fastSTM - a userspace library for software transactional memory

Olivier Saurer

Mathias Payer
Responsible assistant

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

December 2008
Abstract

We present fastSTM, a modular word-based software transactional memory library, which supports commit-time and encounter-time locking. Validation of the read- and write-set is done using a global version-clock. To reduce cache misses we store read- and write-entries together in small blocks of multiple cache line size. For fast membership test in the read- and write-set a bloom filter is used.

The modularity of our system allows researchers to prototype and experiment with new algorithms without the need to design a complete STM system from scratch. We show that our buffer system, provides up to 20% fewer cache misses than other state-of-the-art software transactional memory systems. We also show that our design provides less contention on the locks compared to other implementations. In most cases our design scales better with the increasing number of threads, it though suffers from a constant time overhead.
Contents

1 Introduction 1
  1.1 A Very Short Introduction to Software Transactional Memory 1
  1.2 Motivation 1

2 Related Work 3
  2.1 Address Based Software Transactional Memory 3
  2.2 Different Approaches To STM 3
    2.2.1 tinySTM 4
    2.2.2 Transactional Locking 2 4
  2.3 Bloom Filter 4

3 Design 7
  3.1 Overall Design Requirements 7
  3.2 fastSTM Modules 8
    3.2.1 fastSTM 8
    3.2.2 Read Write Buffer 8
    3.2.3 Contention Manager 9
    3.2.4 Lock 9
    3.2.5 Roll over Clock 9
  3.3 Composed Transactions 9

4 Implementation 11
  4.1 fastSTM 11
    4.1.1 Transaction Descriptor 11
    4.1.2 Write Back Buffer 12
    4.1.3 Contention Manager 16
    4.1.4 Roll Over Clock 17
    4.1.5 Lock 17
  4.2 Challenges 17
5 Test Cases .......................... 19
5.1 Counter ........................................ 19
5.2 Array Counter ................................. 19
5.3 Dining Philosophers ......................... 19
5.4 Intset ............................................ 20
5.5 STAMP ........................................... 20

6 Results ........................................ 23
6.1 API ............................................... 23
6.2 Example ........................................ 25
6.3 Benchmarks I ................................. 26
   6.3.1 Bayes .................................... 27
   6.3.2 KMeans .................................. 27
   6.3.3 Genome .................................. 29
   6.3.4 Intruder .................................. 29
   6.3.5 Labyrinth ................................ 30
   6.3.6 SSCA2 .................................. 32
   6.3.7 Vacation ................................ 32
6.4 Benchmarks II ............................... 35

7 Conclusion ................................. 39
7.1 Future Work ................................. 39

A Assignment of tasks ................. 41

B API ............................................. 45

C Counter Code .............................. 49

D Presentation .............................. 53
1 Introduction

The goal of this thesis is to design and implement a fast and robust word-based software transactional (STM) system, called fastSTM. The fastSTM system was designed and developed in collaboration with Stefan Classen, for more details see his master thesis [CPG08]. The fastSTM system will be used together with a binary translator, which was developed by Marcel Wirth in an earlier semester thesis [WPG08]. In the mean time, the system has been extended and further improved by Peter Suter, for more insights on his work see [SPG08].

1.1 A Very Short Introduction to Software Transactional Memory

Software transactional memory is a control mechanism for manipulating shared memory in a concurrent environment. The approach is an alternative to lock-based synchronization. It allows the programmer to define pieces of code that are executed in an atomic way. The atomicity is then guaranteed by the the STM system.

Recently the first commercial processor, using hardware transactional memory, has been developed by Sun Microsystems. On a start of a transaction a snapshot of the architecture state is taken, which will be restored on a failed transaction. On a successful transaction the architectural state is then modified.

1.2 Motivation

Different software transactional memory systems such as tinySTM and Transactional Locking 2 (TL2) by [DSS06] have been presented. Unfortunately they did not fully satisfy all our needs and gave us reason to implement our own STM system. They are both briefly discussed below. For more details on tinySTM and TL2 see section 2.

tinySTM: The tinySTM system is a very small and high-performance word based STM systems. It lacks modularity and we believe that the data structures used, could be further improved to ameliorate data locality and therefore cache misses.
TL2: At the start of our project we did not have access to the *TL2* source code and therefore could not judge its implementation.

We are looking for a modular, fast and small word-base STM system. It should be specially optimized to the needs of the binary translator [WPG08]. The modularity makes prototyping of new algorithms for STM systems much easier, since only modules affected by the algorithm need to be exchanged, while leaving the remaining modules untouched.

The architecture, the STM system will be working in, is sketched in fig. 1.1 The binary translator rewrites shared memory operations, such that shared memory is now accessed through the *fastSTM* system. This allows calls to unobserved code such as shared libraries or system calls to be embedded in a transaction.

Our STM system was given the name *fastSTM* due to the believe that our design and implementation will out perform current STM systems. If the name is justified or not, can be depicted from section 6.
2 Related Work

This chapter gives the context of the semester thesis. In the first section we illustrate address based Software Transactional Memory. The following section gives a more technical overview of existing implementation of Software Transactional Memory in imperative computer languages such as C/C++. At the end we give an introduction to the bloom filter, a probabilistic data structure, we use for fast membership test in the read- and write-sets.

2.1 Address Based Software Transactional Memory

Address based software transactional memory access memory at the granularity of machine words, where as object based transactional memory access memory at object level, requiring knowledge of the underlying object structure. In this sense, address based software transactional memory is a more general approach to transactional memory. The drawback of address based transactional memory is that big memory chunks require more locks than a single object would need in an object based transactional memory system. On the other hand it allows locking on a much finer granularity as object based transactional memory.

2.2 Different Approaches To STM

This section gives a overview of two different STM systems, namely tinySTM and Transactional Locking 2 (TL2). Both state-of-the-art STM systems, designed and implemented in C/C++.

2.2.1 tinySTM

The tinySTM system was developed at the Université de Neuchâtel by Pascal Felber. The more detailed discussion of tinySTM is based on version 0.7.3, the most recent version at the time of writing. The tinySTM is a light weighted and efficient word-based STM implementation. Its time-based algorithm is derived from lazy snapshot algorithm (LSA) \cite{RFF06}. Which guarantees
that transactions always read from a consistent memory state. The tinySTM library supports
encounter-time locking, where memory writes are done by first temporarily acquiring a lock
for a given location, then writing the current shared memory value to an undo log and directly
writing the new value to shared memory. The read- and write-sets are stored in a dynamic
array, whose size is doubled once it is full. No bloom filter is used for fast membership test
in the read- or write-set. Neither does the implementation support composed transactions.

2.2.2 Transactional Locking 2

The Transactional Locking 2 (TL2) [DSS06] is a state-of-the-art word-based software trans-
actional system which is currently developed and maintained at sun microsystems. Similar
to the tinySTM system TL2 is a time-based STM system, where a global time base (i.e.,
version counter) is used to reason about the consistency of the data accessed during a trans-
actions and about the order different transactions commit. In case time did not advance in
the transactional memory system, then no transaction committed changed to shared mem-
ory. In contrast to the tinySTM system, TL2 uses a commit-time locking strategy, where
memory locations are only temporary locked at commit time. This has the disadvantage,
that read-write conflict recognized at commit-time cannot be solved without aborting at
least one of the transactions.

The implementation is similar to the fastSTM system, though some differences exist. The
read- and write-set are stored in a simple linked list. For fast membership test in the read-
and write-set a bloom filter is used. In the mean time contributers have extended the TL2
system, to support composed transactions.

2.3 Bloom Filter

The Bloom filter was first proposed in the 1970’s by Burton H. Bloom. It’s a probabilistic
data structure, which can add elements in constant time and quickly test set membership
of an element. The Bloom filter consists of a bit array of length $m$, initially all set to 0.
Furthermore $k$ hash functions are used, each function maps the key to a single bit into the
$m$-bit array. On element insertion the key is fed to $k$ hash functions, to obtain $k$ array
positions, which are all set to 1, see fig. 2.1. On element look-up the key is again fed to
$k$ hash functions, if all $k$ array positions are set to 1, the element might be member of the
set. There is still an uncertainty of an element not being part of the set, since the $k$ array
positions could have been set to 1 by different keys. If one of the $k$ array position is not set
to 1, it’s certain that the element is not contained in the set.

The fails positive probability is given in eq. 2.3.1 where $m$ denotes the total number
of bits in the array and $n$ is the number of inserted elements and $k$ the number of hash
2. Related Work

Figure 2.1: The illustration shows how two keys are inserted using two different hash functions $H1$ and $H2$.

$$P[\text{"Fails Positive"}] = (1 - (1 - \frac{1}{m})^{kn})^k \quad (2.3.1)$$

Removing elements from the filter is not foreseen. Though some approaches exist using two bloom filter. One is used to register inserted elements and the other is used to register deleted elements. More on the subject can be found in [B70].
This chapter presents the overall design and requirements of the fastSTM system. We then outline in more detail the different modules which build our fastSTM system. Finally we describe how nested transactions could be included into the fastSTM system.

3.1 Overall Design Requirements

The aim of this thesis, is to build a robust STM system which outperform current STM systems such as tinySTM or TL2 under certain constraints which are: The fastSTM system should not significantly slow down threads, which do not access shared memory. The fastSTM should scale up well on long transactions and be optimized for short transactions, with less than 1000 writes. We also make the assumption that transactions contain more reads than writes. We assume 3 to 5 times more read operations than write operations in single transaction.

To start with, we should use a 32-bit version clock, which allows us to guarantee that only consistent memory is read. On a overflow of the version clock, the STM system should be stopped and the user notified with an error message. In a later implementation cycle a overflow handler should be introduced to handle version clock overflow, by properly resetting the version counter.

A first version of the STM system should abort a transactions on contention. Later the system should correctly handle contention and make sure that at least one transaction commits successfully, to guarantee continuation.

Since our STM system should work together with a binary translator which rewrites shared memory operations on the fly, the fastSTM system should be a word-based system, which opposed to object based STM tracks changes to shared memory one word at a time, rather than on entire objects. To keep maintenance as easy as possible, it should be built in a modular way, such that different components are easily interchangeable and extendable.
3.2 fastSTM Modules

This section describes the different modules of the fastSTM system. A global overview of the system is given in fig. 3.1. It consists of five main modules, *fastSTN*, *Read Write Buffer*, *Contention Manager*, *Locks* and the *Roll Over Counter*. We elaborate on each module below.

![Figure 3.1: Module Overview of the fastSTM system.](image)

### 3.2.1 fastSTM

The *fastSTM* module represents the core of the library, by implementing the main algorithm of our fastSTM system. It should provide functionalities to start, retry and to commit a transaction. On a transaction start, the instruction pointer, pointing the the beginning of a transaction needs be stored, in case of a transaction retry the code is re-executed from the beginning of the transaction. On a transaction commit, changes to shared memory are made visible to all threads.

### 3.2.2 Read Write Buffer

The *read/write buffer* stores the read-set and write-set of a transaction. The write-set represents the changes made to shared memory, which will be made visible to all other threads on a successful commit. The read-set is needed to ensure at commit time, no shared memory has mutated since it was read.

The module should support the following functionalities:

- Acquire locks for write-set
- Release locks of write-set
- Write buffer back to shared memory
3.3 Composed Transactions

There are two simple approaches to extend our existing fastSTM system to make support for nested transactions. One possible extension would be to add a nesting counter to the transaction descriptor. Every time a new transaction is started the counter gets incremented, on every commit of a inner transaction the counter gets decremented, not actually committing the transaction. All changes made to shared memory by inner transaction are hold by read/write buffer of the outermost transaction. At the point where the outermost transaction is encountering a commit, all changes stored in the read/write buffer are written back to shared memory. The disadvange of this strategy is when the innermost transaction of a deeply nested transaction hierarchy invokes a retry, the retry needs to be executed from the outermost transaction.

An alternative approach would be, each inner transaction holds its own read/write sets. On a commit of a inner transaction, the read/write set is merged into the read/write set of the outer transaction. In case of a retry, only the transaction encountering the retry needs to be re-executed.
4 Implementation

This chapter presents the implementation of the fastSTM system. First, we present the different data structures used and the most important functions of the individual modules. At the end of the chapter we state the different challenges we encountered during development.

4.1 fastSTM

So far the fastSTM system supports three different locking strategies.

Commit-time locking: The commit-time locking strategy, stores all writes in the write back buffer, see sub-section 4.1.2. On commit all required locks are acquired and the write-set is written back to shared memory. In case acquisition of all locks fails, the write back buffer is discarded and no changes are made to shared memory.

Encounter-time locking with write back: This locking strategy was implemented by Stefan Classen. At each write operation the lock is directly acquired, the new value to write is written into the write back buffer. When reaching the commit state of the transaction all locks have been successfully acquired and the write-set can safely be written back to shared memory. In case the transaction encountered a retry, all locks are freed and the write-set is discarded.

Encounter-time locking with write through: This locking strategy is similar to the encounter-time locking described above. Instead of writing the new value to the write back buffer, changes are directly made to shared memory. A undo log is kept in the write back buffer. On a successful commit the locks are freed and the write back buffer discarded. On a failed transaction the undo log is written back to shared memory and the locks are released.

4.1.1 Transaction Descriptor

The transaction descriptor, is used to store all the thread specific data.
The *status* field represents the transaction status, which can be one of the following values: TX_IDLE, TX_ACTIVE, TX_COMMITED, TX_ABORTED or TX_WAITING. The *max_version* field holds the maximum version number, which may be read without extending the read-set. The value is set to the current version counter value, at the beginning of each transaction. The *rw_buffer* field holds the read and write set. The write-set holds all changes made to shared memory while the read-set keeps track of the last valid version read. The *env* holds the environment for setjmp operation, which is used on a transaction retry to jump back to the beginning of the transaction. The *ro* holds a flag capturing whether the transaction is read only or not. A value of 1 indicates a read-only transaction. The *allocated* field holds a pointer to the data which was allocated during a transaction. On a transaction retry the data will be de-allocated. The *freed* field holds a pointer to memory which was freed during a transaction. The memory will only be freed after a successful commit. The *waiting* field holds a pointer to the transaction that is currently holding the lock required by the current transaction to successfully commit. The *yielded* field counts the number of times the transaction has been yield while waiting for a lock to be freed.

Listing 4.1: Transaction Descriptor

```c
struct stm_tx {
    stm_word_t status;
    stm_word_t max_version;
    rw_buffer_t rw_buffer;
    jmp_buf env;
    int ro;
    mem_block_t *allocated;
    mem_block_t *freed;
    stm_tx_t *waiting_for;
    unsigned int yielded;
    ...
};
```

### 4.1.2 Write Back Buffer

The *Write Back Buffer* (WBB) is used to store all temporary changes made to shared memory during a transaction by a single thread. The description is based on the commit-time locking strategy.
4.1.2.1 Data Structure

The WBB stores the write-set and the read-set of a transaction. On each write operation the memory address, the value written and the current version number are stored in the \textit{w\_entry} struct, see 4.2. The \textit{w\_entry} also stores a pointer to the according lock, used for fast lock access. On a read operation, the current read version and a pointer to the corresponding lock are stored in the \textit{r\_entry} struct, see 4.3.

To improve data locality, the \textit{read-write-entry} 4.4 consists of a union of the a \textit{w\_entry} and a \textit{r\_entry} array, holding a maximum of 30 write-sets or 60 read-sets. The total size of the ReadWrite-Set is 480 bytes. The union struct is embedded into the \textit{read/write set} struct, which in addition holds a pointer to the next free read-entry and write-entry. Futhermore a pointer to the next read/write-set is stored together with two bloom filters of 4 byte width each. One bloom filter is used for the read-set the other for the write-set. The read/write set has a total size of 512 bytes, which corresponds to the size of multiple cache-lines.

Write entries are inserted from the top to bottom, where as read entries are inserted from bottom to top.

Listing 4.2: Write Entery.

```c
struct w_entry {
    /* Address written */
    volatile stm_word_t *addr;
    /* New value */
    stm_word_t value;
    /* Version overwritten */
    stm_word_t version;
    /* Pointer to lock (for fast access) */
    volatile stm_word_t *lock;
};
```

Listing 4.3: Read Entry.

```c
struct r_entry {
    /* Pointer to lock (for fast access) */
    volatile stm_word_t *lock;
    /* Version read */
    stm_word_t version;
};
```

Listing 4.4: Union of Read and Write struct.

```c
typedef union rw_entry {
```
4. IMPLEMENTATION

w_entry_t write_entry [W_BLOCK_ARRAY_SIZE];

r_entry_t read_entry [R_BLOCK_ARRAY_SIZE];

Listing 4.5: ReadWrite Set.

```c
struct rw_set {
    /∗ Pointer to the next rw_set used for recycling ∗/
    struct rw_set *recycled_next;
    /∗ Reference to next rw_set ∗/
    struct rw_set *next;
    /∗ Pointer to the first read entry ∗/
    w_entry_t* next_read;
    /∗ Pointer to one after the last read entry ∗/
    r_entry_t* previous_read;
    /∗ Pointer to one after the last read entry ∗/
    r_entry_t* rend;
    /∗ Pointer to the pointer in the hash table pointing to this ∗
    struct rw_set **rw_hash_entry;
    /∗ Write filter ∗/
    stm_word_t write_filter;
    /∗ Read filter ∗/
    stm_word_t read_filter;
    /∗ read−write entries!! ∗/
    rw_entry_t values;
};
```

When inserting a new write entry, the address of the write entry is hashed into the read/write-set hash table. If the hash-table already holds a pointer to a rw_set, the rw_set block is inspected for an already existing write entry. Look-up of an existing write entry is optimized by the use of a bloom filter, for more details on the bloom filter see subsection 2.3. In case the query of the bloom filter failed, indicating no write-entry was found, a new write entry is added. If a write entry already exists, the write-set needs to be searched for a possible existing entry. In case an existing entry is found the write-set value is overwritten with the new value. In case no existing entry is found a new write-set entry is added. Fig. 4.1 shows how the read/write-set, the read/write-set hash-table and the lock array are related to each other.
4. Implementation

4.1.2.2 Functions

This section provides an overview of the Write Back Buffer interface, used by the fastSTM system.

```c
void buf_init(stm_tx_t *tx);
void buf_destroy(stm_tx_t *tx);
void buf_free(stm_tx_t *tx);
```

To initialize and de-initialize the WBB, we provide the two functions, `buf_init` and `buf_destroy`. To reset all the blocks of the buffer, we provide the function `buf_free`. All three functions take as argument a transaction descriptor.

```c
int buf_validate(stm_tx_t *tx);
int buf_extend(stm_tx_t *tx);
```

On commit all locks need to be acquired, this is done using `buf_acquire_all_locks`. The function will iterate through all buffers along the recycled_next pointers, trying to obtain the lock of the different address in the write-set. In case a lock is already taken by a different transaction the contention manage is invoked. The `buf_release_all_locks` function works similarly to the `buf_acquire_all_locks` but instead of acquiring the locks the locks are released as the name of the function suggests.

```c
void buf_acquire_all_locks(stm_tx_t *tx);
```
4. Implementation

// Release all locks
void buf_release_all_locks (stm_tx_t *tx);

Read set, check if all read addresses are still valid.

// Write buffer back to shared memory
void buf_write_back (stm_tx_t *tx);

// Reset the read write buffer
void buf_reset (stm_tx_t *tx);

// Read from shared memory
stm_word_t* buf_read_shared_memory (stm_tx_t *tx, 
  volatile stm_word_t *addr, 
  volatile stm_word_t *lock, 
  ...);

// Write to shared memory
stm_word_t* buf_write_shared_memory (stm_tx_t *tx, 
  volatile stm_word_t *addr, 
  int *ret_type);

4.1.3 Contention Manager

When a transaction is about to commit, it first needs to acquire all necessary locks. During lock acquisition some locks may already be taken by a different transaction resulting in contention. In such a case the contention manager is invoked. The contention manager then analyzes the dependencies between the waiting transactions. In case the dependencies form a loop, as illustrated in fig. 4.2 a dead-lock is present, which is resolved by aborting the current transaction and initiating a transaction retry. In case no dead-lock is present the current thread initiates a yield, and retries to acquire the lock on its next schedule. In case the transaction has been waiting for too long, it will free all so far acquired locks and initiate a retry.

inline void cont_handle_conflict (stm_tx_t *tx, stm_tx_t *other);
4. Implementation

Figure 4.2: Waiting transaction form a loop and have dead-locked. The arrows indicate the waiting dependency between two transactions, i.e., transaction 2 is waiting for a lock that transaction 3 is holding.

4.1.4 Roll Over Clock

The global version clock is incremented once by each write transaction at commit time, after all locks have been successfully acquired. The global version clock is only read and never written by all read only transactions. We represent the version clock as a 4-byte word. Accessing the version clock is done through a lock-based approach. On a version clock overflow all transactions are put on hold, using a barrier. The version clock is then reset together with all transactions.

4.1.5 Lock

We use a $2^{20}$ long lock array with a word size of 4-bytes, to lock the transacted memory locations. To indicate that a lock is taken a single bit is used, the remaining 31 bits are used to store the address of the transaction, currently holding the lock. To map the shared memory onto the lock array we use a mapping function which masks the address of the shared memory with LOCK\_HASH\_ARRAY\_SIZE - 1, i.e., $FFFF$. The value obtained through masking can then be added to the base address of the lock array to derive the correct lock address. We use an atomic compare and swap (CAS) operation to acquire the lock and a store operation to release the lock. When the lock is not taken it holds the version number of the shared memory address which is incremented at each successful commit of a write transaction.

4.2 Challenges

This section discusses some challenges and pitfalls we encountered during the development of the project. We hope that this might be helpful to others developing a software transactional system.
4. Implementation

**Debugging:** To get more insights on the *fastSTM* library, we introduced thread safe printing macros. Each printed line starts with the thread id of the thread invoking the print. This helped retrieving all activities of a single thread by greping for its id. The drawback is, the more output is produced the more the threads get executed in a sequential way.

**Single core:** Since the fastSTM system was developed on a single core architecture, we encountered some surprises when running the library on a multi-core architecture. To simulate a more multi-core like execution behavior, we randomly yielded threads. Still some bugs were only found once we moved over to a true multi-core architecture. We recommend development on a true multi-core architecture which will allow finding malfunctioning of the code much earlier.

**Assertions:** We have inserted pre-condition assertions at the top of each function. This greatly helped finding the origin of some rather nifty bugs.
5 Test Cases

This section lists different test cases we used to test and benchmark our STM implementation. We first describe the tests cases we developed. Most of them are very simple test cases, which helped us test different components of the fastSTM system.

5.1 Counter

Probably the most basic example. A shared counter is incremented and decremented by each thread. Starting with a initial value of 0 the result of the counter after all increments and decrements must be 0 again, since the total number of increments and decrements are equal.

This simple example allowed us to test the basic functionalities of our fastSTM system.

5.2 Array Counter

This test case is similar to the previous counter test case. Instead of incrementing only one counter, a whole shared array is incremented and decremented at once, i.e., in an atomic way. In addition every second thread starts incrementing/decrementing the array from top to bottom and every other thread from bottom to top. This generates maximum contention on the shared array, allowing to stress test the contention manager module.

5.3 Dining Philosophers

The dining philosopher problem is a classical multi-process synchronization problem. It can be summarized in the following way: Each philosopher can either eat or think, but only one of both activities can be done at the time. All philosophers sit at a round table, each having one plate of spaghetti in front of himself. Between two philosophers there is only one fork. For a philosopher to be able to eat both the forks laying on the right and left side of the plate are needed.
The implementation uses a single transaction to fetch both forks laying on the right and left of a plate. Once a philosopher owns both forks he may eat for a 1ms, before releasing both forks again in an atomic way.

5.4 Intset

This test-case was provided by [tinySTM] and uses either red-black trees or a sorted linked-list. Typically read and write operations on the red-black tree affect only few memory locations. There for it is not a good indicator of the ability of the STM system. The sorted linked-list must be traversed in order to add, remove, or locate entries, resulting in large read-sets.

5.5 STAMP

[STAMP] is a benchmark suite, designed for transactional memory research. Currently it consists of eight benchmarks which are listed below, together with a short description. The descriptions are cited from the original readme provided together with the STAMP benchmarks. We have shortened some of the descriptions, to see the full version, please see [STAMP] and all the appropriate papers.

**Bayesian Network:** A Bayesian network is a way of representing probability distributions for a set of variables. Conceptually, a Bayesian network is represented as a directed acyclic graph, where each node represents a variable and each edge represents a conditional dependence. By recording the conditional independences among variables, a Bayesian network is able to compactly represent all of the probability distributions.

**Genome:** This benchmark implements a gene sequencing program that reconstructs the gene sequence from segments of a larger gene. For example, given the segments TCGG, GCAG, ATCG, CAGC, and GATC, the program will try to construct the shortest gene that can be made from them, which in this case would be CAGCAGATCGG.

**Intruders:** Signature-based network intrusion detection systems, scan network packets for matches against a known set of intrusion signatures. Network packets are processed in parallel and go through three phases: capture, reassembly, and detection. In the TM version included in STAMP, the capture and reassembly phases are each enclosed by transactions. When operating on these data structures, this benchmark has relatively short transactions. It also has moderate to high levels of contention depending on how often the reassembly phase rebalances its tree.
**K-Means:** K-means is a partition-based method and is arguably the most commonly used clustering technique. K-means represents a cluster by the mean value of all objects contained in it. Given the user-provided parameter $k$, the initial $k$ cluster centers are randomly selected from the database. Then, K-means assigns each object to its nearest cluster center based on the similarity function. Once the assignments are completed, new centers are found by finding the mean of all the objects in each cluster. This process is repeated until two consecutive iterations generate the same cluster assignment.

**Labyrinth:** Given a maze, this benchmark finds the shortest-distance paths between pairs of starting and ending points. The maze is represented as a grid, where each grid point can contain connections to adjacent, non-diagonal grid points. The algorithm searches for a shortest path between the start and end points of a connection by performing a breadth-first search and labeling each grid point with its distance from the start. This expansion phase will eventually reach the end point if a connection is possible. A second traceback phase then forms the connection by following any path with a decreasing distance.

**SSCA2:** The Scalable Synthetic Compact Applications 2 (SSCA2) benchmark is comprised of four kernels that operate on a large, directed, weighted multi-graph. STAMP focuses on Kernel 1, which constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. The transactional version of SSCA2 has threads adding nodes to the graph in parallel and uses transactions to protect accesses to the adjacency arrays. Since this operation is relatively small, not much time is spent in transactions. Additionally, the length of the transactions and the sizes of their read and write sets is relatively small. The amount of contention in the application is also relatively low as the large number of graph nodes lead to infrequent concurrent updates of the same adjacency list.

**Vacation:** This benchmark implements a travel reservation system powered by a non-distributed database. The workload consists of several client threads interacting with the database via the system’s transaction manager. The database consists of four tables: cars, rooms, flights, and customers. The first three have relations with fields representing a unique ID number, reserved quantity, total available quantity, and price. The table of customers tracks the reservations made by each customer and the total price of the reservations they made. The tables are implemented as Red-Black trees.

**YADA:** This benchmark implements the popular mesh generation algorithm Delaunay, which produces meshes with certain quality guarantees that are important for problems in
which the geometry of the problem changes with time. Delaunay mesh generation works by iterative refinement of a coarse initial mesh. The sequential algorithm repeatedly looks for a bad mesh element that does not satisfy the quality constraints, computes a neighborhood of that element called its cavity, and replaces the elements in that cavity with new elements, some of which may not satisfy the quality guarantees themselves.

Delaunay mesh generation can be parallelized in a natural way because elements that are far away in the mesh do not interfere with each other as they are being processed.
6 Results

In this chapter, we evaluate our STM system. We describe the API of the system and provide a minimalistic sample code, which illustrates the usage of the STM system. At the end of the chapter we provide benchmarks between the fastSTM, tinySTM and the TL2 system.

6.1 API

This section elucidates the most important functions of the *fastSTM* library. The full extract of the *fastSTM* header file can be found in appendix B. A working sample code using our STM library is reported in the next section 6.2.

```c
void stm_init(void);
void stm_exit(void);

void *stm_new(void);
void stm_delete(stm_tx_t *tx);
```

The two above functions are used to initialize and un-initialize the whole STM system. They are normally called once per process, before and after using the *fastSTM* system.

```c
stm_tx_t *stm_new(void);
void stm_delete(stm_tx_t *tx);
```

The *stm_new* function creates a new transaction descriptor, usually called once per thread. The function then returns a transaction descriptor which may be stored as a thread specific pointer. The *stm_delete* function, takes a transaction descriptor and deallocates all memory associated with the transaction descriptor. This function is called once the *fastSTM* system is no longer used.

```c
void stm_start(stm_tx_t *tx, jmp_buf *env);
void stm_commit(stm_tx_t *tx);
void stm_retry(stm_tx_t *tx);
void stm_abort(stm_tx_t *tx);
```
To start a transaction the \textit{stm\_start} function is used. When calling the function two arguments are required the transaction descriptor and the environment i.e., the instruction address of the transaction start. On a retry the re-execution of the transaction will start there. The \textit{stm\_commit} function will try to commit the current transaction. In case of a failure a transaction retry is invoked. The function takes as only argument a transaction descriptor. The \textit{stm\_retry} function forces the current running transaction to stop the execution and restart execution from the beginning of the transaction. The function takes as argument a transaction descriptor. The \textit{stm\_abort} function will discard all changes made to shared memory so far and continue execution after the call of \textit{stm\_abort}. The function takes as argument a transaction descriptor.

\begin{verbatim}
stm\_word\_t \textbf{stm\_load}(stm\_tx\_t *tx, volatile \textbf{stm\_word\_t} *addr); void \textbf{stm\_store}(stm\_tx\_t *tx, volatile \textbf{stm\_word\_t} *addr, \textbf{stm\_word\_t} value); void \textbf{stm\_store2}(stm\_tx\_t *tx, volatile \textbf{stm\_word\_t} *addr, \textbf{stm\_word\_t} value, \textbf{stm\_word\_t} mask);
\end{verbatim}

To load data from shared memory the \textit{stm\_load} is used. To call the function two arguments are required: a transaction descriptor and the memory address from which we would like to read. To write data back to shared memory the function \textit{stm\_store} and \textit{stm\_store2} are used. Both function require as argument a transaction descriptor, the address of the shared memory location to write to and the value to be written. The \textit{stm\_store2} function in addition takes a bit-mask which allows to write back one single byte.

\begin{verbatim}
void *\textbf{stm\_malloc}(stm\_tx\_t *tx, \textbf{size\_t} size); void \textbf{stm\_free}(stm\_tx\_t *tx, void *addr); void *\textbf{stm\_realloc}(stm\_tx\_t *tx, void *addr, \textbf{size\_t} size);
\end{verbatim}

To allocate or free memory during a transaction the special function \textit{stm\_malloc}, \textit{stm\_realloc} and \textit{stm\_free} need to be used, replacing the standard library function \textit{malloc}, \textit{realloc} and \textit{free}. The \textit{stm\_malloc} function takes as argument a transaction descriptor together with the size in bytes of the memory to be allocated. The function will then return a void pointer to the newly allocated memory. When calling the \textit{stm\_free} function, a transaction descriptor needs to passed as argument together with the memory address to be freed. Similar to the previous two functions, the \textit{stm\_realloc} functions requires a transaction descriptor and the address of the memory location which needs to be re-allocated together with the new size in bytes.
The function \texttt{stm\_get\_tx} retrieves the transaction descriptor of the current thread. The function \texttt{stm\_get\_env} takes as argument the current transaction descriptor and returns the jump buffer of the current thread. The jump buffer holds an instruction pointer to the beginning of the transaction.

6.2 Example

This section illustrates the usage of our fastSTM system. We present a simple counter which will be incremented by multiple threads. The full source code of the small example can be found in appendix C.

To start with, let's define some macros which will help keeping the code simple.

\begin{verbatim}
#define NB_THREADS 10
#define NB_INCS 100000

#define START
  jmp\_buf *\_e = stm\_get\_env(tx);
  sigsetjmp(*\_e, 0);
  stm\_start(tx, _e)
#define LOAD(addr) stm\_load(tx, (stm\_word\_t *)addr)
#define STORE(addr, value) stm\_store(tx, (stm\_word\_t *)addr, (stm\_word\_t)value)
#define COMMIT
  stm\_commit(tx);
\end{verbatim}

The macro \texttt{START} is used to start a new transaction, it will set the jump buffer to the beginning of the transaction, such that on a retry, execution will start again right at the beginning of the transaction. The \texttt{LOAD} macro will return the value stored at the memory address \texttt{addr}. To write a value back to memory the macro \texttt{STORE} is used together with the argument \texttt{addr} and \texttt{value}. A transaction is then committed using the \texttt{COMMIT} macro.

The shared counter is defined as a static volatile variable.

\begin{verbatim}
static volatile int counter;
\end{verbatim}

The \texttt{inc} function reads the shared counter, increments the value by 1 and writes it back to shared memory. Notice the code between the \texttt{START} and \texttt{COMMIT} statement will be executed in an atomic way.
6. Results

```c
void inc(stm_tx_t* tx)
{
    int val;
    START;
    val = (int)LOAD(&counter);
    val = val + 1;
    STORE(&counter, val);
    COMMIT;
}
```

Each thread will then have to create its own transaction descriptor using `stm_new`. Once the fastSTM system is no longer used it can be deallocated using `stm_delete`.

```c
void* startTest(void* arg)
{
    stm_tx_t* tx;
    int i;

    thread_data_t* data = (thread_data_t*) arg;

    tx = stm_new();

    /* Inc counter */
    for (i = 0; i < NB_INCS; i++){
        inc(tx);
    }

    stm_delete(tx);
    return NULL;
}
```

6.3 Benchmarks I

To evaluate the performance of our fastSTM implementation, we have used the [STAMP] benchmark suite. We measured and compared three different metrics, namely: execution time, retries and commit ratio, L2 cache miss ratio and minor page faults. All benchmarks were run on a Intel Core Duo 3.00GHz processor with 2GB ram, using the compile flags `-O3` and `-DNDEBUG`. Furthermore each benchmarks was run using 1, 2, 4, 8 and 16 threads.
6. Results

Timing: To benchmark the execution time, we ran each benchmark 20 times. We compare 5 different STM implementations, i.e., tinySTM with write through and write back, fastSTM with write through and write back and finally the TL2 system. The execution time graphs show the mean and standard deviation of all 20 runs.

Transaction: This benchmark captures the ratio of retries and commits. The average of retries and commits was obtained by removing the two best and worst samples from 10 runs. We do not provide any results for the TL2 STM system.

Page Faults: This benchmark provides the total number of minor page faults of the fastSTM-wt and tinySTM-wt. Minor page fault occur when accessing a page which is present in memory, but its status is not set to 'present' in hardware. These faults do not involve disk latency and therefore are less expensive than major page faults. Major page fault occurs when a page is accessed, which is not present in memory and need to be loaded from disk. These faults are more expensive, since they add disk latency to the interrupted program execution. During benchmarking we did not encounter any major page faults.

L2 Cache Misses: The plots provide the ratio between the mean number of L2 cache misses and L2 cache access of a total of 5 runs. We only provide results for the fastSTM-wt and the tinySTM-wt system.

6.3.1 Bayes

This benchmark provides very few transactions i.e., less than 800, see also table 6.1. All STMs have similar execution times as shown in the graph 6.1. The ratio of retries and commits is very low and therefore not very informative. The cache misses are similar in both designs and the page faults in the same order of magnitude.

<table>
<thead>
<tr>
<th>Threads</th>
<th>fstm-wb</th>
<th>fstm-wt</th>
<th>tstm-wb</th>
<th>tstm-wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>696</td>
<td>0</td>
<td>696</td>
</tr>
<tr>
<td>2</td>
<td>5.8</td>
<td>684</td>
<td>5.4</td>
<td>682.6</td>
</tr>
<tr>
<td>4</td>
<td>6.8</td>
<td>665</td>
<td>7.2</td>
<td>665</td>
</tr>
<tr>
<td>8</td>
<td>4.8</td>
<td>637</td>
<td>5.2</td>
<td>648</td>
</tr>
<tr>
<td>16</td>
<td>8.2</td>
<td>702.2</td>
<td>7</td>
<td>705.2</td>
</tr>
</tbody>
</table>

Table 6.1: Bayes: The table list the number of retries and commits.

6.3.2 KMeans

Fig. 6.2 shows a poor execution time of our design. When run using 16 threads the fstm-wt can compete with TL2 and tstm-wb, but not with the tstm-wt. The high retry rate of up
6. Results

Figure 6.1: Bayes: top left: execution time, top right: transactions, bottom left: cache misses and bottom right: minor page faults.

to 300% does not significantly affect the execution time of the tinySTM system. While our system provides a retry rate which does not exceed the 50% boundary, tinySTM provides a 2% lower cache miss rate than fastSTM. The page fault rate of fastSTM increases linearly with the number of threads.

<table>
<thead>
<tr>
<th>Threads</th>
<th>fstm-wb</th>
<th>fstm-wt</th>
<th>tstm-wb</th>
<th>tstm-wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1.3109e5</td>
<td>0</td>
<td>1.3109e5</td>
</tr>
<tr>
<td>2</td>
<td>9467.6</td>
<td>1.3114e5</td>
<td>9319</td>
<td>1.3114e5</td>
</tr>
<tr>
<td>4</td>
<td>10594</td>
<td>1.3123e5</td>
<td>76921</td>
<td>1.3123e5</td>
</tr>
<tr>
<td>8</td>
<td>23715</td>
<td>1.3142e5</td>
<td>1.4542e5</td>
<td>1.3142e5</td>
</tr>
<tr>
<td>16</td>
<td>26519</td>
<td>1.3181e5</td>
<td>1.9944e5</td>
<td>2.9077e5</td>
</tr>
</tbody>
</table>

Table 6.2: K-Means: The table list the number of retries and commits.
6.3.3 Genome

In this benchmark all STMs scale similarly with the number of threads. The execution time graph shows how our system suffers from a constant time overhead, which could be related to the many malloc calls required to build-up our buffer data structure. On the other hand our implementation has a much lower retry rate compared to tinySTM. The L2 cache miss rate of our design is roughly 30% lower compared to the tinySTM and the number of minor page faults increases linearly with the number of threads, while the tinySTM shows a nearly constant page fault rate.

6.3.4 Intruder

This benchmark also nicely shows the constant time overhead of our implementation. It scales well with the number of threads and clearly outperforms the tinySTM system when run using 16 threads. The ratio of retries nearly grows exponentially with the number of
6. Results

Figure 6.3: Genome: top left: execution time, top right: transactions, bottom left: cache misses and bottom right: minor page faults.

<table>
<thead>
<tr>
<th>Threads</th>
<th>fstm-wb</th>
<th>fstm-wt</th>
<th>tstm-wb</th>
<th>tstm-wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1.9593e6</td>
<td>0</td>
<td>1.9593e6</td>
</tr>
<tr>
<td>2</td>
<td>4255</td>
<td>1.9593e6</td>
<td>4269.4</td>
<td>1.9593e6</td>
</tr>
<tr>
<td>4</td>
<td>32736</td>
<td>1.9593e6</td>
<td>29052</td>
<td>1.9593e6</td>
</tr>
<tr>
<td>8</td>
<td>71382</td>
<td>1.9593e6</td>
<td>69476</td>
<td>1.9593e6</td>
</tr>
<tr>
<td>16</td>
<td>1.7399e5</td>
<td>1.9593e6</td>
<td>2.0035e5</td>
<td>1.9593e6</td>
</tr>
</tbody>
</table>

Table 6.3: Genome: The table list the number of retries and commits.

threads in case of the *tinySTM* design and stays constant in our implementation. Also, our design shows a 12% lower cache misses rate.

The benchmark provides moderate to high contention, with large read- and write-sets.

### 6.3.5 Labyrinth

In this benchmark all STMs perform about the same. This is due to the computation intensive algorithm, which uses only a few transactions, see 6.5. The cache-miss rate and
6. Results

Figure 6.4: Intruder: top left: execution time, top right: transactions, bottom left: cache misses and bottom right: minor page faults.

<table>
<thead>
<tr>
<th>Threads</th>
<th>fstm-wb</th>
<th>fstm-wt</th>
<th>tstm-wb</th>
<th>tstm-wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>9.4978e6</td>
<td>0</td>
<td>9.4978e6</td>
</tr>
<tr>
<td>2</td>
<td>1.0106e6</td>
<td>9.4978e6</td>
<td>1.029e6</td>
<td>9.4978e6</td>
</tr>
<tr>
<td>4</td>
<td>1.3168e6</td>
<td>9.4978e6</td>
<td>1.2897e6</td>
<td>9.4978e6</td>
</tr>
<tr>
<td>8</td>
<td>1.7221e6</td>
<td>9.4978e6</td>
<td>1.6747e6</td>
<td>9.4978e6</td>
</tr>
<tr>
<td>16</td>
<td>2.7265e6</td>
<td>9.4978e6</td>
<td>2.6978e6</td>
<td>9.4978e6</td>
</tr>
</tbody>
</table>

Table 6.4: Intruder: The table list the number of retries and commits.

number of page faults is very low and nearly identical in both designs.

<table>
<thead>
<tr>
<th>Threads</th>
<th>fstm-wb</th>
<th>fstm-wt</th>
<th>tstm-wb</th>
<th>tstm-wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>514</td>
<td>0</td>
<td>514</td>
</tr>
<tr>
<td>2</td>
<td>13.8</td>
<td>516</td>
<td>13.8</td>
<td>516</td>
</tr>
<tr>
<td>4</td>
<td>33.4</td>
<td>520</td>
<td>37.8</td>
<td>520</td>
</tr>
<tr>
<td>8</td>
<td>76.2</td>
<td>528</td>
<td>74.6</td>
<td>528</td>
</tr>
<tr>
<td>16</td>
<td>135</td>
<td>544</td>
<td>129.8</td>
<td>544</td>
</tr>
</tbody>
</table>

Table 6.5: Labyrinth: The table list the number of retries and commits.
6. Results

Figure 6.5: Labyrinth: top left: execution time, top right: transactions, bottom left: cache misses and bottom right: minor page faults.

### 6.3.6 SSCA2

All four STMs have a similar execution time and scale exceptionally well. The ratio of retries and commits is extremely low on our implementation compared to the tinySTM even though there are more than 2\(K\) commits. In both implementation the rate of cache misses is very high, with over 90%. The number of minor page faults are very high too, but in the same order of magnitude.

Notice the absence of the constant time overhead in our design. This is due to the short transactions, which provide small read- and write-sets.

### 6.3.7 Vacation

Fig. 6.7 shows a constant time overhead of our system. This is due to the long transactions with large read- and write-sets. With increasing number of threads our system scales slightly better compared to the others. The tinySTM has a surprisingly high retry rate with up to
6. Results

figure 6.6: SSCA2: top left: execution time, top right: transactions, bottom left: cache misses and bottom right: minor page faults.

<table>
<thead>
<tr>
<th>Threads</th>
<th>fstm-wb</th>
<th>fstm-wt</th>
<th>tstm-wb</th>
<th>tstm-wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2.2981e6</td>
<td>0</td>
<td>2.2981e6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2.2983e6</td>
<td>3.2</td>
<td>2.2983e6</td>
</tr>
<tr>
<td>4</td>
<td>1430.8</td>
<td>2.2985e6</td>
<td>5643.6</td>
<td>2.2985e6</td>
</tr>
<tr>
<td>8</td>
<td>6577.8</td>
<td>2.2988e6</td>
<td>14452</td>
<td>2.2988e6</td>
</tr>
<tr>
<td>16</td>
<td>15112</td>
<td>2.2995e6</td>
<td>31305</td>
<td>2.2995e6</td>
</tr>
</tbody>
</table>

Table 6.6: SSCA2: The table list the number of retries and commits.

Eight times more retries than commits. While our systems retry rate stays below 1.5. The graph also shows a 20% lower $L2$ cache miss rate while minor page faults dramatically increase with the number of threads.
Figure 6.7: Vacation: top left: execution time, top right: transactions, bottom left: cache misses and bottom right: minor page faults.

<table>
<thead>
<tr>
<th>Threads</th>
<th>fstm-wb</th>
<th>fstm-wt</th>
<th>tstm-wb</th>
<th>tstm-wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 7e5</td>
<td>0 7e5</td>
<td>0 7e5</td>
<td>0 3485.4 7e5</td>
</tr>
<tr>
<td>2</td>
<td>2579 7e5</td>
<td>3.0424e5 7e5</td>
<td>9.9643e5 7e5</td>
<td>3549 7e5</td>
</tr>
<tr>
<td>4</td>
<td>3.3179e5 7e5</td>
<td>5.6708e5 7e5</td>
<td>9.8409e5 7e5</td>
<td>3485.4 7e5</td>
</tr>
<tr>
<td>8</td>
<td>5.4527e5 7e5</td>
<td>9.9932e5 7e5</td>
<td>2.2516e6 7e5</td>
<td>2.4279e6 7e5</td>
</tr>
<tr>
<td>16</td>
<td>9.9932e5 7e5</td>
<td>1.0079e6 7e5</td>
<td>5.3136e6 7e5</td>
<td>4.63e6 7e5</td>
</tr>
</tbody>
</table>

Table 6.7: Vacation: The table list the number of retries and commits.
6.4 Benchmarks II

Here we present execution time measurements on a 8 core machine with 8GB of ram and each core running at a clock rate of 3.00GHz. The benchmarks were executed 10 times, we then removed the 2 best and worst measurements. The discussion on each benchmark is directly enclosed in the caption of the graph. The graphs show the mean execution time and its standard deviation.

Figure 6.8: Bayes: The execution time is similar to the one shown in the previous section. Best performance is provided by the fstm-wb.
6. Results

Figure 6.9: Genome: Similar to the measurements of the previous section we see that our implementation suffers from a constant time overhead. All three STMs scale equally well.

Figure 6.10: Intruder: When run using more than 8 threads our implementation encounters a premature termination, due to too much contention on the locks. In case our design encounters more than 6 failed compare and swap operations, we terminate execution with an appropriate failure message. Notice that the $TL2$ design scales much better on 16 threads than $tstm$ does.
Figure 6.11: KMeans: In this benchmark the *fstm* provides the worst scale, compared to *TL2* and *tstm*. Notice how the execution time of *TL2* increases by less than 0.05 seconds between 4 and 16 threads, while *fstm-wt* nearly doubles its execution time.

Figure 6.12: Labyrinth: Here all STMs perform about the same, this is due to the small transactions size and the low contention rate of the benchmark.
Figure 6.13: SSCA2: Similar to the labyrinth benchmark, this benchmark provides a small transaction size and low contention, resulting in a nearly identical execution time for all five STMs.

Figure 6.14: Vacation: The benchmark nicely shows the constant time overhead of our design. All five STMs scale similar except of TL2 on 16 threads.
7 Conclusion

We have build from ground up a modular and extendable fastSTM library that allows word-based software transactions. Though the library was designed and developed to work together with a binary translator [WPG08], it may also be used as a standalone library in C/C++ programs.

The modularity of the fastSTM system, allows researchers to prototype and experiment with new algorithms without the need to design a complete STM system from scratch.

To make benchmarking of the fastSTM system with current STM system more easy we have used a similar library interface as the [tinySTM] uses. This also allowed us to use the same test-cases provided with the tinySTM.

7.1 Future Work

Here we provide some possible extension to our fastSTM system. The list is by no means complete, but instead provides a point of start for future improvements to the library.

Z-Linearization: Our fastSTM system does not take special care of long transactions, which could result in a long transaction not being able to commit, because constantly smaller transactions are committing to shared memory making the read-set of the long transaction invalid and forcing the long transaction to abort. Currently Felber et al. have investigated the use of weaker semantics for transactional memory and introduced a new concept called z-linearization, which provides a good practical performance even for long transactions, as they show in there paper [RFSF07].

Composable Transaction: So far the fastSTM system does not support nested transactions and therefore limits itself to library calls which do not make use of transactions. A possible approach on how composed transactions could be included in the future is described
Memory Consumption: The buffer structure we propose could encounter not optimal memory usage, leaving many read/write buffers only half full. It would be interesting to evaluate the memory usage of the fastSTM system and the actual memory usage of the system. Furthermore to compare memory usage between our implementation and tinySTM.
A Assignment of tasks
LibSTM - a userspace library for software transactional memory

Semester Thesis for Olivier Saurer

September 2007

Introduction

Software Transactional Memory (STM) is a concurrency control mechanism very similar to database transactions. Threads can start transactions and all following reads and writes are filtered through a buffer and tracked. Later the thread can commit the transaction and all the reads and writes are evaluated for concurrent access and conflicts. If any conflict arises then the transaction can be rolled back. 

A lot of new ideas around STM exists. It can be used to implement a lock-free control of shared resources, speculative execution and much more.

Tasks

Your task is to implement a software transaction system that can be used from a user program. The userspace program should be able to start a transaction. After this (kernel) call all following writes and reads should be hidden in a transaction. After a second (kernel) call the transaction can be evaluated and if there are no conflicts committed back to main memory.

Extending a stm-library to implement transactional memory (rewriting the code so that all writes are redirected into a buffer and are checked on commit-time). This can be achieved by using a buffer for data dependencies and writes and startTransaction, checkTransaction and commitTransaction as library functions.

- Get used to binary translators (PIN) and STM (read some papers in these fields and check the software products)
- Work together with Stephan Classen on the translation system for Transactional Memory
- Help implementing an optimized memory buffer for transactional words
- Help implementing the following functions: startTransaction, endTransaction, checkTransaction and abortTransaction.
- Test your part of the system by developing some MicroBenchmarks
- Write your thesis
- Prepare for the presentation in front of the group
General instructions

• Prepare an overview on the subject and a time plan after two weeks.

• The project should be completed by February 2008.

• The course of work should be discussed regularly with the assistant.

• The deliverables of this work are (1) the source code, (2) a report, and (3) a one page summary.

• Where applicable, the source code should be provided in the form of a patch relative to the modified initial code (e.g., in the case of Linux kernel modifications).

• The report should be phrased as a scientific essay and should be submitted as two paper copies, and in PDF format. The preferred language is English, but at the preference of the student it can be written in German.

• The one page summary should be written in English and provided as an XML file. A template is provided at http://www.lst.inf.ethz.ch/teaching/sada_template.xml.

• There should be an oral presentation at the end of the project.

Professor:  Prof. T. Gross
Assistant:  Mathias Payer
B API

/************************** Functions **************************/

/** Inits the whole STM system */
void stm_init(void);
/** Frees the datastructure of the STM */
void stm_exit(void);

/** Creates a new transaction descriptor */
stm_tx_t *stm_new(void);
/** Deletes the transaction descriptor and */
  * frees the memory it occupies */
void stm_delete(stm_tx_t *tx);

/** Starts a transaction */
void stm_start(stm_tx_t *tx,
              jmp_buf *env);
/** Commits a transaction */
void stm_commit(stm_tx_t *tx);
/** Retries the transaction */
void stm_retry(stm_tx_t *tx);
/**
 * Aborts a transaction -> this will discarded any
 * changes to the shared memory and
 * continues executing after the function call.
*/
B. API

```c
/*
void stm_abort(stm_tx_t *tx);
*/

/** Reads a shared address and returns its value */
stm_word_t stm_load(stm_tx_t *tx,
                   volatile stm_word_t *addr);

/** Stores a value to a shared address */
void stm_store(stm_tx_t *tx,
               volatile stm_word_t *addr,
               stm_word_t value);

void stm_store2(stm_tx_t *tx,
                volatile stm_word_t *addr,
                stm_word_t value,
                stm_word_t mask);

/** Allocates memory */
void *stm_malloc(stm_tx_t *tx,
                size_t size);

/** Frees memory */
void stm_free(stm_tx_t *tx,
             void *addr);

/** Reallocates memory */
void *stm_realloc(stm_tx_t *tx,
                 void *addr,
                 size_t size);

/** Get a pointer to the transactional version of a
 * shared address for reading */
volatile void* stm_get_read_addr(stm_tx_t *tx,
                                  volatile void *addr,
                                  unsigned int num_bytes);

/** Get a pointer to the transactional version of a shared
 * address for reading */
volatile void* stm_get_read2_addr(stm_tx_t *tx, 
    volatile void *addr,
    unsigned int num_bytes);
/** Get a pointer to the transactional version of a shared 
 * address for writing */

volatile void* stm_get_write_addr(stm_tx_t *tx, 
    volatile void *addr,
    unsigned int num_bytes);
/** Get a pointer to the transactional version of a shared 
 * address for modifying */

volatile void* stm_get_modify_addr(stm_tx_t *tx, 
    volatile void *addr,
    unsigned int num_bytes);
/** Tells the STM that the writing/modifying is done */

void stm_finish_writing (stm_tx_t *tx, 
    volatile void *addr,
    unsigned int num_bytes);

/** Gets the transaction descriptor of the current thread */
stm_tx_t *stm_get_tx();
/** Get the jump buffer of the current thread */
jmp_buf *stm_get_env(stm_tx_t *tx);

/** Returns true if the current thread is in a transaction */
int stm_in_transaction(stm_tx_t *tx);

/** Get statistic from the current thread */
int stm_get_parameter(stm_tx_t *tx, const c
C Counter Code

#include <pthread.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <atomic_ops.h>

#include "fastSTM.h"

/* Number of threads */
#define NB_THREADS 10
#define NB_INCS 100000
#define START {
jmp_buf *e = stm_get_env(tx);

#define START_RO START
#define LOAD(addr) stm_load(tx, (stm_word_t *)addr)
#define STORE(addr, value) stm_store(tx, (stm_word_t *)addr, (stm_word_t)value)
#define COMMIT stm_commit(tx);

/* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #*/

static volatile int counter;
/ * Counter operations
 * # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #*/

void inc (stm_tx_t* tx)
{
    int val;
    START;
    val = (int)LOAD(&counter);
    val = val + 1;
    STORE(&counter, val);
    COMMIT;
}

void* startTest (void* arg)
{
    stm_tx_t* tx;
    int i;

    thread_data_t* data = (thread_data_t*) arg;

    tx = stm_new ();

    /* Inc counter */
    for (i = 0; i < NB_INCS; i++){
        inc(tx);
    }

    stm_delete (tx);
    return NULL;
}

/* # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # */
int main(int argc, char **argv)
{
    pthread_t tid[NB_THREADS];
    int idx;

    stm_init();

    thread_data_t data[NB_THREADS];

    /* Create threads */
    for (idx = 0; idx < NB_THREADS; idx++)
    {
        if (pthread_create(&tid[idx], NULL, startTest,
                            (void*)(&data[idx])) != 0)
            fprintf(stderr, "Error creating threads\n");
        fprintf(stdout, "START thread: %p\n", (void*)tid[idx]);
    }

    /* Sync threads */
    for (idx = 0; idx < NB_THREADS; idx++)
    {
        pthread_join(tid[idx], NULL);
    }

    stm_exit();

    fprintf(stdout, "− − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − −\n");
    fprintf(stdout, "Number of threads: \t\t%0i\n", NB_THREADS);
    fprintf(stdout, "Number of inc: \t\t%0i\n", NB_INCS);
    fprintf(stdout, "Final counter value: \t\t%0f\n", counter);

    return 0;
}
D  Presentation
fastSTM - a userspace library for software transactional memory

Olivier Saurer
<saurero@student.ethz.ch>

Overview

• Motivation
• Introduction
• fastSTM Modules
• Locking Strategies
• Benchmarks
Motivation

• LibSTM overview:
  
  Word-Based STM
  • Accesses memory at word level
  • Locking at a fine granularity
  • More general than object based STM

Object Based STM
• Memory accesses at object level
• Requires knowledge of object structure

Introduction

• TL2 by Dice, Shalev and Shavit
  – Commit-time locking
  – Simple linked list stores read- and write-set
  – Uses bloom filter for fast membership test of read- and write-set
  – Non modular architecture

• TinySTM by Felber, Fetzer and Riegel
  – Encounter-time locking
  – Dynamic array stores read- and write-set
  – Non modular architecture

• fastSTM
  – Commit- and encounter-time locking
  – Block buffer stores read- and write-set together, to improve data locality
  – Uses bloom filter for fast membership test of read- and write-set
  – Modular architecture
fastSTM - Overview

- Consists of five modules
  - fastSTM
  - Read / Write Buffer
  - Contention Manager
  - Roll Over Counter
  - Locks

Module - fastSTM

- Example of a read/write transaction
  - Load global version-counter
  - Speculative execution
  - Lock write-set
  - Increment global version clock
  - Validate read-set
  - Commit and release the locks

```
START;
val = LOAD(&counter);
val = val + 1;
STORE(&counter, val);
COMMIT;
```
Module – Read / Write Buffer

Module - Read / Write Set

- Total size of read/write set 512 bytes
- Bloom filter for fast membership test
- Pointers for fast read- and write-set access
- Pointers for inter struct linkage
- Holds 30 write- or 60 read-entries
Module - Contention Manager

- Analyze dependencies of waiting transactions
  - On dead-locks (cyclic dependencies)
    - Abort current transaction
  - Otherwise
    - Yield thread
    - If waiting for too long, abort transaction

Module - Roll Over Counter & Locks

- Roll Over Counter
  - 32-bit global version counter
  - Counter access
    - At transaction start: Atomic load
    - At commit: Atomic fetch and increment
  - On overflow halt all transaction and reset counter

- Locks
  - Array of $2^{20}$ lock entries
  - Not taken locks hold the version number
  - Taken locks hold a pointer to the transaction holding the lock
  - Lock access through atomic compare and swap operation
Locking Strategies

• Commit-time locking
  – Acquires locks at commit time
  – Reads and writes are stored in read / write buffer
  – Detects conflicts late
  – Cheap aborts, expensive commits

• Encounter-time locking
  – Acquires locks when memory is written
  – Detects conflicts early
  – Write-back
    • Reads and writes are stored in read / write buffer
    • Cheap aborts, expensive commits
  – Write-through
    • Writes directly to shared memory
    • Read/Write buffer holds undo log, which is replayed on abort
    • Expensive aborts, cheap commits

Benchmarks - Intruder

• Network intrusion detection system
  – Match network packets against intrusion signature.
  – Three stages: capture, reassemble, detection

• TM version
  – Capture and reassemble are both enclosed in transactions
  – Moderate transaction length
  – Large read- and write- sets
  – High level of contentions
  – One run produces over 9M transactions
Benchmarks - Intruder

Execution Time

Retry rate

Benchmarks - Intruder

L2 cache miss rate
Benchmarks– SSCA2

• Scalable Synthetic Compact Application 2
  – Constructs an efficient graph data structure using adjacency arrays.

• TM version
  – Adding nodes to the graph and accessing the adjacency array are both enclosed in transactions
  – Short transactions
  – Small read- and write-sets
  – One run produces over 3M transactions

Benchmarks– SSCA2

Execution Time

Retry rate
Benchmarks – SSA2

L2 cache miss rate

Conclusion & Future Work

• **Conclusion**
  – Our implementation suffers from a constant time overhead
  – It though provides low contention, less cache misses
  – Modularity simplifies prototyping of new algorithms for STM

• **Future Work**
  – Add composable transactions
  – Give longer transactions a better chance to survive
  – Additional benchmarks:
    • Fill rate of read/write buffer
    • Number of reads and writes per second
    • Number of yields per commit
Questions?

Benchmarks - Vacation

• Travel reservation system
  – Uses Red-Black trees as database tables.
  – Four tables are related: cars, rooms, flight and customers

• TM version
  – Table access is enclosed in a transaction
  – Moderate transaction length
  – Large read- and write- sets
  – High contention
  – One run produces over 700K transactions
Benchmarks - Vacation

Execution Time

Retry / Commit ratio
Bibliography


