Transactional Memory

Boris Korenfeld
Moti Medina
Computer Engineering
Tel-Aviv University

June, 2006

Abstract

Today's methods of writing programs that are executed on shared-memory multiprocessors systems rely mostly on locking mechanism.
On one hand, those methods enable programmers to write correct multi-threaded programs that access shared resources; on the other hand, lock usage reduces performance due to suboptimal use of them.
In this survey paper we will show several implementations of ***Transactional Memory (TM)*** *that enables execution of multithreaded programs without using a locking mechanism, thus improving performance. There are several approaches to handle that issue. Some of them improve the performance by eliminating the locks, whereas others require programmers to write "transaction oriented programs".*
In ***TM*** *systems, every processor speculatively executes transactions. If data conflict occurs, several processors will undo their changes and restart their transaction according to an arbitration mechanism.*

1 Introduction

Transactional Memory (TM) is a multiprocessor architecture that eliminates the current need for locks and semaphores for synchronization, thus eliminating the problems that locks and semaphores usage causes like *Priority inversion*, *Convoying* and *Deadlocks* [2].
It is essential that in a survey paper that handles TM some basic terms be reminded:

- *Transaction* is a term used in databases that refer to the finite sequence of instructions executed together and having the properties of *Serializability* and *Atomicity*.
- *Serializability* means that instructions from two different transactions will never be interleaved.
- *Atomicity* means that a transaction attempts to make a sequence of changes to the shared memory and either commits all the changes, or aborts all of them.
- All memory references which were read by a transaction are called a *read set* and all memory references which were written by it are called a *write set*. The union of these two is called the *data set.*

Today's methods of writing programs that are executed on shared-memory multiprocessors systems rely mostly on locking mechanism. On one hand, those methods enable programmers to write correct multi-threaded programs that access shared resources; on the other hand, lock usage reduces performance due to suboptimal use of them. There are several approaches to handle that issue. Some of them improve the performance by eliminating the locks, whereas others require programmers to write "transaction oriented programs".

We will begin with a premature implementation of the first approach, *Speculative Lock Elision (SLE)* [3], a technique to remove dynamically unnecessary locks and enable highly concurrent multithreaded execution.

We will then continue with an improvement of the latter, *Transactional Lock Removal* [4], such that removes locks entirely.

Transactional Memory expands regular coherence protocols from treating a single block to treating a whole transaction. Transactional memory works in the similar way as in databases. Transactions are speculatively executed in different threads on different processors, but they have to commit. Only one transaction can commit in a single step, whereas other transactions that read\write the same memory block are forced to restart.

Transactional memory Coherence and Consistency (TCC) [5, 6] presents a completely different approach. Instead of eliminating existing locks, TCC requires a "transaction oriented programming". TCC model tend to take advantage of shared memory thus producing an illusion of common memory, but on the other hand it wants to get the relaxed latency as message passing does. TCC replaces the conventional block treating coherence protocols. TCC uses the concepts provided in the earlier work on Transactional Memory [2] and during that article it will be compared to that early proposal.

In order to take advantage of the proposed TCC model, two programming structures were suggested: one for loop based parallelization and second for fork based parallelization [6].

One serious disadvantage of proposed TCC is the assumption that transactions will be relatively small. This assumption is only partially true. While vast majority of transactions are small there are a few very large transactions that should be treated as well. Solution to that problem can be found in the *advanced TM models* (chapter 5 in this paper) such as: UTM, LTM [8], VTM [9], LogTM [10].

Advanced TM models allow transactions to be suspended, to migrate, or to overflow state from local buffers. Such abilities require decoupling transactional state from processor state. VTM uses *lazy version management* and *eager conflict detection*, however ideally, *transactional memory* should use *eager version management* and *eager conflict detection* (those terms will be defined on chapter 5).To this end *Log-based Transactional memory (LogTM)* [9] is proposed with the advantage of relying on existing coherent and consistency protocols, instead of replacing them using a complex hardware like UTM does.

The rest of the paper is organized as follows:
Chapter 2 describes SLE [3]. TLR [4] is summarized in chapter 3. Chapter 4 summarizes TCC [6]. Advanced TM models described in chapter 5.
Finally in Chapter 6, we conclude with a table that compares all the surveyed TM models.

2 Speculative Lock Elision

2.1 The Problem

In multithreaded programs, synchronization mechanisms, usually locks, are often used to guarantee threads have exclusive access to shared data for a critical section of code. A thread acquires the lock, executes its critical section, and releases the lock.
All other threads wait for the lock until the first thread has completed its critical section, serializing access and thus making the entire critical section appear to execute atomically.
For a variety of reasons, concurrent accesses to a shared data structure by multiple threads within a critical section may in fact not conflict, and such accesses do not require serialization, for example:

```
1.LOCK(mutex)
2.if (input_grade > University->max_grade)
3.      University->max_grade = input_grade;
4.UNLOCK(mutex)
```

Since a store instruction (line 3) to a shared object is present, the lock is required. However, most dynamic executions do not perform the store operation and thus do not require the lock.

Additionally, multiple threads may update different fields of a shared object, while holding the shared object lock, and often these updates do not conflict, for example, Updates of a simple `C struct`:

Thread #l

```
LOCK(example_struct.mutex)
ID = example_struct.ID;
if (ID < 0)
    example_struct.ID = INPUT_ID;
UNLOCK(example_struct.mutex)
```

Thread #2

```
LOCK(example_struct.mutex)
GRADE = example_struct.GRADE;
if (GRADE < 100)
   example_struct.GRADE = 100;
UNLOCK(example_struct.mutex)
```

In the last two examples, conventional speculative execution in out-of-order processors cannot take advantage of the parallelism present because the threads will first wait for a free lock and then acquire the lock in a serialized manner.

2.2 Where can we influence? (Or what is the Solution?)

The key insight is that locks do not always have to be acquired for a correct execution.

Ideally, programmers would be able to use frequent and conservative synchronization to write obviously correct multithreaded programs, and a tool would automatically remove all such conservative use. Thus, even though programmers use simple schemes to write correct code, synchronization would be performed only when necessary for correctness; and performance would not be degraded by the presence of dynamically unnecessary synchronization.

In *Speculative Lock Elision (SLE)* [3], the hardware dynamically identifies synchronization operations, predicts them as being unnecessary, and elides them. By removing these operations, the program behaves as if synchronization were not present in the program. Of course, doing so can break the program in situations where synchronization is required for correctness. Such situations are detected using pre-existing cache coherence mechanisms and without executing synchronization operations. In this case, recovery is performed and the lock is explicitly acquired. Synchronization is performed only when the hardware determines that serialization is required for correctness.

SLE has the following key features:

1. **Enables highly concurrent multithreaded execution:**
   Multiple threads can concurrently execute critical sections guarded by the same lock.
2. **Simplifies correct multithreaded code development:**
   Programmers can use conservative synchronization to write correct multithreaded programs without significant performance impact. If the synchronization is not required for correctness, the execution will behave as if the synchronization were not present.
3. **Can be implemented easily:**
   SLE can be implemented entirely in the micro-architecture, without instruction set support and without system level modifications (e.g. no coherence protocol changes are required) and is transparent to programmers.
   Existing synchronization instructions are identified dynamically.
   Programmers do not have to learn a new programming methodology and can continue to use well understood synchronization routines. The technique can be incorporated into modern processor designs, independent of the system and the cache coherence protocol.

2.3 Speculative Lock Elision algorithm

Atomicity means all changes performed within a critical section appear to be performed instantaneously. By acquiring a lock, a thread can prevent other threads from observing any memory updates being performed within the critical section. While this conventional approach trivially guarantees atomicity of all updates in the critical section, it is only one way to guarantee atomicity.
Locks can be elided and critical sections concurrently executed if atomicity can be guaranteed for all memory operations within the critical sections **by other means**.
For guaranteeing atomicity, the following conditions must hold within a critical section:

1. Data read within a speculatively executing critical section is not modified by another thread before the speculative critical section completes.
2. Data written within a speculatively executing critical section is not accessed (read or written) by another thread before the speculative critical section completes.

A processor can provide the appearance of atomicity for memory operations within the critical section without acquiring the lock by ensuring that partial updates performed by a thread within a critical section are not observed by other threads. The entire critical section appears to have executed atomically, and program semantics are maintained.
The key observation is that a lock does not always have to be acquired for a correct execution if hardware can provide the appearance of atomicity for all memory operations within the critical section.
If a data conflict occurs, i.e., two threads compete for the same data other than for reading, atomicity cannot be guaranteed and the lock needs to be acquired. Data conflicts among threads are detected using existing cache protocol implementations.
Any execution not meeting the above two conditions is not retired architecturally, thus guaranteeing correctness.

Algorithmically:

1. When a lock-acquire operation is seen, the processor predicts that memory operations in the critical section will occur atomically and elides the lock acquire.
2. Execute critical section speculatively and buffer results.
3. If hardware cannot provide atomicity, trigger miss-speculation, recover and explicitly acquire lock.
4. If the lock-release is encountered, then atomicity was not violated (else a miss-speculation would have been triggered earlier). Elide lock-release operation, commit speculative state, and exit speculative critical section.

Eliding the lock acquire leaves the lock in a FREE state, allowing other threads to apply the same algorithm and also speculatively enter the critical section. Even though the lock was not modified, either at the time of the acquisition or the release, critical section semantics are maintained.
In step 3, the processor can alternatively try to execute the algorithm again a finite number of times before explicitly acquiring the lock. Forward progress is always guaranteed because after the restart threshold is reached, the lock is explicitly acquired.
The above algorithm requires processors to recognize lock-acquire and release operations

2.4 Speculative Lock Elision, Is it so Perfect?

SLE is a hardware proposal for eliding lock acquires from a dynamic execution stream, thus breaking a critical performance barrier by allowing non-conflicting critical sections to execute and commit concurrently. SLE showed how lock based critical sections can be executed speculatively and committed atomically without acquiring locks if no data conflicts were observed among critical sections.
A data conflict occurs if, of all threads accessing a given memory location simultaneously, at least one thread is writing to the location. While SLE provided concurrent completion for critical sections accessing disjoint data sets, data conflicts result in threads restarting and acquiring the lock serially. **Thus, when data conflicts occur, SLE suffers from the key problems of locks due to lock acquisitions.**
A Solution to that problem will be given on chapter **3 (Transactional Lock-Free Execution of Lock-Based Programs).**

3 Transactional Lock-Free Execution of Lock-Based Programs

3.1 The Problem or Why SLE is not perfect?

We seek to improve multithreaded programming trade-offs by providing architectural support for optimistic lock-free execution.
In a lock-free execution, shared objects are **never locked** when accessed by various threads. While SLE (see chapter 2 in this paper) provided concurrent completion for critical sections accessing disjoint data sets, data conflicts result in threads restarting and acquiring the lock serially. Thus, when data conflicts occur, SLE suffers from the key problems of locks due to lock acquisitions.
We must then, refine the SLE method.

3.2 The Solution or Where can we do better?

A refinement of SLE must include a lock-free method, such a constraint produces another problem, let us discuss it.

An execution of an optimistic lock-free transaction can be made *serializable* if the data speculatively modified by any transaction are not exposed until after the transaction commits and no other transaction writes to speculatively read data. A *serializable* execution can be achieved trivially by acquiring exclusive ownership of all required resources. If the thread executing the transaction does so for all required resources, the thread can operate upon the resources and then commit the updates atomically and instantly, thus achieving *serializability.*
In cache-coherent shared-memory multiprocessors the above requires:

1) Acquiring all required cache blocks (that are accessed within the transaction) in an appropriate ownership state.
2) Retaining such ownership until the end of the transaction.
3) Executing the sequence of instructions forming the transaction.
4) Speculatively operating upon the cache blocks if necessary.
5) Making all updates visible atomically to other threads at the end of the transaction.

However, *livelock* prevents resources from being retained thus preventing successful execution of the lock-free transaction.
Livelock can occur if processors executing critical sections speculatively and in a lock-free manner repeatedly experience conflicts (as with SLE, the lock can always be acquired and forward progress is guaranteed but we require a solution that does not rely on lock acquisitions), for example:
Consider two processors, P1 and P2. Assume both P1 and P2 have elided the lock (using SLE) and are in optimistic lock-free transaction execution mode. Both processors are accessing (and writing) shared memory locations A and B in the critical sections. The two processors write the two locations in reverse order of each other, P1 writes A first and then B while P2 writes B first and then A. That scenario result with a sequence that may occur indefinitely with no processor making forward progress because each processor repeatedly restarts the other processor.
Livelock occurs because neither processor obtains ownership of both cache blocks simultaneously. To ensure *livelock* freedom, among competing processors **one processor must win the conflict and retain ownership**.
Let us conclude that a new scheme should include a **resolution mechanism**.

3.3 Does TLR do the job?

TLR [4] assigns priorities to the lock-free transactions and employs the following key idea:
Transactions with higher priority never wait for transactions with lower priority.
In the event of a conflict, the lower priority transaction is restarted or forced to wait.
Consider two transactions T_1 and T_2 executing speculatively.
Suppose T_2 issues a request that causes a data conflict with a request previously made by T_1, and T_1 receives T_2's conflicting request. The conflict is resolved as follows:
if T_2's priority is lesser than T_1's priority, then T_2 waits for T_1 to complete (T_1 wins the conflict), else T_1 is restarted (T_2 wins the conflict).
For starvation freedom the resolution mechanism must guarantee all contenders eventually succeed and become winners.
We use **timestamps for conflict resolution**, earlier timestamp implies higher priority. Thus, the contender with the earlier timestamp wins the conflict.

TLR elides locks using SLE (see chapter 2 in this paper) to construct an optimistic transaction but in addition also uses a timestamp-based conflict resolution scheme to provide lock-free execution even in the presence of data conflicts.
A single, globally unique, timestamp is assigned to all memory requests generated for data within the optimistic lock-free critical section.
Existing cache coherence protocols are used to detect data conflicts.
On a conflict, some threads may restart (employing hardware miss-speculation recovery mechanisms) but the same timestamp determined at the beginning of the optimistic lock-free critical section is used for subsequent re-executions until the critical section is successfully executed.
A timestamp update occurs only after a successful execution. Doing so guarantees each thread will eventually win any conflict by virtue of having the earliest timestamp in the system and thus will succeed in executing its optimistic lock-free critical section. If the speculative data can be locally buffered, all non-conflicting transactions proceed and complete concurrently without serialization or dependence on the lock. Transactions experiencing data conflicts are ordered without interfering with non-conflicting transactions and without lock acquisitions.

3.4 The Algorithm and a final observation

1. Calculate local timestamp.
2. Identify transaction start:
    a) Initiate TLR mode (use SLE to elide locks).
    b) Execute transaction speculatively.
3. During transactional speculative execution.
    a) Locally buffer speculative updates.
    b) Append timestamp to all outgoing requests.
    c) If incoming request conflicts with retainable block and has later timestamp, retain ownership and force requestor to wait.
    d) If incoming request conflicts with retainable block and has earlier timestamp, service request and restart from step 2b if necessary. Give up any retained ownerships.
    e) If insufficient resources, acquire lock.
        i. No buffer space.
        ii. Operation cannot be undone (e.g. I/O).
4. Identify transaction end:
    a) If all blocks available in local cache in appropriate coherence state, atomically commit memory updates from local buffer into cache (write to cache using SLE).
    b) Commit transaction register (processor) state.
    c) Service waiters if any.
    d) Update local timestamp.

In the TLR algorithm three key invariants must hold:

a) The timestamp is retained and re-used following a conflict-induced miss-speculation.
b) Timestamps are updated in a strictly monotonic order following a successful TLR execution.
c) The earlier timestamp request never loses a resource conflict and thus succeeds in obtaining ownership of the resource.

If TLR is applied, these invariants collectively provide two guarantees:

1. A processor eventually has the earliest timestamp in the system.
2. A processor with the earliest timestamp eventually has a successful lock-free transactional execution.

The two properties above result in the following observation:

In a finite number of steps, a node will eventually have the earliest timestamp for all blocks it accesses and operates upon within its optimistic transaction and is thus guaranteed to have a successful starvation-free lock-free execution.

4 Transactional memory Coherence and Consistency (TCC)

System with multiple processors always engage problem with communication and synchronization of processors. Today there are two common methods to treat these problems: shared memory and message passing.
Shared Memory creates an illusion of one single shared memory to all processors. Whereas that conception eases the programmers work by avoiding manual data distribution, it requires an additional hardware. There are two major ways to keep the coherence in the shared memory: snooping on the shared bus and using directory-based technique over the scalable interconnection network. In cache coherence protocols that using snooping the most up-to-date cache line has a special state, like a Modified state in MESI protocol for example. Directory based cache coherence achieved by tracking the place of the most up-to-date block in a special structure - a Directory. Both these methods require a complex hardware.
Message passing on the other hand didn't require that hardware support. Programmers should combine in a single message all the data they wish to transfer to another processor. These actions require programmers work much harder than in the Shared Memory, however message passing still has two important advantages over Shared Memory. Firstly, message passing require a relatively small bus traffic as there is no signal creation following each load/store like in the shared memory. Secondly, message passing enable synchronization of the processors as the messages are sent and received.
When Shared Memory method used the synchronization achieved by use of locks, semaphores and barriers. Unwise use of these tools may cause serious problems like: *Priority inversion*, *Convoying* and *Deadlock*. Attempt to eliminate the use of locks mechanism by software methods have shown to be very harmful to performance [2]. Thus lock free data structures need a hardware support. TCC architecture provides such support.

4.1 TCC Principles

The TCC tends to take the advantages of both these method and combine them in a single coherence and consistency protocol. On the one hand the TCC will provide the programmers with the illusion of one shared memory, but on the other hand it will provide a way for processors synchronization and lower the bus traffic by tracking a whole transaction instead of every load and store.

There are two methods for validating the transaction [1]:

1. *Backward validation*: the committed transaction validates itself against active transaction and restarts itself if a violation is detected.
2. *Forward validation*: every active transaction validates itself against a committed one and restarts itself if a violation is detected.

The first approach was used in earlier Transactional Memory proposal [2]. The violation was discovered only in the validation stage thus creating an *orphan*. The *orphan* is a transaction that executes after its data was changed by a committed transaction. Of course, *orphan* cannot commit, but it is able to create an erroneous situation like divide by zero, for example. The solution that is proposed here is a frequent use of validation instructions in a transaction code.

The other, much better, approach is used in TCC. Processors in a TCC system will execute the transaction's instructions speculatively. When the transaction finishes the

TCC hardware will arbitrate system wide for giving the transaction the right to commit its stores. If the transaction commits the processor will combine all the stores of the transaction in a single message and broadcast it to the rest of the system. Other processors by snooping will detect those writes and if they recognize that they have used modified data they will have to roll back and restart the transaction [5]. Combining all transaction's read in a single packet is greatly relaxes the bus traffic since there is no need to send a message and make an arbitration for every load/store. Creation of an *orphan* is impossible since every active transaction is being forced to restart when a committed one has made a change to its data set.

TCC enable us to replace the existing coherence and consistency protocols.
The consistency and coherence in TCC kept by order of transactions commits. For the main memory all memory accesses from transactions that committed earlier are happened before all accesses from the transactions that committed later, even if in reality those accesses were interleaved in time. Processor which read data that was modified by committed transaction is forced to restart the transaction.
During the transaction all the stores are kept locally so it is possible that the same blocks with different values are kept on the same time in different processors caches. After the transaction commits and the stores are broadcasted, it is possible for other processors to see which blocks are modified. The processors should invalidate appropriate blocks if only addresses were broadcasted or modify them if the new data was also broadcasted with the addresses. Of course, if was detected that some data was used too early there is a need to restart the currently running transaction.
Detecting that the block that was read in current transaction is modified by another committed transaction and rolling back is analogous to RAW situation in a single processor that issues out of order instructions. In TCC the commit specifies the "right execution order". Therefore if load *'D'* executed in processor *P1*'s transaction and afterwards processor *P2*'s transaction containing store to the same address as *'D'* committed then it means that load *'D'* was executed before the preceding store *'D'* was. Thus the processor should roll back and reload the right *'D'*. It should discard all the instruction that *'D'* was involved in them and all instruction that were executed after branches that use of *'D'* caused.
Analogously to RAW there can be WAR and RAW dependencies all are treated by TCC. WAR avoided by the fact that data read in the 'earlier' transaction didn't know about the 'later' transaction's writes, even if the write was chronologically performed before that read on other processor local cache. Similarly WAW dependency is avoided since the 'later' transaction's write will overwrite the 'earlier' transaction's write.
Now let's take a glance in the most common problems of using locks: *Priority inversion, Convoying and Deadlock. Priority inversion* never occurs in TCC. There are no locks that may lock a lock needed by a higher priority process. Moreover the arbitration made in TCC can prefer a higher priority transaction over the lower ones in giving the commit approvals. *Convoying* eliminated. The fact that some process was de-scheduled cannot make other to wait. On the contrary, there will be less competition on the bus. Deadlocks are also avoided in absence of locks. The conclusion of these is that the use of TCC already pays itself by elimination of these problems that are very common and very difficult to debug.

4.2 TCC Hardware Support

Every processor in TCC system should have the following information in their local cache [5]:

- *Read Bits*: these bits indicate the cache line was speculatively read. When another processor's transaction committed the hardware will compare the addresses of cache lines with those bits set on in order to detect data that was read too early. If committed transaction modified data that has the same address as data in the cache line with that bit on has, the read will be discarded and the transaction will be restarted.
- *Modified Bits*: these bits indicate the cache line was speculatively written. When another processor's transaction committed the hardware will compare the addresses of cache lines with those bits set on. If committed transaction modified data that has the same address as data in the cache line with that bit on, all writes will be discarded and the transaction will be restarted.
- *Renamed Bits*: these bits indicate that the word or byte was read from the local generated data thus there is no need to set the *Read Bits* on that cache line. These bits are optionally and their purpose is to improve the performance by reducing the number of false transactions restarts.

In order to avoid false violation caused by writes to different places in the same cache line it can be useful to use multiple *Read/Modified Bits* per cache line.

During the transaction the lines with *Read/Modified Bits* should not be removed from the local cache. If capacity problems arise then the lines should be stored in the victim buffer. If that solution is not good enough, the processor must be stalled and request to commit the transaction should be made. If atomicity of the transaction requires that the remaining part of the transaction should be executed without being interleaved with another transaction, the bus has to be stalled till that transaction fully commits. That means that there may be a situation where the bus is not utilized at all, but there are many transactions which want to commit, thus stalling their processors. Of course, it is very bad situation and they should happen rarely when the transactions are very big. Another solution to that problem seen in *Unbounded Transactional Memory* (UTM) [8] see chapter 5.1.
The TCC hardware should provide a way to save the registers file in order to restore it if a violation occurs. That can be implemented in software by executing some code before the transaction begins and on a violation. Another option is to implement this in hardware by copying the register file to a special register file. As usual the software solution is cheaper since no additional hardware is required but the overhead in hardware implementation is much lower.
We need a mechanism to collect all the writes into a single packet. That can be done by using a dedicated buffer for speculative writes or by storing the list of all these writes.
Finally there should be a mechanism to store the phases of processors. The phases needed by the bus controller in order to arbitrate the commits, thus the arbiter will not permit the commitments of transaction when there are processors with elder phases.

In order to exploit the TCC hardware new programming techniques should be use. These techniques explained in the next chapter.

4.3 Programming with TCC

TCC hides from the programmer most of the difficulties of concurrent programming, however in order to achieve better performance results the programmer should be aware to some goals:

1. **Minimize Violations**: the programmer should try to avoid using multiple transactions that compete on the same data, thus causing one another to restart. The transactions preferred to be a relatively small, thus abort and restart in the case of data violation will cause a relatively small work waste.
2. **Minimize Transaction Overhead**: using too small transaction should also be avoided. Too many transactions cause a huge overhead created by starting and ending the transactions.
3. **Avoid Buffer Overflow**: all data changes are stored in a special buffer. Hence a long transaction that makes a lot of writes can exceed the hardware limit. That situation greatly reduces the performance (see chapters 4.2 and 5.1 for more details).

Parallelization of the code for TCC requires very small changes. TCC programming techniques enable to make a tradeoff between programming effort and performance. The code can be easily and quickly transformed to one that guaranties correctness, however in order to increase the performance by the means of the three goals above additional changes optionally can be done. The programmers can use the feedback from the hardware reports, detect the system bottleneck and make improvements in the code.

The programming in TCC can be divided to three steps:

1. **Division into Transactions**: the programmer should divide the code into transactions that can be executed in parallel on different processors. During that process the programmer is not required to assure that there are no data violations between concurrent transactions, since the TCC hardware will detect any data violation and assure the correctness.
2. **Order Specification**: If transactions will be committed in the same order as the instructions in the sequential program, the program will be executed correctly. That order is a default commitment order, however if a programmer knows that order is not essential he/she can fully or partially remove that constrain and thus improving the performance.
3. **Performance Tuning**: After the program was run in TCC system, the programmer can see a report about the locations of the violations. That knowledge can help the programmer to make additional performance improving changes.

Two methods are proposed for defining transactions and specifying commitment order: *Loop-Based and Fork-Based Parallelization*.

4.3.1 Loop-Based Parallelization

The principles of Loop-Based Parallelization will be presented through a code example. This is a procedure that counts the appearance of each letter in some string:

```
char str[1024]={0};
int letter_counter[26]={0};

//str gets the string (assume all letters in lower case)
str = load_string();

while (str[i] !=0)
{
        if (str[i]≥'a' && str[i]≤'z')
                letter_counter[str[i]-'a']++;
        i++;
}
```

The compiler will recognize a whole code as a one single transaction and there will be no parallelism. However the programmer is able to change the while loop to t_while loop, thus converting the loop to a parallel one. Every loop iteration will become an independent transaction that can be executed on another processor. The commitment order will be the original loop order, thus preserving the correctness.

Of cause, now we will have too many transactions thus reducing the performance. The solution will be to combine number of iteration to a larger iteration like in loop enrolling. That can be achieved by use of t_while_n(*predicate*; #), where # is the number of iteration that should be enrolled to a single one. Preferably that enrolling should be done automatically by a compiler. The value of # should either be decided by the compiler or even in the run time according to the input.
As we can see in the code there is no need to assure that some transaction will commit before another does. Therefore we can enable the transaction to be committed out of order. For that purpose we can use t_while_unordered and t_while_unordered_n. In unordered loops there is additional requirement that the programmer should take care of. The loop terminating predicate should be false not only for a specific iteration, but it should give false for all iteration after it. In our example the condition (str[i] !=0) is good only if we assume that the array was initialized thus all the chars after the first '\0' are themselves '\0'. Otherwise we can get a string like "memory\0j" and will count the letter 'j' even though it appears after the first '\0', while the original program won't count it.

Similarly to t_while loop the f_for loop can be used. Example and explanations of t_for use can be found in [6].

4.3.2 Fork-Based Parallelization and Commit Order Settings

Another TCC software structures enables us to run the transactions on a different threads and even specify commit order of several transactions.

The syntax of t_fork function reminds the C function fork.
void t_fork(void (*child_function_ptr)(void*), void *input_data,
int child_sequence_num, int parent_phase_increment, int child_phase_increment);

Where child function is of the form:
void child_function(void *input_data);

Before t_fork is called, the running (parent) transaction is forced to commit. That done because, if the running transaction will be aborted later the child tread will be logically canceled and then will be created again on transaction restart. Thus we will get into malicious situation, where two or even more identical threads in the system doing the same.

Every transaction has two parameters: *sequence* and *phase*. The TCC arbitration will allow only to the transaction with the lowest *phase* in a given *sequence* to commit.
The *sequence* of the child is set in child_sequence_num argument. If T_SAME_SEQUENCE constant was used, the parent and the child will have the same *sequence*, otherwise the child will have a different sequence specified in child_sequence_num.
The parent transaction *phase* will be parent_phase_increment over the old parent transaction *phase*. For example, parent_phase_increment == 0 will cause the new parent transaction be of the same *phase* as the old one.
The child *phase* will be child_phase_increment over the parent transaction if T_SAME_SEQUENCE was used, otherwise it will be child_phase_increment over the youngest transaction with the same *sequence*.

We can explicitly force a transaction to commit by using:
void t_commit (int phase_increment) ;
void t_wait_for_sequence (int phase_increment, int wait_for_sequence_num);

t_commit function will commit the current transaction and start a new one with the *phase* incremented by phase_increment.

t_wait_for_sequence routine does the same operation, but before it creates a new transaction it waits until all transactions with wait_for_sequence_num will commit.

All the structures and routines we have mentioned provide the programmer with the tools to write a good parallel running program, but where and how exactly to use them is not a simple question. To answer it we can use the TCC feedback reports. Running the program and getting a feedback, will help a programmer to locate the bottleneck and to improve the performance of the program.

5 Advanced Transactional Memory Models

One serious disadvantage of proposed TCC is the assumption that transactions will be relatively small. This assumption is only partially true. While vast majority of transactions are small there are a few very large transactions that should be treated as well. Solution to that problem can be found in the *advanced TM models*.

TM simplifies parallel programming by guaranteeing that transactions appear to execute atomically and in isolation. Implementing these properties includes providing data version management for the simultaneous storage of both new (visible if the transaction commits) and old (retained if the transaction aborts) values.
Most (hardware) TM systems leave old values "in place" (the target memory address) and buffer new values elsewhere until commit. This makes aborts fast, but penalizes (the much more frequent) commits.

TM systems must provide transaction *atomicity* (all or nothing) and *isolation* (the partially-complete state of a transaction is hidden from other transactions).
Providing these properties requires data *version management* and *conflict detection*, whose implementations distinguish alternative TM proposals.

Version management handles the simultaneous storage of both *new* data (to be visible if the transaction *commits*) and *old* data (retained if the transaction *aborts*). At most one of these values can be stored "in place" (the target memory address), while the other value must be stored "on the side" (e.g. in speculative hardware).
On a store, a TM system can use *eager version management* and put the new value in place, or use *lazy version management* to (temporarily) leave the old value in place.

Conflict detection signals an overlap between the *write set* (data written) of one transaction and the write set or *read set* (data read) of other concurrent transactions. Conflict detection is called *eager (Forward validation) if* it detects offending loads or stores immediately and *lazy (Backward validation)* if it defers detection until later (e.g. when transactions commit).

We will present these advanced TM models with respect to *version management* and *conflict detection*.

5.1 Unbounded transactional memory

The main goal of the *Unbounded Transactional Memory (UTM)* [7] is to enable execution of transaction of unlimited size and duration.
UTM adds two new instructions to the ISA: XBEGIN pc and XEND.
On XBEGIN, UTM saves a snapshot of the register-renaming table and keep it till the XEND when the transaction commits. If data conflict is detected and transaction is forced to restart, the register file will be restored from the snapshot. UTM ensures that these register won't be reused till the transaction's commit. The pc of XBEGIN specifies the relative location of the abort handler function that will be called on transaction abort.

UTM keep a track of all the transactions in the system in a dedicated data structure a *transaction log*. All *transaction log*s in the system are combined in a single data structure ***xstate*** which is kept in the main memory. Every log is built of a *commit record* and a list of *log entries*. Commit record keeps the transaction's state: Pending, Committed and Aborted. Every *log entry* has the pointer to the block that was read / written by the transaction, the old value for backup and the *readers list* (other transactions that have read the block).
Finally ***xstate*** has a *RW bit* and the *log pointer* for each memory block. If some block was written by some transaction the value of *RW bit* will be W, otherwise R. *Log pointer* points out to the newest transaction log for that memory block. Using these two can help in detection of data conflicts.
When a conflict is detected the system uses RW bit, the log pointer and the readers list to get all the transactions that have used data from some block. These transactions can be restarted and the old value from *transaction log* will be restored.

UTM has the following system characteristics:

- UTM uses *eager Version management*, thus writes the new values to the memory and keeps the old ones for backup.
- If the space allocated for a given transaction log is not large enough the system will abort the transaction allocate a larger space and restart the transaction again. In order to free the memory allocated for already committed transactions the system will constantly iterate the log entries and clean up the *transaction logs* as well as the *log pointers*.
- Like as in TLR, UTM uses timestamps to ensure a progress.
- Cache will be used to accelerate the process. Since the transaction log is not important for committed transactions the log can be stored on the cache. If transaction commits without problems it will be deleted from the cache, without being written back to the main memory, thus accelerating the process.
- UTM should enable use of transactions for indefinite time, with possible migration between processors and data set bigger than the physical memory like it does in the virtual memory. Therefore UTM should use the paging method to page ***xstate*** parts to the disk.
- Of course I\O during the transaction will cause an error since transaction cannot be restarted.
- Nested transactions are supported by keeping track of the nesting depth.

5.1.1 LTM

Implementation of UTM is very difficult, thus the *Large Transactional Memory (LTM)* [7] was proposed. LTM uses the same concepts and structures as UTM with the following restrictions:

- Memory size is limited to the physical memory size and not to the virtual memory size.
- The duration of transactions should be less the time slice and transactions cannot migrate between processors.
- Transactions are not linked to data structure, but to a specific cache.
- LTM uses *lazy Version management***,** thus the memory contains only the original values.
- Conflicts are detected using the cache coherence protocol.

5.2 Virtual transactional memory

Virtual Transactional Memory (VTM) [8], a combined hardware/software system architecture that allows the programmer to obtain the benefits of transactional memory without having to provide explicit mechanisms to deal with those rare instances in which transactions encounter resource or scheduling limitations.
The underlying VTM mechanism transparently hides resource exhaustion both in space (cache overflows) and time (scheduling and clock interrupts). When a transaction overflows its buffers, VTM remaps evicted entries to new locations in virtual memory. When a transaction exhausts its scheduling quantum (or is interrupted), VTM saves its state in virtual memory so that the transaction can be resumed later.
VTM virtualizes transactional memory in much the same way that virtual memory (VM) virtualizes physical memory. Programmers write applications without concern for underlying hardware limitations. Even so, the analogy between VTM and VM is just an analogy.
VTM achieves virtualization by decoupling transactional state from the underlying hardware on which the transaction is executing and allowing that state to move seamlessly (and without involving the programmer) into the transaction's virtual address space. This virtual memory based overflow space allows transactions to bypass hardware resource limits when necessary. Since the virtual address space is independent of the underlying hardware, a transaction can be swapped out, or migrate without losing transactional properties.
VTM doesn't slow down the common case where hardware resources are sufficient. Virtualization provides essential functionality, and has an insignificant effect on performance.
VTM detects inter thread synchronization conflicts, and commits or aborts multiple updates to disjoint virtual memory locations in an atomic, non-blocking way.
To this end, VTM provides two operational modes:

- A hardware-only fast mode provides transactional execution for common case transactions that do not exceed hardware resources and are not interrupted. This mode is based on mechanisms used in UTM (see 5.1 in this paper), SLE (see 2 in this paper), TLR (see 3 in this paper)-this mode's performance effectively determines the performance of the overall scheme.
- A second mode, implemented by a combination of programmer-transparent software structures and hardware machinery, supports transactions that encounter buffer overflow, page faults, context switches, or thread migration.

5.3 Log-based transactional memory

LogTM implements *eager version* management by creating a per-thread *transaction log* in cacheable virtual memory, which holds the virtual addresses and old values of all memory blocks modified during a transaction, like UTM does. LogTM detects in-cache conflicts using a directory protocol (MOESI) and read/write bits on cache blocks. LogTM extends the directory protocol to perform conflict detection even after replacing transactional data from the cache. In LogTM, a processor commits a transaction by discarding the log (resetting a log pointer) and flash clearing the read/write bits. No other work is needed, because new values are already in place and. On abort, LogTM must walk the log to restore values. It was found that aborts sufficiently rare thus, using a trap handler to perform them in software. For ease of implementation, the processor whose coherence request causes a conflict always resolves the conflict by waiting (to reduce aborts) or aborting (if deadlock is possible). Currently, LogTM does not permit thread movement or paging within transactions, as do UTM and VTM.

We would like to point out several key points regarding LogTM:

- LogTM uses eager version management to store new values “in place,” making commits faster than aborts. On commit, no data moves (even when transactions overflow the cache).
- LogTM efficiently allows cache evictions of transactional data by extending a MOESI directory protocol to enable:
    - Fast conflict detection on evicted blocks.
    - Fast commit by lazily resetting the directory state. LogTM does *not* require log or hash table walks to evict a cache block, detect a conflict, or commit a transaction, but works best if evictions of transactional data are uncommon.
- LogTM handles aborts via a "log walk" by software with little performance penalty.
- LogTM implementations could use other coherence protocols (e.g. snooping), extended with appropriate filters to limit false conflicts.
- LogTM uses hardware to save user-visible register state and restore it on transaction abort. This could also be done by the compiler or run-time support.
- LogTM implementation uses timestamps to prioritize transactions and resolve conflicts. This simple, but rigid policy may result in *convoys* [2](e.g., if a transaction gets pre-empted) or *priority inversion* (e.g., if a logically earlier transaction holds a block needed by a higher priority, but logically later transaction).
- Because LogTM stores the old values in the user program’s address space, these mechanisms appear possible.
- LogTM’s log structure also lends itself to a straight-forward extension to nested transactions.

6 Conclusions

Transactional Memory appears to be a powerful architecture that increases the performance and eases the programming on shared-memory multiprocessors systems. We saw that there is a huge progress in that area beginning from the early Herlihy proposal [2], through various TM models, till the latest proposals of UTM, VTM and LogTM. We think that it is desirable to develop a Transactional Memory system and believe that it will be the next evolution in the multiprocessors architecture. Finally we summarize all TM proposals in a single table (table 6 on the following page).

Acknowledgement

We specially thank Dr. Shlomo Weiss for his assistance and guidance.

TM System	version management	Conflict detection	Demands from the Programmer	Relation to Coherence and consistency protocols	Data Conflict arbitration	Actions due to resources depletion
Ideal	**Eager (puts new values "in place," making commits faster than aborts. This makes sense when commits are much more common than aborts, which we generally find)**	**Eager (finds conflicts early, reducing wasted work by conflicting transactions. This makes sense, since standard coherence makes implementing eager conflict detection efficient**	**No Locks are used (Ideal from programmer's point of view).**		**Exist**	**Unlimited resources.**
Early TM [2]	Lazy	Lazy	Special instructions are used (e.g. validate, commit). No Locks are used.	Uses existent and extends them.	Exist	It is required that transactions will not exceed resources limits.
SLE [3]	Lazy	Eager	Transparent.	Doesn't Implement, Uses existent.	Doesn't Exist (Lock acquisition required and restarting the transaction).	Use locks.
TLR [5]	Lazy	Eager	Transparent (≡ The programmer uses Locks to define critical sections, as he usually does).	Doesn't Implement, Uses existent.	Exist	Use locks.
TCC [6 7]	Lazy	Lazy	Special software structures are used: transactional loops and forks. No Locks are used.	Replaces the existent.	Exist	Commits the changes and stalls the bus until The transaction ends.
VTM [9]	Lazy	Eager	No Locks are used.	Replaces the existent.	Exist	VTM remaps evicted entries to new locations in virtual memory.
UTM[8]	Eager	Eager	Special instructions are added to ISA (e.g XBEGIN, XEND) No Locks are used.	Replaces the existent.	Exist	It keeps track of all transactions and use paging when Physical memory is full.
LTM [8]	Lazy	Eager	No Locks are used.	Uses the existent.	Exist	It keeps track of all transactions, assuming that it won't exceed physical memory space.
LogTM [10]	Eager	Eager	No Locks are used.	Uses the existent.	Exist	It keeps a log of all transactions and use paging when Physical memory is full.

Table 6: Comparison of all surveyed TM Models.

References

[1] Jiandong Huang, John A. Stankovic, Krithi Ramamritham and Don Towsley. Experimental Evaluation of real-Time Optimistic Concurrency Control Schemes. *February, 1991*

[2] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures

[3] R. Rajwar and J. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34), December 2001.*

[4] Ravi Rajwar and James R. Goodman. Transactional Lock- Free Execution of Lock-Based Programs. In *Proc. of the Tenth Intl. Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2002.*

[5] L. Hammond, V. Wong, et al. Transactional memory coherence and consistency. *In ISCA 31, pp. 102.113, June 2004.*

[6] L. Hammond, V. Wong, et al. Programming with Transactional coherence and consistency (TCC).

[7] C. Scott Ananian, Krste Asanovic', Bradley C. Kuszmaul, Charles E. Leiserson and Sean Lie. Unbounded Transactional Memory

[8] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Annual Intl. Symp. On Computer Architecture*, Jun. 2005.

[9] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill & David A. Wood. LogTM: Log-based Transactional Memory.
Appears in the proceedings of the12th Annual International Symposium on High Performance Computer Architecture (HPCA-12) Austin, TX February 11-15, 2006