocator is used [25].

D. Design decisions for the global locking table

Two parameters can be changed for the global locking table: (i) the size of the table, 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 locking table 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 locking table is mainly responsible for the initialization cost during the STM startup.

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. The stride covered by the hash-function should be large enough to cover the data structure and small enough so that it does not hinder concurrent access between data structures that are next to another. Table V shows the different interaction patterns. Our experiments indicate that more complex hash-functions with more instructions result in too much overhead for which the better hash distribution does not compensate.

The expected result for different configurations of the locking table 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 use STAMP and adaptSTM in the default configuration without adaptation to evaluate different combinations of number of shift bits and the table size to

# 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 V
DIFFERENT CONFIGURATIONS FOR HASH-FUNCTIONS

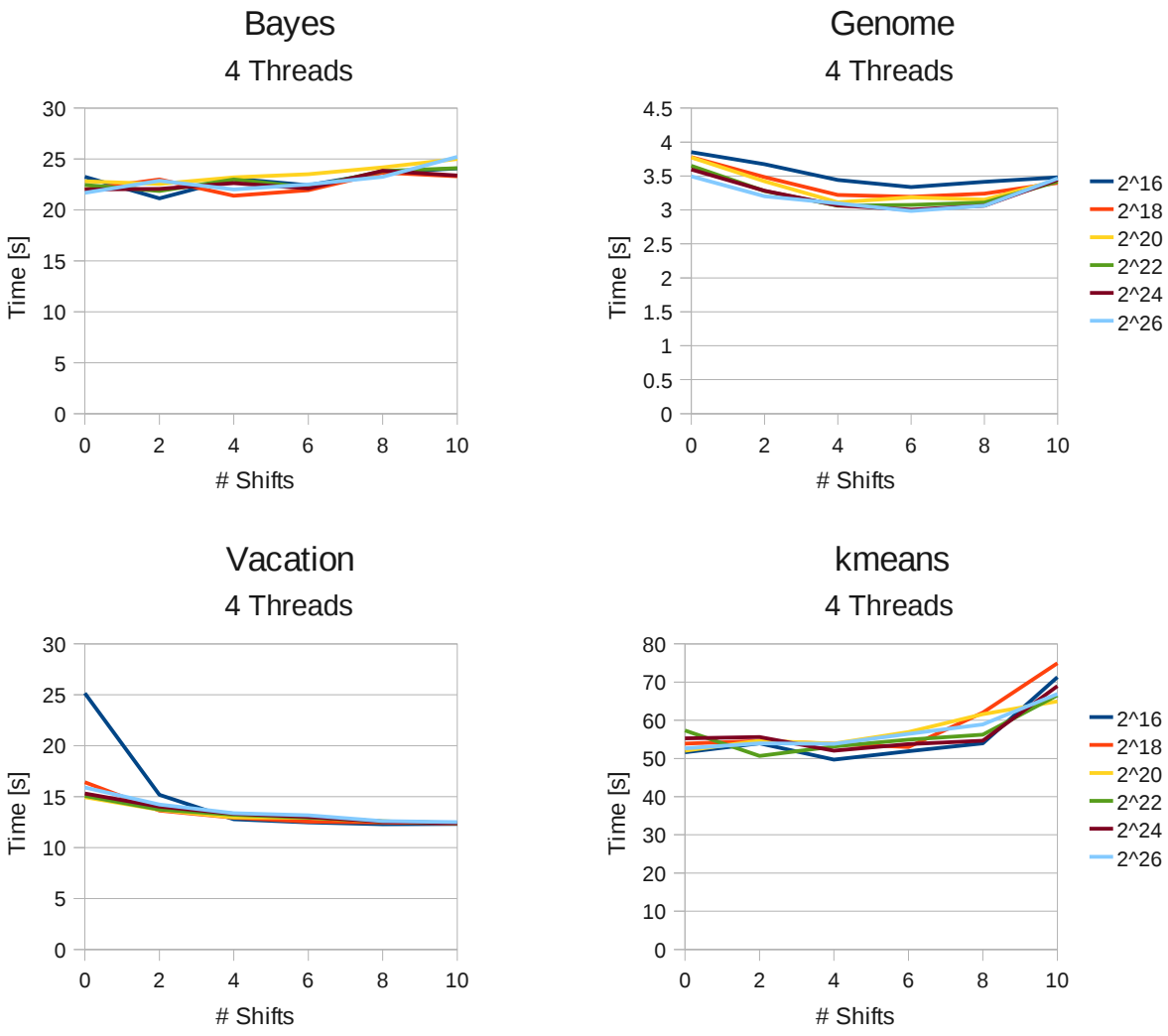

Figure 3. Execution time for different locking table sizes and shifting hash-functions, first 4 STAMP benchmarks, 4 threads.

reason about the variance in these parameters. Figure 3 compares locking table sizes from 2^{16} to 2^{26} entries and from 0 to 10 shift bits and shows averages for 4 threads of 5 runs for Bayes, Genome, Vacation, and kmeans. The results for the remaining STAMP benchmarks are similar and not shown here because of space limitations. The standard deviation was low for most runs, except some corner cases with a large number of shifts.

Except for the Genome benchmark, the size of the locking table has no influence on the runtime of the benchmark. Only for tiny tables and a low number of shifts (0, or 2) the table size has a noticeable effect because of the larger amount of locks a transaction must hold. Larger locking tables lead to better performance in the Genome benchmark with diminishing returns after 2^{20} entries. The results show that any table size of 2^{20} to 2^{24} entries is reasonable, and there is no need to adapt the size of the locking table at runtime. The number of shifts must also be smaller than 8, otherwise benchmarks like Genome, kmeans, Intruder, and YADA will experience overlocking.

Reasonable choices of the locking table size and a reasonable number of shift bits result in low changes of the program behavior. Runtime differences are not dominant and global adaptivity is not needed for locking table sizes and number of shifts. Adaptivity is not beneficial in all cases, therefore adaptSTM uses a fixed locking table size of 2^{22} entries and a fixed number of 5 hash bits.

E. Breakdown of the adaptive parameters' effectiveness

Depending on the workload of the program different adaptive parameters result in optimal throughput. This section analyzes the different thread-local adaptive parameters and their contribution to the overall result.

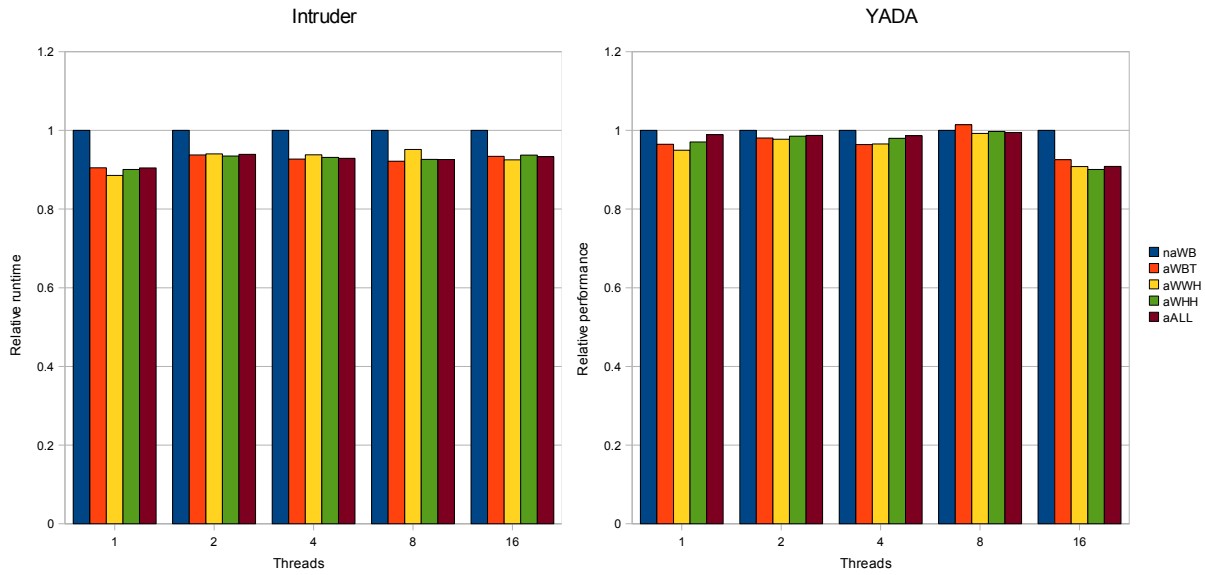

Figure 4. Effects of different STM parameters showing different adaptive configurations, runtime is relative to the non-adaptive case (lower is faster).

The adaptation process and online optimization is stable and converges to optimal results quickly, leading to a low runtime standard deviation. Figure 4 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 thread-local adaptation, offering dynamic configuration of the write strategy (write-back or write-through), and an exponential drop-off in the waiting time for contended 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 [1] to speedup the lookup of write entries.

Figure 4 shows that fine-grained thread-local adaptive tuning increases performance of a STM library by 4.3% on average for 16 concurrent threads, 3.4% for 8, and 3.8% for 4 concurrent threads over the non-adaptive configuration. Individual benchmarks show performance improvements of up to 10% compared to the non-adaptive baseline.

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

Thread-local adaptation enables an STM system to adapt to changing situations like higher contention through concurrent threads, program phases, or different workloads. The parameters are adapted as soon as the changed workload is detected. Thread-local adaptation relies on an exact online sampling system to offer a broad optimization spectrum and to enable better performance for STM libraries.

The adaptation system adds some overhead to the total processing time. 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., the number of threads increases or there is some background activity due to concurrent tasks), the adaptation system increases performance by tuning the correct parameters.

F. Comparison with other STM libraries

This section compares the performance of adaptSTM with adaptivity against two fast STM systems: TL2 [5] version 0.9.6, and tinySTM [9] version 0.7.3 and 0.9.9. Two versions of tinySTM were used, because the newer version is slower than tinySTM 0.7.3 for some STAMP benchmarks.

Table VI and Figure 5 show the competitive performance of the adaptSTM system using the average of 5 runs. Standard deviation is low for all benchmarks. TL2 did not complete for some contended runs with 8 and 16 threads of the YADA benchmark and was unable to run the Bayes benchmark.

Combining Table VI with contention information from Table IV shows that adaptSTM outperforms the other STM libraries for the Intruder, YADA, and Vacation benchmarks if contention is high. The adaptation system switches from a write-through to a write-back strategy, and the contention manager increases the time a transaction spins for a taken

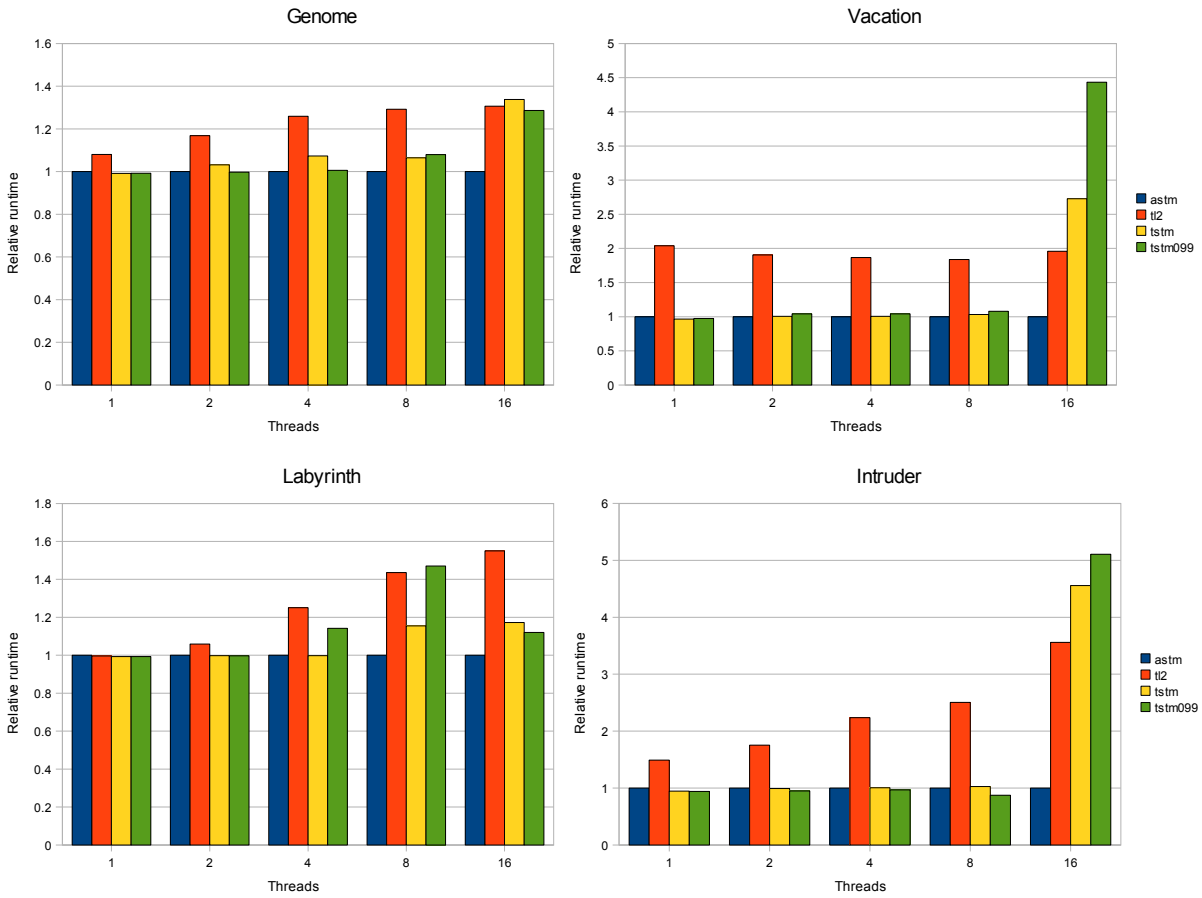

Figure 5. Comparison of adaptSTM with TL2, tinySTM version 0.7.3 (tstm), and tinySTM version 0.9.9 (tstm099). Runtime relative to adaptSTM baseline (lower is faster).

lock before it aborts.

In Table VI adaptSTM 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 shows better performance for both versions of tinySTM for most of the benchmarks. Especially in highly contended environments adaptSTM adapts to the given situation. adaptSTM outperforms 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 result is the runtime for 16 threads. These numbers show the case with higher contention through concurrent programs. adaptSTM copes with the additional contention and still delivers good performance. The Intruder, kmeans, Vacation, and YADA benchmarks show how adaptSTM handles higher contention compared to TL2 and tinySTM.

In a heavily contended environment other factors influence the result of STM benchmarks. We attribute the performance edge of an adaptive system in contended situations to a large extent to the design decision to use an (adaptive) backoff strategy instead of retrying immediately and to the dynamic switch from a write-through to a write-back strategy for transactional writes.

V. Related work

The book *Transactional Memory* [16] by Larus and Rajwar gives an overview about the history of many TM systems. It shows different STM designs and includes implementation details.

Early STM systems [10, 11, 13, 18, 26] evolved out of limited hardware TM systems [14]. 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/versioning table [4, 6, 7, 12, 22, 25, 27] and a hash-function to distribute the available locks over the

Bench.	STM	1 thr.	2 thr.	4 thr.	8 thr.	16 thr.
Bayes	astm	27	27	21	21	20
	tl2					
	tstm	27	26	21	20	20
	t099	32	24	22	22	23
Genome	astm	12	6.2	3.0	**1.8**	**2.2**
	tl2	13	7.3	3.8	2.4	2.9
	tstm	12	6.4	3.2	1.9	3.0
	t099	12	6.2	3.0	2.0	2.9
Vacation	astm	43	25	13	**6.8**	**11**
	tl2	87	47	24	12	22
	tstm	41	25	13	7.0	31
	t099	41	26	13	7.3	50
kmeans	astm	124	90	51	35	**32**
	tl2	197	129	68	43	114
	tstm	97	70	39	27	42
	t099	109	78	44	33	88
Labyrin.	astm	83	43	24	**14**	**16**
	tl2	83	46	29	19	25
	tstm	83	43	23	16	19
	t099	83	43	27	20	18
Intruder	astm	38	23	13	10	**10**
	tl2	57	41	29	26	37
	tstm	36	23	13	11	47
	t099	36	22	13	9.0	53
SSCA2	astm	26	22	15	19	19
	tl2	25	22	16	19	19
	tstm	24	20	15	20	20
	t099	23	19	15	13	14
YADA	astm	16	13	8.8	**8.5**	**10**
	tl2	37	26	16	13	21
	tstm	15	12	8.4	9.1	297
	t099	15	12	9.2	9.6	350

Table VI
COMPARISON OF ADAPTSTM (ASTM) WITH TL2, TINYSTM VERSION 0.7.3 (TSTM), AND VERSION 0.9.9 (T099). RUNTIME IN SECONDS, LOWER IS BETTER. BOLD ENTRIES HIGHLIGHT CONFIGURATIONS WHERE ADAPTSTM SIGNIFICANTLY OUTPERFORMS OTHER IMPLEMENTATIONS.

complete memory region. 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. *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. *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.

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.

Another important criterion is the use of a global clock [3, 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. Versioning 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. Hudson et al. [15] studied the effects of concurrent transactional memory allocation and proposed an alternative memory allocator for STM systems.

Two current fast lock-based STM systems with available source code are TL2 [3, 5] and tinySTM [8, 9]. Both systems use a combined lock/version-array and a global clock. TinySTM is based on a single-version, word-based, eager-locking variant of the lazy snapshot algorithm [23]. TL2 on the other hand uses lazy locking. TinySTM and TL2 are used to compare between different design decisions and optimizations for adaptSTM.

The performance impact for many different parameters and optimization strategies was already measured. Most of the related work favors static optimization and static tuning of global parameters [4, 5, 7, 12, 21, 30]. Static tuning is a key for an efficient baseline, but an important property is to adapt to different workloads. This can only be achieved with dynamic adaptive tuning [9, 17, 20, 29].

VI. CONCLUDING REMARKS

This paper presents a detailed evaluation of different STM parameters and how the program behavior and performance reacts to changes of these parameters. A detailed analysis describes reasonable parameters and argues for a fixed locking table size and a fixed number of shift bits in the hash-function. Reasonable preconfigured values outperform the global adaptive mechanism in all investigated cases.

The analysis of the data provided by the online adaptation system leads to three different thread-local adaptive optimizations. The hash-function and the size of the write-set's thread-local hash-table are changed adaptively, the write-strategy is adaptively changed from write-through to write-back, and the contention manager uses an exponential backoff strategy depending on the current contention.

Coarse-grained adaptation is an interesting and powerful tool that extends fine-grained adaptation. It is important to direct the effort to those parameters of the adaptation system that significantly contribute to overall performance. Adaptivity improves performance but the effects of individual parameters must be measured carefully because adaptivity does not pay off everywhere.

The presented system can be used for further analysis of different adaptation parameters. Additionally the generated statistics can be used for the analysis and characterization of challenging workloads.

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.

Source code is available at: http://people.inf.ethz.ch/payerm/adaptSTM/

REFERENCES

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

[2] CAO MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08* (Sept. 2008).

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

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

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

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

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

[8] 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).

[9] 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.

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

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

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

[13] 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.

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

[15] HUDSON, R. L., SAHA, B., ADL-TABATABAI, A.-R., AND HERTZBERG, B. C. McRT-Malloc: a scalable transactional memory allocator. In *ISMM'06* (New York, NY, USA, 2006), ACM, pp. 74–83.

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

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

[18] 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.

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

[20] PAYER, M., AND GROSS, T. adaptSTM - an online fine-grained adaptive stm system. Tech. rep., ETH Zurich, http://www.lst.ethz.ch/research/publications/ADAPTSTM_2010.html, 2010.

[21] PORTER, D. E., AND WITCHEL, E. Understanding transactional memory performance. In *ISPASS'2010* (March 2010), pp. 97–108.

[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 *TRANSACT'06* (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.

[30] ZYULKYAROV, F., STIPIC, S., HARRIS, T., UNSAL, O. S., CRISTAL, A., HUR, I., AND VALERO, M. Discovering and understanding performance bottlenecks in transactional applications. In *PACT '10* (New York, NY, USA, 2010), ACM, pp. 285–294.