Scientific Report for STSM project

Vesna Smiljković
vesna.smiljkovic@bsc.es

January, 2013

Reference Code	COST-STSM-ECOST-STSM-IC1001-100912-021990
STSM title	Compiler-guided techniques for reducing overhead in TM-based fault-tolerant systems
STSM Period	2012-09-10 to 2012-12-07
Participant	Vesna Smiljković Barcelona Supercomputing Center, Spain
Host	Prof. Christof Fetzer Technische Universität Dresden, Germany

1 Purpose of the STSM

Transactional Memory (TM) [1, 2] has been proposed as a concurrency control mechanism to simplify multithreaded programing. TM ensures that a group of load and store operations is executed atomically and in isolation, and that programmers do not have to take care of locking and unlocking data accessed in these operations. Therefore, TM provides robustness and scalability of applications. Many researchers have been interested in TM in last decades. Numerous implementations in software (e.g. as a library [3] or a compiler extension [4]) and in hardware (as parts of mainstream processors [5, 6]) motivate researchers to investigate even further - to find other possible purposes of TM.

FaulTM [7] is a fault-tolerance technique that exploits hardware TM characteristics to ensure fault detection and recovery. A programmer defines a vulnerable section of an application and FaulTM executes it in two parallel threads, as transactions. At the end of the transactions, any mismatch in write sets indicates faulty execution. In case of a mismatch, transactions have to be re-executed. In order to provide full error coverage, it is essential to compare the register file as well, since a soft error in registers can be propagated to other components. The register file comparison introduces additional overhead.

The purpose of this short term scientific mission (STSM) was to implement compiler-guided techniques to decrease TM and reliability costs while remaining the full error coverage. To reduce the comparison overhead, we proposed analyzing variables and registers liveness and transforming code to decrease the lifetime of a variable, i.e. to decrease the vulnerability.

According to the latest FaulTM paper [7], the comparison of the register file is performed in the commit phase of transactions and produces overhead of 13% on average for SPEC2006 benchmarks. To reduce this, authors propose an optimization - comparing hash-based signatures of write sets and register files.

After an initial research on this topic, we conclude that we would not be able to reasonably reduce this overhead further. First, code transformations could affect only the general purpose registers, and their number is relatively small. Second, we assume that code transformations might introduce additional overhead (e.g. due to register spilling), which was not considered in the proposal. Therefore, we decided to change the plan of this STSM and to investigate another interesting topic - deterministic execution of TM applications.

In general, multithreaded programs are nondeterministic. Threads interleave in a nondeterministic order and might produce a different output in each execution started with the same input. Sources of nondeterminism are other processes running in parallel, the state of memory, caches and the state of any microarchitecture structure [8].

Non-deterministic programs are hard to test, debug, and tolerant to faults. On the other hand, determinism provides repeatability - for the same input each execution gives the same output, and repeatability provides easier testing, debugging and fault tolerance. A programmer can estimate the output of correct execution and compare it with the output got from the program's execution. If there is a mismatch, the program contains a bug, and the programmer re-

executes the program expecting that the bug manifests again, which is important for debugging. Importantly for fault-tolerant system, any mismatch in replicas' outputs should be only due to faulty execution one of the replicas, rather than due to non-deterministic threads interleaving.

Previous implementations of systems for deterministic multithreaded execution [9, 8, 10, 11] focus on lock-based applications and provide the deterministic order of lock acquisitions (weak determinism) or memory accesses (strong determinism). Our goal is to provide deterministic execution of TM applications. We take advantage of functionalities of TM tools and libraries: a TM compiler instruments memory accesses and distinguishes shared data accesses from thread-local data accesses, and a TM library provides the synchronization of the shared data accesses.

We rely on an existing TM library and provide deterministic execution of TM benchmarks. Our modifications remain within the TM library and do not require modifications of neither system library nor benchmarks. We suggest several different implementations, evaluate and compare performance with non-deterministic execution. In addition, our modifications can improve performance of benchmarks with high contention.

2 Description of the work carried out during the STSM

We provide a description of four different implementations of a deterministic TM system: *Serial-run*, *Write-with-token*, *Abort-others* and *Write-X-times*.

2.1 Serial-run

The first implementation is a modified global-lock TM implementation. Originally, TM serializes execution of all threads in the way that threads acquire the global lock when they start their atomic sections. The first thread that becomes the lock's owner can continue the execution, and others have to wait for the lock to be released. However, the order of threads executing in serial order can be arbitrary.

Minor modifications of a TM implementation with the global lock provide deterministic execution. Adding a global counter for counting beginnings of transactions (`begin_counter`) and a condition:

`begin_counter % thread_number == thread_id` (1)

ensures that only one thread can acquire the global lock. Other threads spin.

In the codition, `thread_number` is the total number of threads running, and `thread_id` is in the range [0, `thread_number`-1].

However, this solution is only sufficient when all threads have the equal number of transactions to execute. If that is not the case, one thread can finish its execution and exit, and the other threads spin and wait on the condition that is true only for the finished thread. Therefore, we implemented a sorted list with

thread ids (`list_ids`) to maintain the round-robin order. An id is added when a thread is created, and it is removed at the end of the round in which the thread finished its execution. We use `turn` instead of `begin_counter`, and change the condition (1) to:

`list_ids[turn] == thread_id` (2)

At the end of a transaction, a thread increments the `turn` value (modulo number of threads), so the next thread can acquire the global lock.

This implementation provides no parallelism, and therefore, we propose three other implementations to rely on TM functionalities and exploit parallelism in deterministic execution.

2.2 Write-with-token

In this implementation threads are allowed to execute **write operations** only when it is their turn, i.e. when they have the token. Otherwise, they spin.

Transactions execute in parallel if they access thread-local data or if they read shared data. However, when they try to update shared data for the first time, they will spin until they get the token. With the token, a transaction executes, commits and passes the token.

In Figure 1 we show an example where three threads access global variables `a`, `b` and `c` within transactions. The code we show is before and after a TM compiler transformation: all memory accesses to shared variables within transactions are transformed to transactional memory accesses, i.e. `TX_READ` and `TX_WRITE`.

In Figure 2 we show pseudocode of a write-through TM implementation with modifications to provide deterministic execution. Our modifications include: a thread has a permission to write shared data or to commit changes only if it is its turn (lines 2 and 8), i.e. it is the owner of a single global token, and the turn is changed at the end of the commit phase (the line 12), so the next thread in the round-robin order can execute its transaction.

Read-only transactions benefit from this implementation since they can execute in parallel and only wait at commit. However, a thread that does not

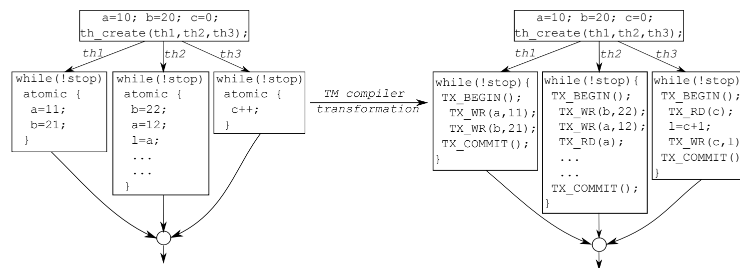


Fig. 1: An example of three threads running transactions in parallel - pseudo source code and compiler's transformed code.

```
1  TX_WR(var,val){
2   while(list_ids[turn]!=this.id){} //spin until you get the token
3   while(!try_set_owner(var)){}     //set your ownership of the data
4   var = val;                       //write to memory
   }

7  TX_COMMIT(){
8   while(list_ids[turn]!=this.id){} //spin until you get the token
9   CAS(status, running, committed); //change the status to committed
10  validate_reads();                //validate all reads
11  release_owner();                 //release all your ownerships
12  (turn++) % list_ids.size;        //pass the token
   }
```

Fig. 2: Write-with-token implementation. TM functions TX_WR (transactional write) and TX_COMMIT (commit of a transaction) modified to provide deterministic execution.

```
1  TX_WR(var,val){
2   if(list_ids[turn]==this.id)      //if you have the token,
3     if(get_owner(var)!= this.id)   //but you are not the owner of data
4       TX_ABORT(get_owner(var));    //abort the owner of the data
5   while(!try_set_owner(var)){}     //set your ownership of the data
6   var=val;                         //write to memory
   }

9  TX_COMMIT(){
10  while(list_ids[turn]!=this.id){} //spin until you get the token
11  CAS(status, running, committed); //change the status to committed
12  validate_reads();                //validate all reads
13  release_owner();                 //release all your ownerships
14  (turn++) % list_ids.size;        //pass the token
   }
```

Fig. 3: Abort-others implementation. TM functions TX_WR (transactional write) and TX_COMMIT (commit of a transaction) modified to provide deterministic execution.

conflict with the owner of the token will spin when it tries to update another shared data (in our example - the thread *th3* from Figure 1). To provide more parallelism, we propose Abort-others and Write-X-times.

2.3 Abort-others

In this implementation all threads speculatively run in parallel and our system guarantees that the owner of the token always executes with progress and finally commits. If there is a conflict with another thread, the system aborts the other thread. Identically to *Write-with-token*, the owner of the thread releases the token at the commit time. Figure 3 shows our modifications (lines 2, 3, 4, and 10, 14).

```
1  TX_WR(var,val){
2   if(list_ids[turn]==this.id)       //if you have the token,
3     if(get_owner(var)!= this.id)    //but you are not the owner of data
4       TX_ABORT(get_owner(var));     //abort the owner of the data
5   while(!try_set_owner(var)){}      //set your ownership of the data
6   var=val;                          //write to memory
7   wr_counter++;                     //increment the counter of writes
   }

10 TX_COMMIT(){
11  while(list_ids[turn]!=this.id){} //spin until you get the token
12  CAS(status, running, committed); //change the status to committed
13  validate_reads();                //validate all reads
14  release_owner();                 //release all your ownerships
15  if(wr_counter >= X){             //if X number of writes were executed
16    wr_counter=0;                  //reset the counter of writes
17    (turn++) % list_ids.size;      //pass the token
    }
   }
```

Fig. 4: Write-X-times implementation. TM functions TX_WR (transactional write) and TX_COMMIT (commit of a transaction) modified to provide deterministic execution.

2.4 Write-X-times

In the previous implementation, the thread *th3* from Figure 1 contains a short transactions with only one update and has to wait for its turn for a long time just to be able to commit the short transaction once.

In this proposal, we provide a fair token possession. The owner of the token executes at least the X number of write operations before it releases the token. When the owner executes X write operations, it continues the execution until the next commit, where it releases the token. The modifications are based on *Abort-others* and extended by the counter for write operations (Figure 4, lines 15, 16, 17).

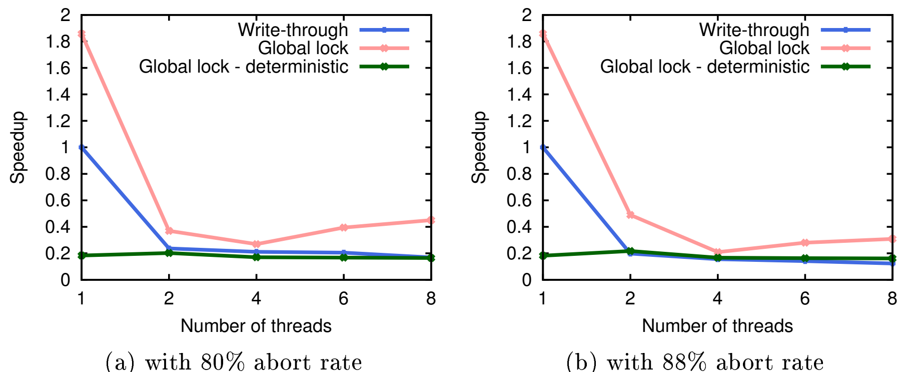
2.5 Strong Determinism

Since the TM library provides weak atomicity and does not instrument memory accesses out of transactions, our modifications presented so far provide only weak determinism. However, an application can have data races for memory accesses out of transactions. Different interleaving of these accesses might lead to non-deterministic execution.

To prevent data races and to provide full determinism of a system, we propose wrapping non-transactional code into new transactions and executing them in the deterministic order.

3 Description of the main results obtained

Figure 5 shows preliminary evaluation of the implementation of deterministic execution with the global lock (*Serial-run*). In our example, threads execute small conflicting transactions with one counter increment. The number of



(a) with 80% abort rate

(b) with 88% abort rate

Fig. 5: Performance of a counter increment example. Since it is a modified existing global lock TM implementation, it performs worse than it, but due to high contention, in some cases it performs better than the write-through implementation.

threads is: 1, 2, 4, 6, or 8. TinySTM++ (C++ implementation of TinySTM [3]) Write-through implementation shows a significant abort rate (for 8 threads running, up to 80% (left) and 88% (right)). We compare three implementations of TinySTM++: Write-through, Global lock and Global lock - deterministic. We show speedup normalized to single-threaded Write-through execution. Our modifications slowdown the original Global lock implementation. However, due to the high contention running with Write-through, Global lock performs better, and even Global lock - deterministic performs better in some cases.

For a benchmark with the abort rate less or equal than 80%, Global lock - deterministic performs worse than Write-through, e.g. with the abort rate exactly 80%, Global lock - deterministic is 2.8% slower for 8 threads (Figure 5a). But if the abort rate increases, Global lock - deterministic performs better, e.g. with the abort rate 88%, it is 23.9% faster for 8 threads (Figure 5b).

Serial-run provides deterministic execution with low or no overhead for benchmarks with very high contention. For benchmarks with lower contention, we suggest other three implementations.

4 Future collaboration with host institution

The participant sides (BSC and TUD) intent to continue the collaboration on deterministic execution. Apart from finishing the implementation and evaluation parts of this work, we want to 1) investigate other deterministic orders, rather than using only round-robin, 2) provide partially parallel commits, and 3) and finally publish a joint paper containing the work started in this STSM.

5 Confirmation by the host institution of the successful execution of the STSM

Prof. Christof Fetzer has sent the confirmation directly to the STMS coordinators.

References

1. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th annual international symposium on computer architecture. ISCA '93 (1993) 289–300
2. Harris, T., Larus, J., Rajwar, R.: Transactional memory. Synthesis Lectures on Computer Architecture **5**(1) (2010) 1–263
3. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. PPoPP '08, New York, NY, USA, ACM (2008) 237–246
4. Felber, P., RivieÌĂre, E., Moreira, W., Harmanci, D., Marlier, P., Diestelhorst, S., Hohmuth, M., Pohlack, M., Cristal, A., Hur, I., Unsal, O., StenstroÌĹm, P., Dragojevic, A., Guerraoui, R., Kapalka, M., Gramoli, V., Drepper, U., TomicÌĄ, S., Afek, Y., Korland, G., Shavit, N., Fetzer, C., Nowack, M., Riegel, T.: The velox transactional memory stack. Micro, IEEE **30**(5) (sept.-oct. 2010) 76 –87
5. Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G.T., Boyle, P., Chist, N., Kim, C.: The ibm blue gene/q compute chip. Micro, IEEE **32**(2) (march-april 2012) 48 –60
6. J., R.: Transactional synchronization in haswell. (february 2012)
7. Yalcin G., Unsal O., C.A.: Faultm: Error detection and recovery using hardware transactional memory. The Design, Automation, and Test in Europe Conference (DATE) (march 2013)
8. Devietti, J., Lucia, B., Ceze, L., Oskin, M.: Dmp: deterministic shared memory multiprocessing. In: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems. ASPLOS '09, New York, NY, USA, ACM (2009) 85–96
9. Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. SIGPLAN Not. **44**(3) (march 2009) 97–108
10. Bergan, T., Anderson, O., Devietti, J., Ceze, L., Grossman, D.: Coredet: a compiler and runtime system for deterministic multithreaded execution. In: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems. ASPLOS '10, New York, NY, USA, ACM (2010) 53–64
11. Devietti, J., Nelson, J., Bergan, T., Ceze, L., Grossman, D.: Rcdc: a relaxed consistency deterministic computer. SIGPLAN Not. **47**(4) (2011) 67–78