CONSISTENT STATE SOFTWARE TRANSACTIONAL MEMORY

Gonçalo Cunha João M. S. Lourenço Ricardo Dias
CITI–Centro de Informática e Tecnologias de Informação, and
Departamento de Informática, Universidade Nova de Lisboa
goncalo.cunha@gmail.com joao.lourenco@di.fct.unl.pt rjfd@di.fct.unl.pt

Keywords: Software Transactional Memory, Concurrency Control, Multicore Programming.

Abstract: Software transactional memory (STM) is a promising programming model that adapts many concepts borrowed from the databases world to control concurrent accesses to memory (RAM) locations. In this paper we propose a new classification for the active states of a transaction; a new memory quiescing algorithm, to allow the safe transition of a memory block form transactional to non-transactional space; we compare word and object transactional grain units; and evaluate the cost of consistent state validation, arguing that this cost can be minimized by performing partial validation on problematic code regions.

1 INTRODUCTION

The current trend of having multiple cores in a single CPU chip is leading to a situation many would believe absurd not long ago: we may have more processing power then we can use. To invert such situation we need to find and explore concurrency where before was sequential code. The transactional programming model is an appealing approach as it deals with concurrency by making use of high-level constructs. The transactional programming model may or may not count with support from the underlying hardware, most specially the CPU. The term *software transactional memory* (STM) refers to the transactional programming model when there is no special support from the hardware.

Due to performance reasons, most existing STM implementations use optimistic methods for concurrency control. Such methods allow transactions to conflict with each other and conflict detection may occur when the conflicting operation is executed or later at commit time, depending on the STM design and implementation options. If the conflict is not detected when the conflicting operation is executed, transactions will execute temporarily in an inconsistent state. This may lead to many hazards such as infinite loops, dereferencing invalid pointers or fail algorithmic invariants that would not happen if the transaction would execute only in consistent states.

Section 2 briefly describes the main approached towards software-based implementations of the transactional programming model. Section 3 introduce the obsolete consistent state, the new quiescing algorithm and describes the object-mode variants. Section 4 presents the experimental results and Section 5 summarizes our results.

2 STM IMPLEMENTATION ALTERNATIVES

Other literature [1] covers in detail the different approaches towards STM implementation. In this Section we will concentrate on those that are relevant for the work described ahead.

STM Operational Model
Within the context of a transaction, any memory change should be considered tentative, thus the STM engine must be able to restore the previous value if the transaction aborts. Blocking STMs use two main operational models: direct [2, 3] and deferred [4, 5, 6, 7, 8] update.

Concurrency Control
Due to performance reasons, most STM systems use optimistic concurrency control for reads with versioned write locks. Each memory location has a version number that is incremented for every commit updating it. When a transactional read is performed, the version number of the variable is stored in a transaction local data structure (the *read-set*). On commit, the version of variables listed in the read-set are checked against the recorded version. If any of variables has changed, the transaction is in an invalid state and must be rolled back. This strategy prevents non-serializable orderings from being committed.

Transactional granularity
Transactional memory locations may have differ-

ent levels of granularity: it may be at the object level (object-based STMs) or block level (word-based STMs). With block level granularity, the block may have the size of a memory word or the size of a cache line, while in the object-based approach the grain may vary with the object being protected. Word based STMs have more room for concurrency as the granularity is finer, each word has its own metadata and collisions are less frequent. However the overhead introduced to manage these fine grain units is higher, since the transactional API must be called once for every field (instead of every object) the transaction accesses. Object based STMs have metadata shared for all the fields of the object. They have a coarser granularity than Word based STMs, as the transactional API is called only once for every object. Simultaneous accesses from different transactions to different fields of the same objects will conflict. Object based STMs are less suitable for accesses to transactional variables that are not objects, like single variables and arrays. Some Object-based STM systems do not allow this kind of accesses, while others treat an array as a single object. Word based STM, on the other hand, can easily access single variables and arrays word-by-word.

Lock Placement
There are two main strategies for lock placement on lock based STMs: adjacent to the data; or in a separate table.

Placing the lock adjacent to the data has the advantage of having a higher chance that both, the lock and the data, will stay on the same cache line. This may reduce the number of cache misses, yielding a better performance. However, placing the lock and data next to each other, requires changes to the structure of the objects on the heap. Also, the object handling is different. Without support from the compiler, the developer holds a pointer to the header and, to access the data itself, the pointer is incremented by the header size. These changes limit the re-usage of existing libraries, as they change both the structure of the objects and the way they are handled.

Placing the locks in a separate table does not require changes to the heap objects, nor to the way they are handled. With this technique it is easier to reuse existing libraries, as it only requires the accesses (reads/writes) to be instrumented. To map a variable to the lock in the lock table, it is necessary to have a hash function that maps the variable address to the table position.

3 NEW PROPOSALS FOR STM

TL2 [9] introduced two interesting features: 1) it doesn't require the use of specialized malloc/free implementations (even for non garbage collected languages) and allows memory to be recyclable between transactional and non-transactional space; and 2) it uses the global version clock algorithm to prevent user code from running in an inconsistent state. However, it was only able to provide both features with Word based and Deferred Update mode (or commit time locking as described by the authors).

Our work was based in a TL2 implementation. On the port to Linux/X86-32,64/GCC, the TL2 scheduler control was removed, since this is not available on Linux. Also the SPARC non-faulting load instruction (not available in the X86 processors) was replaced with fault handlers. The synchronization instructions were inspired on the Ennals STM implementation [8].

3.1 Obsolete Consistent Transaction State

When using versioned write locks, transactions may collide with each other and fall into states where the transaction cannot commit. We divide active transaction state into three categories: *updated consistent*, *obsolete consistent* and *inconsistent*, as illustrated in Fig 1.

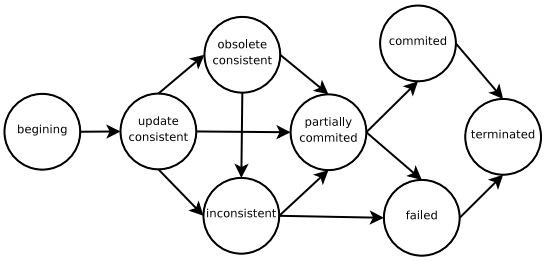

Figure 1: Transaction states.

A transaction running in an updated consistent state is one in which all reads have returned all the latest committed values and these values haven't been updated since the read. Transactions that try to commit with an updated consistent memory snapshot will succeed.

A transaction is in an obsolete consistent state if it has a valid memory snapshot, but the snapshot itself is outdated because it was updated by some other transaction. Figure 2 show an interleaving leading T_A to an obsolete consistent state at step 6. A transaction running in a obsolete consistent state has a correct behavior but will fail to commit.

A transaction is in an inconsistent state if the read values do not correspond to a valid memory snapshot. It may be due the reading dirty values or read-after-write hazards, e.g., two consecutive reads to the same variable return different values because it was updated by another transaction. The transaction may have unpredictable results but in the end it will abort.

	Transaction A	Transaction B
1	`atomic {`	
2	`  a=x;`	
	`.....`	`.....`
3		`atomic {`
4		`  x ++;`
5		`}`
	`.....`	`.....`
6	`  y=z;`	
7	`}`	

Figure 2: Obsolete consistent state: on step 6, transaction A is consistent but obsolete.

3.1.1 Concurrency Control and Transaction States

When using read write locks, a read is guaranteed to always return a consistent value, as the transaction has acquired a read lock and no other transaction can update the value while the read lock is being held.

When using versioned write locks there is no such guarantee and some hazards may occur. While the transaction is being run, the loads may see dirty variables and/or see inconsistent states as a result of unfortunate interleavings. These transactions will probably abort later at commit time. Although, they may also enter endless loops in the middle of the transaction, dereference invalid pointers, fail assertions that otherwise would be valid if the transaction was being run in a consistent state, etc. Figure 3 shows an example of an interleaving where the invariant $a = b$ should hold for transaction T_A but, as an invalid state was observed, T_A enters an infinite loop.

	Transaction A	Transaction B
1	`//x =0 && y=0`	`//x =0 && y=0`
2	`atomic {`	
2	`  a=x;`	
	`.....`	`.....`
3		`atomic {`
4		`  x ++;`
5		`  y ++;`
6		`}`
	`.....`	`.....`
7	`  b=y;`	
8	`  if(a != b)`	
9	`    while(true){}`	
10	`}`	

Figure 3: Infinite loop caused by improper ordering. Transaction A is inconsistent.

3.2 New Quiescing Algorithm

Transactional variables must only be accessed from within transactions; however, it is desirable that variables leaving the transactional space can be re-used by non transactional operations. To handle such a transition the variable must be quiesced before it is freed. Quiescing on the original TL2 consisted of waiting for the writes to be drained and the lock to be released. This however fails under some circumstances. Consider the situation of a list with two nodes (A and B) being used by three threads (T1, T2 and T3); T1 is looking for node B; T2 is deleting node B; T3 is not running any transaction. The following sequence of events may takes place:

1. T1: starts (with *transaction timestamp* 10) and looks up node A, which is prior to node B (i.e., it reads a pointer to node B);
2. T2: starts, (with *transaction timestamp* 10) looks up node B and removes all references to node B;
3. T2: commits (increments the global clock to 12[1]);
4. T2: quiesces node B (no thread is currently locking B);
5. T2: frees node B;
6. T3: starts, calls malloc and receives a pointer to the *same memory where node B was previously referenced*;
7. T3: writes to that memory;
8. T1: follows the pointer to the late node B and reads its contents. At this point T1 is reading memory already recycled for usage of another thread. Hence T1 is running on an inconsistent state, which violates the idea of never seeing inconsistent memory states.

The solution found was for the quiesce operation to treat the delete as a regular write. In other words, quiesce updates the lock version to the current value of the global clock. It is like a mini transaction writing to the lock of the node to be deleted. Since quiesce is always called after the commit and before the free, updating the nodes version number will invalidate any further reads to the node by all other transactions. Quiesce does not need to increment the value of the global clock because it was already incremented when the commit was done (quiesce can not be called inside an active transaction).

With the new quiesce, the previous operations would be transformed into the following:

1. Steps 1 to 7: same as above;
8. T1: follows the pointer to the late node B and verifies that B's lock is greater (12) that its own transaction timestamp (10). T2 aborts and the user code never sees an inconsistent state.

[1]Clock is always incremented by 2, odd numbers mean the lock is held, even number mean the lock is acquired

Even if some other transaction incremented the clock between step 3 and 4, the result would be the same.

3.3 Object Based Direct Update with Partial and Full Validation

This mode is based in the same algorithms as the word based modes. The main differences are on the API to handle the objects and on the fact that the read and write operations are made directly to the objects.

3.3.1 Algorithm

The API for object based mode differs from the word based mode API, and its basis is similar to many other object-based STMs [8]. Before and object is read *TxOpenRead* is called. It verifies if the lock version of the objects is greater than the transaction timestamp. If that's the case the transaction aborts, otherwise it records the read operation in the read set. After calling *TxOpenRead*, the object can be directly accessed for read. However, each object read may render the transaction to an inconsistent state, as it may have been changed by another transaction. To avoid this, the transaction must call *TxVerifyAddr*. *TxVerifyAddr* checks again if the lock version of the objects is grater than the transaction timestamp. If it is not, the object has not been changed since the beginning of the transaction and the transaction continues to be consistent. The macro *TxReadField* does the last two steps, returning the field value and verifying if the state continues to be consistent.

```
TxOpenRead(Thread *t, void *addr);
TxOpenWrite(Thread *t, void *addr, int size);
TxVerifyAddr(Thread *t, void *addr);
#define TxReadField(t,addr,field)
```

Figure 4: Simplified API for handling objects in object based mode

In fact *TxVerifyAddr* does not enforce the transaction to terminate with a correct result; it only guarantees that the transaction is running in a consistent memory states. The other STMs that use one version counter per variable (e.g., [8]) may also do this by re-validating the read set after every read, but it would have a cost of $O(n^2)$, where n is the number of transactional reads. Our prototype does this validation with a cost of $O(n)$.

3.3.2 API Usage Example

Next, follows two examples showing how the prototype API may be used. Both illustrate a part of a transaction reading the key and value of a list node. After the read, an assertion is made to verify the invariant that the key must be greater than zero. The assertion must be checked within a consistent state; otherwise, interleavings with other transactions may fail the assertion even if it is valid according to the algorithm.

The first example in Fig. 5 shows the validation being performed after the read of key and value. This approach however allows the transaction to run inconsistent in steps 2 and 3. After the re-validation on step 4, the transaction is again consistent. The second example in Fig. 5 shows the validation being performed on every read. This approach makes the transaction always run in a consistent state, at the cost of additional checks.

```
   ...
1  TxOpenRead(t, node);
2  key = node->key;
3  value = node->value;
4  TxVerifyAddr(t, node);
5  assert(key>0);
   ...
```

```
   ...
1  TxOpenRead(t, node);
2  key = TxReadField(t,node,key);
3  assert(key>0);
4  value = TxReadField(t,node,value);
   ...
```

Figure 5: Explicit consistent state validation

The second approach is simpler to be used by the programmer, because there is not need to keep track of which objects have to be verified for consistency.

The first approach has less overhead due to the fewer number of verifications made. Yet, it is more complex for a programmer to use it directly. It is better suited for compilers that generate the transactional code from a higher level language [10]. The compiler may validate the transaction state only before some event that needs a consistent state, e.g., before an assertion, dereferencing a pointer, or testing a loop condition.

Since this mode locks the objects when they are opened for write, the reads don't need to be validated on objects opened for write (Fig. 6).

```
   ...
1  TxOpenWrite(t, node, sizeof(node_t));
2  key = node->key;
3  value = node->value;
4  assert(key>0);
   ...
```

Figure 6: Consistent state validation for locked variables

4 EXPERIMENTAL RESULTS

Since our prototype is based on TL2, we decided to use a similar test harness. Our tests were made with a Red Black Tree implementation based on the one

found on TL2 package. Several modifications were made to adapt it to use the different API of object mode.

The tests consist on series of operations on a STM based Red Black Tree. The implementation has three methods: add, delete and get. The tree nodes have a key and a value and all methods are indexed by the key. Duplicate keys are not allowed and adding an already existing key just updates the node value.

The test harness launches a number of threads in parallel. Each thread randomly chooses an operation to execute (add, delete, or get), then generates a random key and executes the selected operation on that key. The operations are chosen based on probability factors given on the command line. The key range also passed as a command line argument and it limits the maximum number of elements in the set.

The tests were performed on a Sun Fire X4600 M2 x64 server with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz with 1024 KB cache.

4.1 Comparing Direct/Deferred and Word/Object modes

The first set of tests (Fig. 7) intend to compare the performance of direct/deferred strategies and word/object based modes. We used the same test loads than [2].

The tests include a small (200 keys) and a large (20.000 keys) red black tree. The small tree represents a high contention structure and the large tree represent a low contention structure. Each tree is subject to two load patterns, one with mostly reads (writes are 2-3%), other with a higher write percentage (6-12%). The first pattern is 5% puts; 5% deletes; 90% gets. The second pattern is 20% puts; 20% deletes; 60% gets. The test loads included 1, 2, 4, 8 and 16 threads. In this way is possible to evaluate the scalability as the number of available CPUs increase. Tests were performed using the three combinations of our prototype. The Object based STM tests were performed with partial verification (on the required places to prevent the transaction from entering infinite loops and fail several assertions made on the code).

All tests show that word based direct (undo log) and deferred (redo log) update modes perform similarly in all scenarios. But the deferred update strategy scales better in high contention environments, while the direct update strategy performs better in low contention environments.

All tests show that, with a high number of threads, object mode with partial verification performs worst than word mode with direct and deferred strategies. Although having a smaller overhead (there is only one

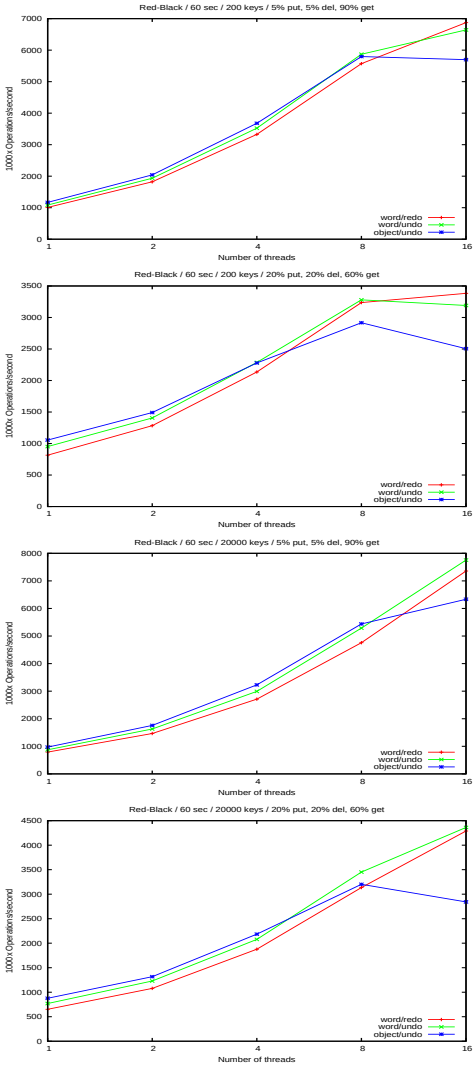

Figure 7: Evaluation of implementation alternatives.

lock and unlock per node instead of one per field), the lock granularity is coarser in object mode than in word mode, which limits concurrency and lowers its scalability for a higher number of threads.

4.2 The cost of consistent state validation

The second set of tests (Fig. 8) evaluate the cost of consistent state validation in object mode (with a direct update strategy). Experiments show that partial validation regularly performs better than full validation by a factor of 10% to 15%. Experiments also show evidence that the cost of verification is benevolent, as it allows to detect inconsistent states sooner and, thus, transactions do less useless work. The *no verification* mode only detects inconsistencies at

commit time and, therefore, has to re-execute the entire transaction again, imposing a considerable performance penalty when compared to the other modes.

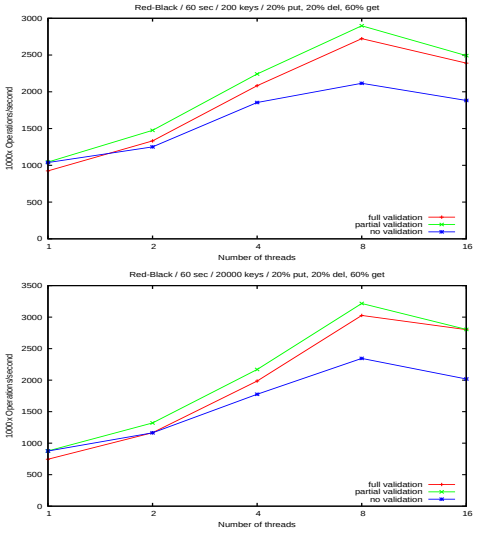

Figure 8: Cost of verification on small and large Red Black Trees

5 CONCLUSIONS

We have presented an initial study of several implementations options for STM engines with emphasis in running transactions in consistent states. We evaluated the direct/deferred update strategies and word/object modes. The benchmarks show that both update strategies perform similarly on word based mode and both scale better than object mode. There is also evidence that in object mode the overhead introduced by validating frequently the transaction can avoid to performing useless computations and achieve good performances.

Acknowledgements

This work was partially supported by Sun Microsystems under the "Sun Worldwide Marketing Loaner Agreement #11497", by the CITI–Centro de Informática e Tecnologias da Informação and by the FCT/MCTES–Fundação para a Ciência e Tecnologia in the context of the Byzatium research project and of grant SFRH/BD/41765/2007.

References

[1] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[2] David Dice and Nir Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.

[3] Bratin Saha et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[4] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[5] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM.

[6] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

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

[8] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[10] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.