Transactional Memory

Anshuman Tripathi* 07CS3024
Gautam Kumar 07CS1021
Manaal Faruqui 07CS3011
Parin Chheda 07CS3023

Abstract

In this paper we explore a hardware-based technique for lock-free data structures. Such a solution is as efficient as conventional lock-based techniques in the absence of priority inversions, convoys and deadlocks but easily surpasses them in the presence of such problems. We first describe the overview of the architecture and implementation of Transactional Memory using Goodman's bus-based protocol. We follow by presenting in detail how Transactional Memory preserves coherence and consistency and illustrate its performance improvements. We also describe the Transactional memory design that can be used to implement transactions whose memory footprint becomes very large even with footprints nearly as large as virtual memory. We conclude by describing how Transactional Memory can be exploited for writing parallel programs by presenting some programming constructs based on underlying TM harware and how some optimizations can be done by exploiting parallelism

1 Architectural Support for Lock-Free Data Structures

1.1 Introduction

A shared data structure is said to be *lock-free* if processes operating on it do not require to hold an exclusive lock to be able to operate on them, or in other words, they do not require mutual exclusion. Conventional locking mechanisms pose the problems of priority inversion, convoying and deadlocks in highly concurrent systems and perform quite inefficiently in their presence. Software techniques to implement *lock-free* structures do not perform as well as locking-based techniques in the absence of these problems[1][9]. We explore *transactional memory*, a hardware based solution to implement such structures, that is as efficient lock-based techniques in the absence of priority inversions, convoys and deadlocks, and easily surpass them in the presence of these problems. The essential idea is to extend the cache-coherence protocols to be able to define customized read-modify-write operations on words of memory. In the subsequent sections we explain the notion of transactional memory, the instructions required to access them and the hardware support required for its implementation using Goodman's snoopy protocol for a bus-based architecture [6]. It is interesting to note that an implementation using Chaiken's directory protocol[4] is also possible but has not been covered in this artcile.

All the notation and architecural design has been taken from [10]

*Names of authors are in lexicographic order

1.2 Transactional Memory

Shared memory is defined as a set of words of memory that may be simultaneously accessed by multiple programs to facilitate communication amongs them. A critical section is a part of code that reads/writes on shared memory and cannot be accessed concurrently by more than 1 process. A traditional method of providing mutual exclusion is by making a process hold an exclusive lock on the data that exists in it's critical section and releasing the lock when the critical section ends. *Lock-free* structures is a new paradigm which mantains consistency but does not require such locks, i.e. a process need not wait to acquire a lock to be able to execute. The basic idea stems from the notion of *transactions* in database management systems. At a deeper level, a *transaction* is a finite sequence of machine instructions that is executed by a single process satisfying the properties of *Serializability*, the actual execution sequence should result in a state than can be attained by one of the possible serial executions of the transactions, and *Atomicity*, which states that a transaction can either complete in full or is aborted.

1.2.1 Instructions

Transactional memory provides the following special instructions to access memory instead of the simple load and store.

LT	*Load-transactional*	loads the value of a shared memory location into a register
LTX	*Load-transactional-exclusive*	same as LT but means that the location is likely to be updated
ST	*Store-transactional*	stores the value from the register to the shared memory location tentatively

A transactions *read set* is defined as the set of locations loaded using LT, *write set* as the set of locations loaded using LTX or stored using ST and *data set* as the union of the two.

The following instructions are also provided to change the transaction state.

VALIDATE	returns *true* if the present transaction has not been aborted and *false* otherwise
COMMIT	makes the transaction's changes permanent, succeeds only if no updates to it's data set have been made by other transactions and no other transaction has accessed it's write set.
ABORT	all changes to the transaction's write set are discarded

1.2.2 Transaction Style

Using the above primitives from transactional memory access and changing transaction state, a short transaction can be implemented in the following way.

1. *Read using LT or LTX from a set of locations*
2. *VALIDATE to check the consistency of the values. If it fails, go to step 1*
3. *ST to store from register to the memory locations*
4. *COMMIT to try to make the transaction's changes permanent. If it fails, go to step 1*

1.3 Architecure and Implementation

An operation can either be *transactional* or *non-transactional*. The design presented in [10] ensures that non-transactional operations use the regular caches and their respective protocols. Custom hardware required is only limited to the primary caches and the transactional instructions mentioned in section 1.2.1. The basic idea is to to leverage access-rights which are a part of standard multiprocessor cache coherence protocols. A memory location, at any given time, is in 1 of the following 3 states.

1. not resident in any cache, present in memory only.
2. accessible directly by one or more processors (*non-exclusive*) and permitting only reads.
3. accessible by exactly 1 processor (*exclusive*) and permitting reads and writes.

Here's how these rights are put to use to identify a transaction conflict. Consider that a processor M wants to read some memory locations. It needs to obtain non-exclusive rights to this location. When another processor N wants to write to one of these locations, it must acquire exclusive access and needs to revoke P's access and a conflict can be determined.

1.3.1 Implementation using Goodman's "snoopy" protocol

We describe here an architecture that uses Goodman's "snoopy" protocol for a shared bus [6] to implement transactional memory. In this architecture, each processor maintains a special primary cache, apart from a *regular* cache to be used for non-transactional operations, known as the *transactional* cache. These caches are exclusive, all the non-transactional data goes in the regular cache and all the transactional data goes in the transactional cache. This transactional cache is small and fully-associative with parallel logic to execute COMMIT and ABORT in a single cycle.

1.3.2 Augmenting Cache Line States

As Goodman protocol says, each cache line can be in one of the following states.

Name	Access	IsShared	IsModified
INVALID	–	–	–
VALID	R	True	False
DIRTY	R, W	False	True
RESERVED	R, W	False	False

The transactional cache line has an extra state which can be one amongst the following states.

Name	Explanation
EMPTY	does not contain any data
NORMAL	contains committed data
TC_COMMIT	to be discarded on commit
TC_ABORT	to be discarded on abort

Whenever a memory location is read into the cache, two entries are created one marked TC_COMMIT and the other marked TC_ABORT. All the modifications are only made to the TC_ABORT entry. On committing the transaction, the entries marked TC_COMMIT are marked

EMPTY and the entries marked TC_ABORT are marked NORMAL. When a transaction is aborted, the entries marked TC_ABORT are marked EMPTY, and TC_COMMIT entries are marked NORMAL.

With the present scheme, whenever a new entry has to be inserted, the transactional cache first searches for an EMPTY entry, then for a NORMAL entry, and finally for an TC_COMMIT entry. (Note that the TC_COMMIT entry needs to be written back to the memory before replacement.) Updates made by ST are made to the TC_ABORT entry, and thus the old version is maintained in it's complementary TC_COMMIT entry.

1.3.3 Augmenting Bus Cycles

As per the Goodman's original protocol, there are three kinds of bus cycles. The READ cycle acquires shared ownership of a chache line, whereas the RFO (read-for-ownership) acquires exclusive rights for the same. The WRITE cycle updates the main memory on write-through and is also used when dirty lines are replaced. Memory snooping is done on the bus and if a dirty line is read by another processor, the WRITE cycle is executed again and data is written back to the memory. To integrate the transactional cache in the architecture, three new bus cycles are added to the protocol. To request entries into the transactional cache, the T_READ and T_RFO need to be executed which are simple counterparts of READ and RFO cycles respectively. A BUSY cycle is also added that is used to *refuse* entry into the transactional cache. This is done to prevent transactions from aborting each other too much.

1.3.4 Integration with the Processor

There are two flags that the processor needs to maintain to execute transactions suitably.

TACTIVE	indicates if a transaction is presently in progress, implicityly set on execution of its first transactional operation
TSTATUS	if TACTIVE is true, it indicates if the transaction is active or has been aborted

Let us now consider the steps required to execute the various transactional operations issued by an active transaction (TACTIVE is *true* and TSTATUS is *true*).

LT :

1. *Probe Transactional Cache for a line marked TC_ABORT*
2. *If not found, but a NORMAL entry is found, change the NORMAL entry to TC_ABORT entry and allocate another entry marked TC_COMMIT (to remember the old version)*
3. *If neither a TC_ABORT nor a NORMAL entry is found, a T_READ cycle is executed whose successful completion creates two entries in the transactional cache, one marked TC_COMMIT and the other TC_ABORT*
4. *On receiving a BUSY response, abort the transaction setting TSTATUS to* false*, mark all TC_ABORT entries as EMPTY and set all TC_COMMIT entries to NORMAL*
5. *Update Goodman's state in accordance with LOAD operation*

LTX :

1. *Execute the same steps as in LT, however on a miss, issue a T_RFO cycle and mark the cache line RESERVED on its successful completion*

2. *Update Goodman's state in accordance with LOAD operation*

ST :

1. *Proceed like LTX but update TC_ABORT entry's data*

2. *Update Goodman's state in accordance with STORE operation*

VALIDATE :

1. *Get the TSTATUS flag*

2. *If TSTATUS is* false*, set TACTIVE to* false *and TSTATUS to* true

ABORT :

1. *Discard cache entries marked TC_ABORT by marking them EMPTY and mark TC_COMMIT entries NORMAL*

2. *Set TSATUS to* true *and TACTIVE to* false

COMMIT :

1. *Return TSTATUS*

2. *Set TSTATUS to* true *and TACTIVE to* false

3. *Discard all TC_COMMIT entries and change all TC_ABORT entries to normal*

.

1.3.5 Integration with the Processor

Both the regular and the transactional cache snoop on the bus. The regular and the transactional cache behave differently as explained below.

Behaviour of Regular Cache :

1. *On a READ or T_READ, return the value if the state is VALID. Else If the state is RESERVED or DIRTY, return the value and reset the state to VALID*

2. *On a RFO or T_RFO, return the data and invalidate the line*

Behaviour of Transactional Cache :

1. *If TSATUS if* false*, or if the operation is non-transactional (simple READ or RFO), act like a regular cache, but ignore entries with tag other than NORMAL*

2. *On T_READ, return the value if the state is valid, and return BUSY for all other transactional operations*

Either of the cache can issue a write request when a replacement of the cache line is needed. A sample counting benchmark demonstrating the use of Transactional memory is given below.

```
shared int counter;

void process (int work) {
  int success = 0, backoff = BACKOFF_MIN;
  unsigned wait;
  while (success < work) {
    ST (&counter, LTX (&counter) + 1) ;
    if (COMMIT()) {
      success++;
      backof f = BACKOFF_MIN;
    }
    else {
      wait = randomo \% (01 << backoff) ;
      while (wait -- ) ;
      if (backof f < BACKOFF_MAX )
        backoff ++;
    }
  }
}
```

2 Transactional memory Coherence and Consistency

2.1 Introduction

Transactional memory Coherence and Consistency[1] is a model in which instead of single instructions the basic unit of parallel work, communication, memory coherence and memory reference consistency are **atomic transactions**. A transaction is a sequence of instructions that is guarunteed to execute and complete only as an atomic unit. This property of atomicity prevents updates occurring only partially which can often cause more problems than no updates occurring at all. TCC greatly simplifies parallel software by eliminating the need for synchronization using conventional locks and semaphores.

Most of the parallel processing systems today use one of two basic models to coordinate synchronization and communication: **message passing** or **shared memnory**. However both of these models have disadvantages - message passing makes software desing difficult, while shared memory requires complex hardware to make progamming slightly simpler. TCC model on the other hand presents a shared memory modle to programmers and reduces the need for extensive hardware support as would be illustrated later in the paper.

The described work and used figures have been taken from [8].

2.2 Overview of the model

Each transaction produces a block of writes called the write state which are committed to shared memory only as an atomic unit, after the transaction completes execution. When the transaction is complete, the hardware sends out messages system-wide to get permission to commit the "writes" associated with it. After getting this permission the processor simply broadcasts all writes for the entire transaction out as a large packet to the rest of the system. Snooping by other processors on these packets maintiains coherence in the system, and allows them to detect when

[1] Denoted by TCC henceforth

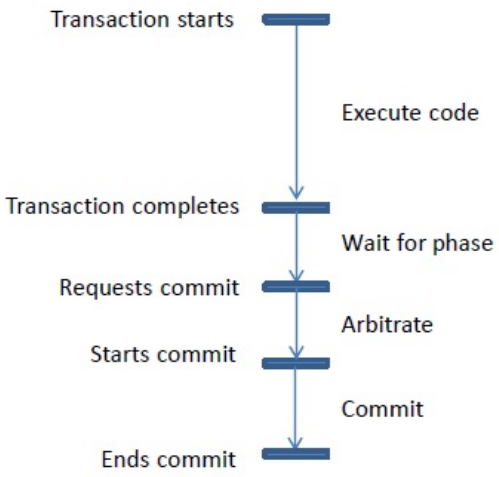

Figure 1: Sample transaction timeline

they have used a data which has subsequently been changed and must rollback – a *dependence violation*. Figure 1 illustrates a sample transaction execution process.

Such kind of combining all writes from a transaction together minimizes the latency sensitivity as fewer interprocessor messages are required. Also, since we only need to control the sequencing between entire transactions, instead of individual instructions, we leverage the commit operation to provide innate synchronization and a simplified consistency protocol. The conventional consistency and coherence models are changed as follows:

- Consistence: TCC imposes a sequential ordering between transaction commits instead of imposing some sort of ordering rules between individual memory reference instructions. This reduces the number of latency-sensitive arbitration and synchronization events. A processor that reads a data that is subsequently updated by other processor's commit, before it cancommit itself, is forced to violate and rollback. Interleaving between processors' memory references is only allowed at transaction boundaries.

- Coherence: Store operations are buffered and kept within the processor while the transaction is still executing in order to maintain atomicity. At the end of each transaction the broadcast notifies all other processors about what state has changed during the completing transaction. If they have read any data modified by the committing transaction during their currently executing transaction, they need to restart and reload the correct data: preventing *data dependencies*. Also, *data antidependencies* are handled simply by the fact that later processors will eventually get their own turn to flush out data to memory.

2.3 Programming with TCC model

Programmers only need to satisfy one requirement for successful transactional execution: insertion of transaction boundaries into their parallel code occasionally keeping in mind that the transactional breaks should never be inserted in between a load and any subsequent store of a shared value i.e, during a lock's critical region. Parallel programming with TCC can be looked upon as a three step process:

1. Dividing into transactions: This step is similar to conventional parallelization, which requires programmers to coarsely divide the program into blocks of code that can run

concurrently on different processors. But the advantage to note here is this, that the programmer need not guaruntee that the parallel regions are independent as these violations will be caught and corrected during the runtime.

2. Specifying order: If required the programmer can specify an ordering between transactions to maintian a program order that must be enforced. This can be achieved by assigning hardware-managed *phase numbers* to each transaction. At any instant of time only the transactions with the least phase numbers are allowed to commit and transactions from other phases are forced to wait. This forces the commits to be in order and is also deadlock-free as at least one processor is always running the lowest phase-numbered transactions.

3. Performance Tuning: TCC can be made to give out information regarding where violations occurred in the program which can direct the programmer to perform further optimizations like choosing the transactions to maximize parallelism and minimize inter-transaction data dependencies. Large transactions are preferable but small transactions should be used when violations are frequent to minimize them, or when the memory-buffer is often overflowing.

2.4 Speculative buffering of memory references and commit control

Individual processor nodes within a TCC system have some features to support speculative buffering of memory references and commit control. The required features are as follows:

- Read bit: When a cache line has been read spculatively the read bit is set. These bits are then snooped by other processors to know whether the line has been specultively read and thus correct any occurred violation.

- Modified bit: This bit is set for a line when any part of the line has been written speculatively. These are used to invalidate all speculatively written lines at once when a violation is detected.

- Renamed bits: In a cache line these bits are associated with individual words or bytes. these bits can only be set if the entire word in written by a store and not only some part of it. When set, any further reads from these words do not need to set read bits, because they are guarunteed to only be reading locally generated data that cannot cause violations.

Its important to note that cache lines with set read or modified bits should not be flushed from the local cache hierarchy in mid-transaction. If at all this occurs, the discarded cache lines must be maintained in a victim buffer or the processor should be stalled temporarily. The processor must also have a way to checkpoint its register state at each commit point in order to provide rollback capabilities. This is done either in hardware by flash-copying the register state to a shadow register file at the beginning of each transaction, or in software, by executing a small handler to flush out the live register state at the start of each transaction.

Finally the code must have a mechanism for collecting all of its modified cache lines together into a commit packet. This is implemented either as a write buffer completely separate from the caches or as an address buffer that maintains a list of line tags that contain data which needs to be committed.

2.5 Further improvements in TCC

2.5.1 Double buffering

Double buffering implements extra write buffers and additional sets of read and modified bits in every cache line, so that successive transaction can alternate between sets of bits and buffers. As expected this mechanism allows a processor to continue working on the next transaction even while the previous one is waiting to commit or is committing. The expense in implementing it is in replicating the additonal sets of speculative cache control bits and write buffers.

2.5.2 Hardware-controlled transactions

Till now the TCC we have learned uses explicitly marked transaction boundaries by the programmers, however even hardware could divide program execution into transactions automatically as the speculative buffer overflows. This would lead to optimal transaction size by not letting transaction get so large that buffering becomes a problem, while letting them be large enough to minimize committing overhead cost. Thus, the hardware can automatically insert transaction commits whenever the speculative buffer is filled, thereby breaking up large transactions into smaller perfectly sized ones.

2.5.3 Localization of memory references

So far we have assumed that all loads and sotres must be speculatively buffered and broadcasted system-wide. However, compilers can actually give programmers hints to reduce the need for buffering by letting the programmer know that some loads and stores are “local” and thus they need not be broadcasted.

2.5.4 Input-Output handling

In TCC a transaction cannot violate and rollback after an input has been obtained. The input is read only after commitpermission is obtained and when the transaction can never roll back. Outputs that require to write in a specific order can force the writes to propagate out from the processor immediately as the stores are made. Also, outputs that can accept potentially reordered writes may simply be updated at commit time, along with normal memory writes, thereby improving performance.

2.6 Performance Evaluation

2.6.1 Parallelism

The speedups achieved with several benchmarks are close to the optimal linear case as shown in Figure 2 although there are several reasons why speedups are limited, one of them being the presence of sequential code in applications which hinders speedup as per Amdahl’s law. Also the occurence of occasional deadlocks in many applications caused by inter-transaction dependencies remaining in the programs.

2.6.2 Buffering requirements

It is important that the amount of state read and/or written by an average transaction be small enough to be buffered on-chip avoiding overflow. Except a few all of our benchmarks worked

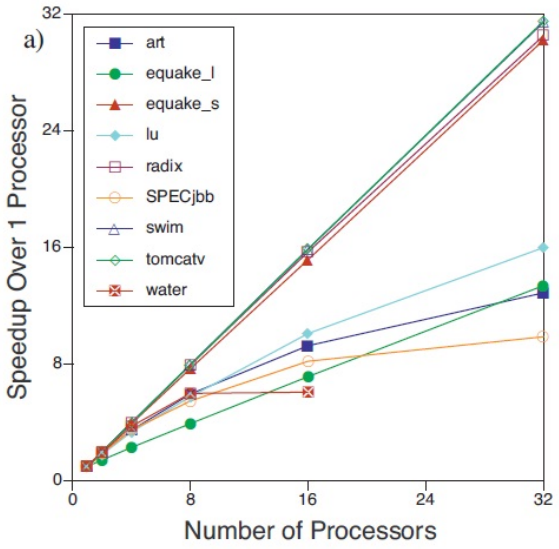

Figure 2: Speedups for varying number of processors on benchmarks

fine within about 6-12 KB of read state and 4-8 KB of write state which is well within the size of the smallest cache sizes used today.

2.6.3 Bus Bandwidth

For all the benchmarking applications, the number of addresses per cycle is well below 1, so a single snoop port on every processor node should be sufficient for designs of upto 32 processors, and can probably scale up to about 128 simple processors. This indicates that small TCC systems using an invalidate protocol would ususally produce less than 0.5 bytes/cycle with 32-bit addresses which is desirable.

3 Unbounded Transactional Memory

Shared memory data structures is one of the most versatile methods to share data across processes/threads in a distributed application. With the notion of shared memory the problem of mutual exclusion immediately pops-up. The problem of mutual exclusion is to enable *atomic* access to the shared resource while avoiding process starvation and providing fair resource sharing. Borrowing from theory of transactions in Database Systems, any transaction management system should satisfy (ACID) properties viz. *A*tomicity, *C*oncurency, *I*solation and *Durability*. *Lock* based mutual exclusion is one of the most commonly used methods in practice but it has several drawbacks:-

- *Progress*: There is a possibility of deadlock with lock based protocols. Thus in general there is to time bound for process to get a resource i.e. progress is not guaranteed.
- *Concurrency*: To prevent deadlock the locks shall be acquired in some linear order. This may lead to acquiring unnecessary lock on some resources that are not even used which will result in reduced concurrency.
- *Overheads*: For locking based protocols there is always an added overhead of lock acquisition, release and management even when no other process might be using the resource.

Similarly there is another approach of *non-blocking synchronization* [7] but it is unsuitable form programming and implementation point of view. However it has been verified in literature that *non-blocking synchronization* performs better than *Lock-based synchronization*.

designs have been proposed in past for solving the problem of mutual exclusion and synchronization at the level of memory reads(*load* instruction) and writes(*(*store) instruction).TMs may be Hardware (implemented at the hardware level) or Software (implemented at kernel level). The main motivation being to use the concepts of Database Systems at the level of memory by considering a critical section execution as a transaction. Most of *TM* designs are based on the underlying cache-coherence protocol for guarantying *isolation* and processor *re-order buffer* for handling *roll-back*. These designs thus put an upper bound on the size of transaction in the terms of number of instructions (bounded by the window size of processor) and also require presence of cache memory.

Here we describe an approach called that extends the line of TMs by removing the constraints of size on the transactions. The UTM model described herein supports memory transactions of unbounded size (bounded only by the total size of virtual memory). Furthermore the design does n't depend on presence of cache memory for correctness, but presence of cache can be advantageous for the speed of implementation. As we shall see UTM requires a lot of hardware support and modification in the architecture processor and memory, thus another design called has been explained that relaxes the requirement of unbounded transaction to transaction bounded only by the size of main memory (There are some more relaxations w.r.t the UTM model as we shall see in the following sections).

3.1 Unbounded Transactional Memory (UTM) design

As described earlier UTM does n't depend on the presence of cache, thus for the concurrency management UTM uses in-memory *Transaction logs*. As described in the following sections, UTM can support unbounded transactions limited only by the size of total virtual memory available.

3.1.1 Model Specification

UTM model introduces two new instructions to the processor's Instruction Set Architecture (ISA). These new Instructions are:-

`XBEGIN pc` : Marks the start of a memory transaction. **`pc`** is the pc-relative address where a branch is made in case the transaction fails (aborts). There `XBEGIN` instructions can also be nested i.e. there can be multiple sub-transactions inside any transaction in the UTM model.

`XEND` : This instruction marks the end of an instruction. It means that all the effects of the transaction have been successfully committed.

Speaking of semantics of the `XBEGIN` instruction, it is similar to a conditional branch instruction in which if the transaction aborts (the condition) a jump is made to `pc` (the jump address). Initially the model treats `XBEGIN` as an untaken branch (assumes transaction will be successfully committed), however in case of an abort the transaction is rolled back and after rolling back the control passes on to instruction at `pc`. As mentioned earlier the transactions can be nested in nature, in which case the atomicity domain of any transaction is same as the atomic domain of the transactions that encapsulates all the transactions (the outer most transaction).

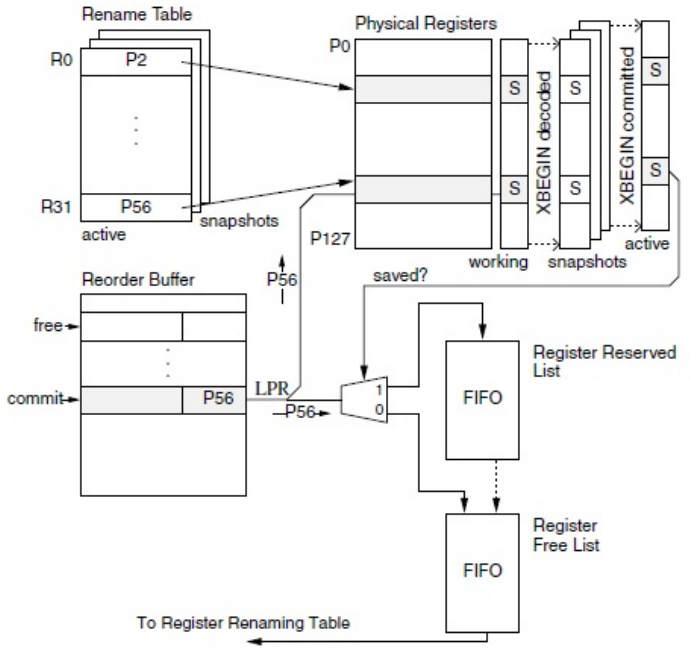

Figure 3: Modifications Introduced by UTM model in the processor architecture

Further modifications required in the architecture of memory and processor are (figure 3 shows the modifications required in a schematic diagram):-

1. "S" bit in the register rename file for each physical register. This is used in collecting the snapshot before graduation of `XBEGIN` instruction
2. RW bits in each block of the main memory. These bits signify the participation of a memory block in a transaction.
3. *log pointer* for each block of the main memory. This pointer points to the *log-record* (part of *xstate* data structure design) that corresponds to the last access of the memory location by a transaction.

Before describing the complete outline of the working of UTM model, here is the description of the roll-back mechanism used. The memory and the processor register file can be cumulatively visualized as a system state and any transaction can be seen as a transition from one state to another. Thus when we talk of roll-back, we essentially mean to bring the participating parts of main memory and processor register file to the same state, they were before the graduation of `XBEGIN` instruction corresponding to the transaction that aborts. Thus the state can be divided into two parts:- processor state and memory state. In the following section we see how the UTM design models both of these states so as to support rollback and concurrency control.

3.1.2 Modeling Processor State

The state of a processor is essentially constituted by values of all the physical registers, thus a trivial way to collect the processor state is to copy the value of all the physical registers in a transaction snapshot and in case of failure, just restore the register values form the snapshot. However in this approach there will be several registers which are independent of the transaction but still their values will be stored (which is inefficient). To overcome this inefficacy the UTM

design assumes that processor has a unified register file and a renaming table (mapping from architectural registers to physical registers). UTM specification introduces a "S"(saved) bit associated with each physical address in the register-rename file to distinguish the registers which are independent of the transaction to those that shall be preserved in case of transaction failure. When a `XBEGIN` instruction is decoded, a snapshot is taken of the present processor state. The snapshot contains the register-rename file and the associated "S" bit vector. After graduation of a `XBEGIN` instruction the "S" bit vector defines the set of *architecturally active registers*, all the physical registers with "S" bit set are put in a FIFO queue: *Reserved Register List* (L_{res}) (which is the list of physical registers whose corresponding architectural register may not be used as the target of any write instruction). Those registers whose "S" bit is 0 are enqueued in the *Free Register List* (L_{free}). Thus $L_{res} \bigcup L_{free}$ =all physical registers.

In case the transaction commits successfully (reckoned by graduation of a `XEND` instruction) the registers in the *reserved list* (L_{res}) enqueued to the *free register list* (L_{free})and the transaction snapshot is deleted. In case the transaction aborts (ascertained by a jump to `pc`), the register rename file is restored to the processor and the reserved list(L_{res}) is drained down to the free list(L_{free}). Note that "S" bit serves the purpose of preserving the contents of the state registers during the execution of the transaction. Since the state registers are preserved during the transaction, restoration of the register-rename file suffices to restore the CPU state. Note that in this design the CPU re-order buffer has no role to play, thus the size of transaction is not limited by the window size of the processor architecture. Thus snapshot plays the same role as re-order buffer plays in conventional TM [10] approaches.

It may be noted that the transactional snapshot, nested counter (to keep track of depth of nesting) and `pc` (transaction abort handler) constitute the transaction state and are made available to the operating system as set of register values so that it may be saved/restored while context-switching along with the Program Control Block (PCB).

3.1.3 Modeling Memory state

UTM approach to transaction memory does n't depend on the cache-coherence protocols and coincidence properties to ensure the concurrency control. It does entire bookkeeping of the memory block using a data structure called *xstate*. The specification of the *xstate* structure is as follows (detailed structure illustrated in figure 4):-

1. *xstate* data structure contains an entry (called *transaction log*) for every active transaction in the main memory.

2. The *transaction log* consists of a *commit record* and an array of *log entries* for each memory access done by the transaction.

3. The *commit record* can be either of:- *pending* or *aborted* or *committed*.

4. The *log-entry* contains the address of the memory location, the type of operation (read-/write) and the old value stored at the memory location (in case of a write log record).

Note that the old values are stores with the log rather than new values because the common case is that transaction will commit. So there won't be any need to restore the memory blocks to previous values. Another design requirement of the UTM is RW bits in each block of the memory, these bits denote whether the block has participated in a transaction, and if yes then what type of operation was performed on the block (`store/load`). Furthermore each block has a *log-pointer* that points to the log entry for the last access on the block.

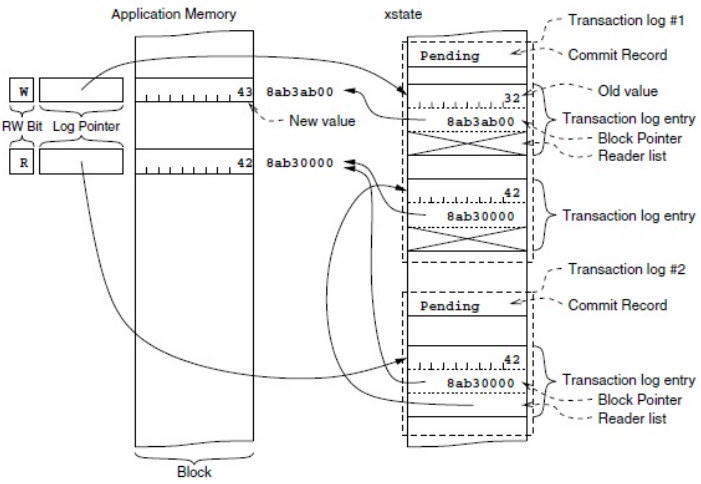

Figure 4: Schematic representation of xstate data structure

Thus for committing an instruction the *commit-entry* of the *xstate* data structure is changed from *pending* to *committed*. After that *transaction log* for that transaction may be deleted. For rollback action the transaction log is traversed and every memory location is changed to its previous value, also the RW bits of the corresponding block are set to 00 and the *log-pointer* is set to `NULL`.

3.1.4 Effect of Cache and System issues

Presence of cache greatly improves the performance of UTM design, because if the transaction fits into the main memory then the UTM model sniffs the cache-coherence protocol traffic to implement coherence and identify conflicts. The cache protocol gets an exclusive ownership of a cache line when a transaction attempts to write, thus until there is no request for upgrading the ownership there is no conflict and the transactions run with the overhead of normal cache coherence traffic. If the transaction does n't fit in the cache then the xstate data structure overflows into the main memory (or a L2 cache as the case may be with memory hierarchy). The *log pointers* to cache blocks are not required to be updated until the cache block is evicted (because while in cache the concurrency control is done via the help of cache coherence protocol), the the block pointer in kept clean until the cache block is not evicted. To make the common case of small transactions fast [3] the new data is always kept in the cache while the old value for the memory location may be kept in the main memory. Thus if the transaction completes successfully (most common case) then cache is already upto-date, in the other case when transaction fails there is an added overhead of chasing pointers to rollback (thus the common case is optimized). Furthermore it is not necessary that the *xstate* data structure in the main memory is consistent with the in-cache transaction state.

UTM systems also support migration of the transaction from one processor to another (for long transactions this might be the case). This is archived by making the log-pointer for the process log in the shared *xstate* structure as a part of the CPU register which results in the *log-pointer* for the process being saved with the PCB.

The transaction size may span beyond the physical memory into the virtual memory which can be accomplished by using *Global virtual address* (it is a unique address for the each memory location in the entire virtual memory space) in the *xstate* data structure. The transaction log information is also stored in the main memory with the process control block which enable the

Operating System to swap-in and swap-out the process without any modifications to the scheduler. The paging of *xstate* data structure is dependent on the operating system, either the entire range of memory locations "touched" by the processor may be loaded into the main memory or the operating system must allow restarting of another `load` instruction (for swapping in the page from disk) in middle of a memory transaction (which is also a `store/load` instruction). It may me noted that the UTM system does n't allow I/O in between transactions i.e. there can't be any device I/O in the code excerpt between a `XBEGIN` and the corresponding `XEND` instruction.

3.2 Large Transactional Memory (LTM)

The previously discussed UTM system requires a lot of modifications in the computer architecture and operating system, further a transaction seldom overflows the main memory (and in modern systems there are humongous main memory sizes so it shall never be a problem). This motivates to decrease the power of UTM system to allow transactions in only the size limit of main memory but at a reduced and practical modification of architecture (a compromise between the practicability and programmability). This design of memory called *L*arge *Transactional M*emory (LTM) compromises on the following aspects of UTM:-

"Unboundedness" : The LTM design limits the size of the transaction footprint by the size of the main memory. Thus a transaction can be at most as large as the main memory (well not as large but almost considering the space requirement of in-memory book keeping).

Time Bounded : In the LTM architecture the memory transaction is bounded by the time slice OS allocates to the transacting process. Thus the transaction state is no longer a part of the process state.

No Migration : Unlike UTM processes (or transactions) can't be migrated from on process to another. This again follows logically from the previous limitation that the entire transaction state is no longer contained in the process state.

These relaxations on the requirements of UTM system drastically reduces the implementational complexity of the LTM systems. The LTM systems described herein were the first step towards creating an "unbounded" system like UTM. LTM shares a lot of concepts of concurrency management from UTM is shall be clear from the design specifications.

3.2.1 Model Specification

As described in the UTM model the semantics for the `XBEGIN` and `XEND` are same for the LTM systems also. There are however some design differences that reduce the complexity of implementation. LTM does n't use any shared memory data structure like *xstate* to manage transaction state, instead the transaction state is maintained partly in the cache and partly in the operating system reserved memory.

LTM system has some small changes in the cache memory organization. The speculative transaction state of small transactions is stored directly in the cache memory itself, for large transactions it is overflowed to a hash table in the main memory. For the purpose of maintaining transaction state in the cache following cache modifications are proposed:-

"T" bit : Each cache block contain an additional "T" bit that is set to 1 if a cache hit occurs on the cache line during execution of a transaction (i.e. after graduation of a `XBEGIN` instruction). Thus for any cache line this bit signifies whether the cache line is participating in any transaction.

"O" bit : This additional bit is added per cache *set* (as in set-associative cache), which is set to 1 if a cache line in the particular set is evicted because of cache capacity reasons. The cache line is the available in the overflow data structure.

The main memory always contains the original data while the cache and the overflow hash table contains the speculative data (based on the speculation that the transaction completes). Thus when the transaction completes all the "T" bits are cleared from the corresponding cache blocks and all overflowed data is written back to the main main memory. The atomicity of transactions is guaranteed using cache-coherence protocols, if there is any cache hit on a cache line that already has "T" bit set and the cache line does n't belong to the transaction in question then it is taken as a conflict and the transaction is aborted.

The overflow hash table acts like an extension to the cache, if there is a cache miss then processor tries to find the cache line in the overflow hash table within a limited amount of time. If the cache line is found then the it is swapped with the existing cache line and it is treated by all means as a cache hit, otherwise if the line could n't be found in the overflow data structure then it is taken as a cache miss. The hash table takes uses the higher order of the memory address as the hash key, hash conflicts are handled by using linear chaining of the hash entries.

3.2.2 Memory and Processor state

As discussed in the case of UTM the transaction state can be divided into memory and processor state. For the LTM the processor state is saved just like in the UTM by introducing "S" bits in the processor register-rename file. As far as the main memory state is concerned it is never changed until the transaction commits, as described above the speculative transaction state of the system is limited to the cache and the overflow block. The cache state is easily restored by clearing all the corresponding "T" bits and clearing the entries in the overflow hash table. Thus for transactions footprints that fit in the cache the case is equivalent to a conventional HTM system.

3.2.3 Importance of cache

As should be evident from the previous discussion the presence of cache is mandatory for correctness of the LTM system (compared to the UTM systems where the cache just add a performance boost to the system). But in most of the commonly available architectures cache memory is present. The overflow hash table serves as an "in-memory" cache serving just as an expansion of the present cache system. Thus cache memory ceases to be just a speedup factor but becomes versatile for the LTM implementation. The overflow hash table is however maintained by the processor itself (with support for management by operating system itself), so there is minimal overhead of interrupts.

3.3 Conclusion

To conclude it can be reckoned that UTM system are representative of idealized solutions to the "unbounded" transaction footprint problem and like most idealized solutions suffers the drawbacks in practicability, cost and implementational complexity. LTM on the other hand represents a much more technical approach because the transactions are in general small enough to fit in the main memory, as a matter of fact study [2] shows that $\approx 99.99\%$ of transactions touch below 54 cache lines. However UTM is much more generic and provides beter encapsulation and functionality than the LTM which compromises programmability for complexity.

4 Parallel Programming with Transactional Memory

4.1 Introduction

Since the growth of fast uniprocessor has reached saturation the chip manufacturers are turning to multi-core processors. In order to reduce the total run time by increasing the cores, the programs have to be parallelized. This is expressed by Amdahl's law as:

$\frac{1}{(1-P)+\frac{P}{S}}$.

Here P is the fraction of the program that can be parallelized and S is the number of execution units.

The description work and the codes are taken from [5]

4.1.1 Synchronization problems

Implementing parallization is faced by many problems:

- The program should comprise of multiple independent parts which can be written as seperate programs. Shared memory is usually used to collaborate these multiple programs.
- Read and write accesses to shared data should be synchronized in order to avoid inconsistent states.
- To deal with multiple memory locations, mutex (mutual exclusion) directives are used. But, it brings new set of problems. If single mutex is used in the whole program then the portion of program that can be parallelized(P) would be reduced and the program performance would be affected. Multiple mutexes increases the probability of deadlock and also the overhead for locking and unlocking mutexes.

4.1.2 Programmer's dilemma

By increasing the portion of code which can be parallelized (P), the complexity and overhaead of code increases which introduces new problems. So, an optimum code has to be decided upon which is not very complex and at the same time sufficiently parallelized.

4.2 Transactional Memory

The problem of consistency exists in computer science especially in databases in which it is solved using transactions. Transactions can be performed in any order and are committed only if successful. They are rolled back and aborted if conflict arrises. The same concept can be use to perform memory operations in which the in-memory data a program keeps corresponds to the tables in the databases. But this will restrict the programmers to write their programs in a certain way. Fortunately, the concept of Trancsactional Memory[2] has been defined without this restriction.

The description of the hardware implementation of TM in [10] is generic and has been explained above. It is not necessary to implement TM in harware and the implementation details must be transferred to today's available hardware. 'Hardware TM[3] provides a means to implement atomic operations operating on more than one memory word' [5]. For further support in

[2]Denoted by TM henceforth

[3]Denoted by HTM henceforth

implementing TM, software support is needed. Software TM[4]-based solutions provide interfaces to TM funtionality, which later could be integrated with harware solutions to form **hybrid TM**.

4.2.1 Show me the problem

A small example can be shown to highlight the problems that can happen in real code:

```
long counter1;
long counter2;
time_t timestamp1;
time_t timestamp2;

void f1_1(long *r, time_t *t) {
 *t = timestamp1;
 *r = counter1++;
}
void f2_2(long *r, time_t *t) {
 *t = timestamp2;
 *r = counter2++;
}
void w1_2(long *r, time_t *t) {
 *r = counter1++;
 if (*r & 1)
  *t = timestamp2;
}
void w2_1(long *r, time_t *t) {
 *r = counter2++;
 if (*r & 1)
  *t = timestamp1;
}
```

Our goal is to make this code runnable on multiple threads which can execute any function concurrently without producing invalid results in which the return counter and timestamp values dont belong together.

If we use a single mutex in the whole program and assuming that most of the time only the functions $f1\_1$ and $f2\_2$ are called then the perfomance is reduced unnecessarily since $f1\_1$ and $f2\_2$ are already conflict free.

If we use two locks, the semantics would have to be in the one case when counter1 and timestamp1 are used and when counter2 and timestamp2 are used, respectively. Although it will work when $f1\_1$ and $f2\_2$ are called but not for the other two functions where the pairs counter1/timestamp2 and counter2/timestamp1 are used together. So, then, we would have to use separate locks for each variables.

In that case the following code could be written (only two functions are mentioned here; the other two are mirror images):

```
void f1_1(long *r, time_t *t) {
 lock(l_timestamp1);
 lock(l_counter1);
 *t = timestamp1;
 *r = counter1++;
}
void w1_2(long *r, time_t *t) {
 lock(l_counter1);
```

[4]Denoted by STM henceforth

```
 *r = counter1++;
 if (*r & 1) {
  lock(l_timestamp1);
  *t = timestamp2;
  unlock(l_timestamp1);
 }
 unlock(l_counter1);
}
```

In the code for $w1\_2$ delay in getting the $l\_timestamp1$ lock might produce inconsistent results. The lock has to be accessed at the start:

```
void w1_2(long *r, time_t *t) {
 lock(l_counter1);
 lock(l_timestamp1);
 *r = counter1++;
 if (*r & 1) {
  *t = timestamp2;
  unlock(l_timestamp1);
  unlock(l_counter1);
}
```

Still, the code has flaws. The order in which the required locks are accessed is different in $w1\_2$ from that in $f1\_1$. This will bring in the possibility of deadlocks. Hence, we can conclude that there can be situations when multiple mutex locks become necessary. But the code becomes too complicated. STM provides a different approach to consistency problem.

4.2.2 Rewriting using TM

In the following example nonstandard extensions to C are used which might appear in a TM-enabled compiler.

```
void f1_1(long *r, time_t *t) {
 tm_atomic {
  *t = timestamp1;
  *r = counter1++;
 }
}
void f2_2(long *r, time_t *t) {
 tm_atomic {
  *t = timestamp2;
  *r = counter2++;
 }
}
void w1_2(long *r, time_t *t) {
 tm_atomic {
  *r = counter1++;
  if (*r & 1)
  *t = timestamp2;
 }
}
void w2_1(long *r, time_t *t) {
 tm_atomic {
  *r = counter2++;
  if (*r & 1)
  *t = timestamp1;
 }
}
```

Here, all the instructions in the **tm_atomic** block form a transaction. Functions should have prior knowlege of the transactions. So, two versions of each function, one with TM fucntionality and one normal, are necessary. The following following steps show how TM can be implemented [5]:

1. Check whether the same memory location is part of another transaction.
2. If yes, abort the current transaction.
3. If no, record that the current transaction referenced the memory location so that step 2 in other transactions can find it.
4. Depending on whether it is a read or write access, either
   (a) load the value of the memory location if the variable has not yet been modified or load it from the local storage in case it was already modified, or
   (b) write it into a local storage for the variable.

If the transaction already accessed the same memory location previously then step 3 can be ignored. Transaction can be aborted in lazy/lazy (lazy abort/lazy commit) method or in eager/eager (eager abort/eager commit) method. In the former, the transaction is executed until it reaches commit stage even it is aborted beforehand. In the latter, the trasaction is aborted and stopped as soon as conflict is detected. Eager/eager method is usually followed in database transactions.

The commitment of a transaction can be described as follows [5]:

1. If the current transaction has been aborted, reset all internal state, delay for some short period, then retry, executing the whole block.
2. Store all the values of the memory locations modified in the transaction for which the new values are placed in local storage.
3. Reset the information about the memory locations being part of a transaction.

4.3 Correctness and Fidelity

While discussing about correctness, it is obviously assumed that STM is correctly implemented without bugs. A transaction is committed only if it remains unaborted and all its atomic blocks succeed. So, with respect to memory accesses, the thread is the only thread executing and the new code is as correct as the original code without locks. Also, it is theoretically possible to show that this TM technology is deadlock-free. It can be explained by stating a similar problem. In IP-based networking if more than one machine start sending out data concurrently then this conflict is detected and the sending is postponed until a short interval of time. Practically IP-based networking is successful and hence a similar result can be expected from the TM problem. Also, in our example the code for $f1\_1$ and $f2\_2$ will run concurrently since they refer to disjunct memory locations.

4.4 Where is TM today?

TM is still a topic of research today. Few implementations of STM include the VELOX project[5] and TinySTM[6]. Also, first proprietary compilers which support STM are also available [7]. These implementations will help gain experience and find solutions to remaining problems:

1. **Recording transactions** The state of the atomic block including the memory locations of the variables used in it have to be recorded before entering the atomic block. Hence, there would be an overhead of several words for each variable in the cache which is very costly. Although, in case the memory location is already in the block, it would not have to record it. But, if in the above example we assume that in the final code all four variables are in the same atomic block leads to high abort rate. This reduces the perfomance substantially.

2. **Handling aborts** As described earlier aborts can be handled in lazy/lazy (lazy abort/lazy commit) method and eager/eager (eager abort/eager commit) method. In both the cases the old values of the variables have to be stored locally to be restored in case of an abort. There are many other ways in which aborts can be handled but similar to the above two methods in efficiency. There is a high dependency on the abort rate of individual transactions.

3. **Semantics** TM has to be integrated with the established computer languages. Also the semantics of the atomic block have to be specified. Nested TM and treatment of local variables are other issues.

4. **Performance** Performance is the sole reason why TM is a topic of research. The compiler has to optimize the code. For example, if the TM is not needed (e.g., in a single threaded-program) then, to prevent the overhead of TM two versions of each function has to be written.

4.5 Conclusion

TM is attractive and makes parallel programming much efficient. The first implementations are out but much research has to be done.

References

[1] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, PODC '92, pages 125–134, New York, NY, USA, 1992. ACM.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. ACM, Feb 2005.

[3] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[5] http://www.velox-project.eu/

[6] http://tinystm.org

[7] http://www.hipeac.net/node/2419

[4] David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: A scalable cache coherence scheme. *SIGPLAN Not.*, 26:224–234, April 1991.

[5] Ulrich Drepper. Parallel programming with transactional memory. *Queue*, 6:38–45, September 2008.

[6] James R. Goodman. Using cache memory to reduce processor-memory traffic. *SIGARCH Comput. Archit. News*, 11:124–131, June 1983.

[7] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. pages 123–136, 1996.

[8] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[9] M. Herlihy. A methodology for implementing highly concurrent data structures. *SIGPLAN Not.*, 25:197–206, February 1990.

[10] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.