Transactional Memories: a theoretical introduction

Selim Arsever - *selim.arsever@epfl.ch* & Pascal Perez - *pascal.perez@epfl.ch*

1 abstract

Concurrent programming is usually very complex when dealing with data structures consistency shared among many processes. Traditional lock-based techniques do not scale and may even break when placed in arbitrary contexts. In this paper, we introduce modern ways to implement lock-free dynamic data structures, simply and efficiently using transactional memories.

2 preliminary

Before delving into the lock-free world, we are going to add syntactic sugar to the polyadic version of the π-Calculus introduced by Robin Milner in [7]. We will use

$$[u = v]\, P \stackrel{def}{=} (\nu y)\,((\nu uv)\,(\overline{u}\,\langle y\rangle \,|v(z).\overline{z}\,)\,|y.P\,)$$

which is a π-Calculus formulation of a simple channel equality. If the u is the same name as v, they will be able to interact and thus synchronize with the guard posted in front of P to make is start. Otherwise, there will be no further interactions and the resulting process will be bisimilar to the **0** process.

3 sharing memory

Before digging into transactional memories, we shall have a brief overview of lock-based techniques, why they are required and why they are inefficient. Throughout this article, we will consider a one bit memory defined by the following π-process .

$$M(w,r,v) := w(f).M\langle w,r,f\rangle + \overline{r}\langle v\rangle.M\langle w,r,v\rangle$$

Its behavior is straightforward, the memory cell is initialized to a certain value v and can be written (w) or read (r) at any moment. We can then use this memory for temporary storage whilst doing some *work*, symbolized by a $WORK$ process (which we suppose to be autonomous, $fn(WORK) = \emptyset$).

$$A(w,r) := (\nu v)\,(\overline{w}\langle v\rangle.WORK.r(v').[v' = v]A)$$

The process writes to memory, *works* and either reads a correct value from memory and continues or fails. We can then compose both $A\langle w,r\rangle\,|M\langle w,r,v\rangle$ which result in a simple process writing to memory, working, reading and starting over. This process could correspond to a simple single threaded program interacting with main memory. Now, making the situation a little more complex $A\langle w,r\rangle\,|A\langle w,r\rangle\,|M\langle w,r,v\rangle$ – which may be a naïve way to use a shared memory – introduces failures which will result in one process stopping.

For example, if the first A process begins to write, does some work, is preempted for the second A process that writes something else to memory, then the first A process continues to execute, we find our self in a situation where it expect one value but finds another, thus failing.

4 locking

The common problem outlined above pushes developers to use locks on memory cells, thus forbidding interleaved accesses resulting to erroneous and incoherent data. We can implement that feature on our memory cell.

$$\begin{aligned} LM(w,r,v) &:= l.M\langle w,r,v\rangle \\ M(w,r,v) &:= \overline{u}.LM\langle w,r,v\rangle + \ldots \end{aligned}$$

Here the l and u channels correspond to *lock* and *unlock*, respectively. Modifying our preceding process to use the locking techniques.

$$A(w,r) := (\nu v)\left(\overline{l}.\overline{w}\langle v\rangle.WORK.r(v').[v'=v]u.A\right)$$

Note that if a process fails (reads incoherent data) it does not release the lock. This unfortunate "property" will be addressed in more detail later. Nevertheless, the designed solution works fine when composed with any number or processes $!A\,|M$.

5 deadlocks

Problems with locks begin to occur when more than one lock is needed. For example let's look at this simple system:

$$A_1\langle w,r\rangle\,|A_2\langle w,r\rangle\;|LM_1\langle w,r,v\rangle\;|LM_2\langle w,r,v\rangle$$

In such a situation we find ourself with the risk off a deadlock when the two processes take and release the locks in a particular order. Let's partialy define those two processes in such a way.

$$\begin{aligned} A_1(w,r) &:= (\nu v)\left(\overline{l_1}.\overline{w_1}\langle v\rangle.\overline{l_2}.\overline{w_2}\langle v\rangle...\right)\\ A_2(w,r) &:= (\nu v)\left(\overline{l_2}.\overline{w_2}\langle v\rangle.\overline{l_1}.\overline{w_1}\langle v\rangle...\right)\end{aligned}$$

It is easy to see that if A_1 begins to execute, stops after taking the first lock, A_2 begins and then takes the second lock the system is dead - nothing can happen anymore.

Of course such a simple situation can be avoided by programming conventions but they require a global knowledge of the locking policy. With very big systems this becomes nearly unfeasible. The situation is even worst when dealing with functions. As a matter of fact, if a function uses locked data structure, one should be very careful when using in arbitrary environment. There is no systematic approach for dealing with such problems.

One can easily be convinced of the practical reality of these issues by looking at the Linux kernel where a Big Kernel Lock (BKL) has been implemented to deal with such messy situations. Indeed, fine grain locking is way too complicated to achieve! Locking an entire system is clearly a very sub-optimal solution - not even to mention that it is very unaesthetic.

In addition, real life processes (threads) can fail. If they do whilst holding a lock the system may not be able recover at all.

To sum up we argue that lock based techniques are bad because they require a global knowledge of the application, may durably degrade the system's performance due to obstruction and do not scale well. Those problems justify the need for another way of dealing with shared memory. Transactions and in particular transactional memories are one of them.

6 transaction

A transaction is a set of sequential operations that can be interleaved freely with other operations and yet result in the same, overall, result. Thus a transactional system seems - to an outside observer - as advancing by atomic steps. A transaction is usually said to have the *ACID* properties, that is:

Atomicity refers to the capacity of a transaction to either *commit* or *abort*. If a transaction commits, the system is updated atomically - as seen by an outside observer - and if it aborts the system is left unchanged.

Consistency states that if the system is in a *legal* state, it will also be in a *legal* state after a transaction commits.

Isolation ensures that opened transaction cannot observe another transaction's effect that is not committed.

Durability requires the transactions that commits to alter durably the system (changes are durably recorded).

For instance, in a system where $A+B+C$ must stay constant, the two transactions $\mathbf{T}_1$ and $\mathbf{T}_2$ that follow conform to the required properties.

$\mathbf{T}_1$	$\mathbf{T}_2$
$A = A - 10$	lock L_1
lock L_1	$B = B - 20$
$B = B + 10$	unlock L_1
unlock L_1	$C = C + 20$

Nevertheless, it is simple to construct two "transactions" that fail and produce wrong result.

$\mathbf{T}_1$	$\mathbf{T}_2$
lock L_1	lock L_2
$A = A - 10$	$B = B - 20$
unlock L_1	unlock L_2
$B = B + 10$	$C = C + 20$

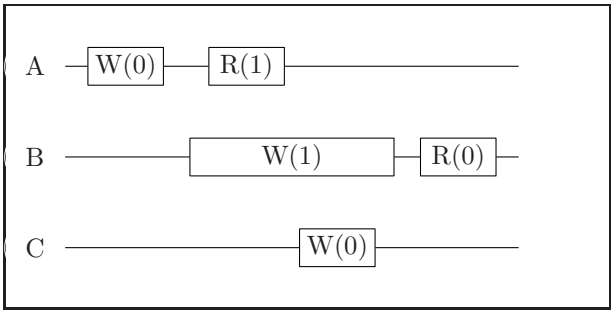
The problem here arises from the shared variable B which can be read by $\mathbf{T}_1$, modified by $\mathbf{T}_2$ meanwhile and then overwritten to a wrong value by $\mathbf{T}_1$.

7 transactional memories

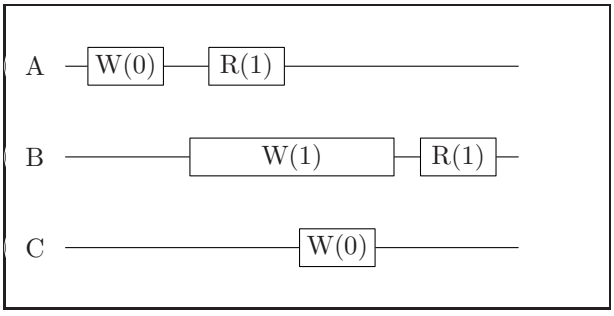
We are now going to introduce and explain the transactional memory (TM) principle. Classically, memory is regarded as a rather passive entity. Its state - or content - can be read and modified at any time, without any control. On the other hand, transactional memories constrain the access and ensure the memory will be accessed in a transactional way. Whatever the outside world, the TM ensures that the ACID properties are met.

In the TM system we present, processes wishing to interact must start a transaction explicitly and then open a transactional object (in our case a one bit memory cell) requesting a transactional object. When they have a transactional object, they can freely update it as if being in sequential context. When finished, they either commit or abort the transaction. Aborting a transaction simply discards all changes and closes the transaction. On the other hand, committing requires the TM system to validate the transaction and may launch and exception, "forcing" the process to restart. Nevertheless, if the commit succeeds, the internal object is atomically updated.

Here are some time lines representing three processes reading and writing a one bit transactional memory. The first on shows a possible flow of event whereas the second one is incorrect:



first time line (*correct example*)



second time line (*incorrect example*)

The second example is incorrect because when A reads one it implies that B has already finished to write one. Therefore, C writes a zero after B and then B cannot read a one later on.

8 π-Calculus modeling

In order to have cloneable memory cells, we are going to reuse the one introduced earlier, adding the possibility to create a new copy (clone the memory):

$$\begin{aligned} M(w,r,v,n) \quad := \quad & w(f).M\langle w,r,f,n\rangle + \overline{r}\langle v\rangle. \\ & M\langle w,r,v,n\rangle + (\nu w',r',n') \\ & (\overline{n}\langle w',r',n'\rangle.\,(M\langle w,r,v,n\rangle \,|M\langle w',r',v,n'\rangle\,)) \end{aligned}$$

Here, the write (w), read (r) and value (v) channels are homomorphic. The extra channel "new" (n) is there to copy the memory. When called, it returns the new communication channels to the copied memory cell.

Now that we have the basics, we can construct the TM system. Lets emphasize here that we require our

system to be non-blocking [5] requiring that the failure or indefinite delay of a process cannot prevent other processes from making progress. The TM system is able to answer to any begin request at any time. Each processs interaction with the TM can be seen as an independent sequence of operations, requiring synchronization only at commit.

The TM system presented here has a protection mechanism *crash* which permits the "banged" process G to be deafened. As you can see, there is only one open *crash* output at a time. Well will see later that the G process consumes one in a special state and therefore terminates the replication possibilities.

$$TM(b,n) := \left(\nu\vec{t}\right)\left(\overline{crash}\,|!\left(crash.G|\overline{crash}\right)|!p(x).k.\overline{x}\right)$$

where $\vec{t} = (crash,p,k)$. Another subtle point is the $!p(x).k.\overline{x}$ parralel part. The p channel stands for *put* and the k channel for *kill*. This little structure stores temporarely x channels (*exceptions*) for a deffered call triggered by k. Let's now look at G's process definition. The channels meanings are as follows: b for *beginning* a transaction, o for *opening* a transactional object, g to *get* the communication channels connected to the newly created transactional object, a for *abort* and c for commit.

$$\begin{aligned} G \quad := \quad & b(o,g,a,c,x).\overline{p}\langle x\rangle.o.n(w',r',n').\overline{g}(w',r'). \\ & \left(a + c.crash.n'(w'',r'',n'').\left(TM\langle b,n''\rangle|!\overline{k}|\overline{c}\right)\right) \end{aligned}$$

When a process wishes to open a transaction, it must send its open, get, abort, commit and exception channel so that the TM system can interact. When these are received, the exception channel is stored using the put channel in the $!p(x).k.\overline{x}$ structure studied before. After a call to put, the structure will look like $k.\overline{x_\alpha}|!p(x).k.\overline{x}$, therefore waiting for a kill to launch an exception. After the put has been made, the process can open a transactional object. The memory to which the TM system is linked (via the n channel) is copied and the write and read channel (w' and r') are sent via the g channel. Finally, the process can either abort a - discarding all changes - or c commit. The system must ensure that only one process can commit. Here, as there is only one *crash* output at a time, the first process to request a commit and whose G process internally synchronizes with *crash* will be able to confirm the commit ($\overline{c}$), all of the other process will have an exception raised on their private exception channel. After the *crash*, the system copies the transactional object on the n' channel (wich was received when copying the original object), confirms the commit and instantiates a new TM system linked to the transactional object of the successful process. In parallel, we kill $!\overline{k}$ all the other other process. The unsuccessful processes will have to restart the transaction on the b channel, that is connected to the newly created TM.

We are now going to describe the processes using this TM system.

$$\begin{aligned} T_\alpha(b) \quad := \quad & (\nu o,g,a,c,x)\left(\overline{b}\,\langle o,g,a,c,x\rangle\,.\right. \\ & \left.\overline{o}.g\,(w,r)\,.W_\alpha.\left(\overline{a} + \overline{c}.\left(c + x.T_\alpha\,\langle b\rangle\right)\right)\right) \end{aligned}$$

Here, processes begin their transactions (b) by sending private channels. The next step is to open (o) a transactional object, wait to get (g) the channels, work (W_α) and finally abort (a) or commit (c). When a process commits, it waits for the commit confirmation or for an exception (x). If an exception is raised, the transaction restars.

The overall system is a parallel composition of every described part with the necessary restrictions.

$$S := (\nu b)\left(\prod_\alpha T_\alpha\langle b\rangle\,|(\nu w,r,v,n)\,(\,TM\langle b,n\rangle\,|M\langle w,r,v,n\rangle\,)\right)$$

Event though the system is correct from a theoretical point of view, it has some flaws that we would like to remove. The deferred exception mechanism (DEM) $!p(x).k.\overline{x}$ is not satisfactory. Consider two transactions T_1 and T_2 that have already executed the put (p). Then the DEM will look like $k.x_1|k.x_2|!p(x).k.\overline{x}$. When one transaction successfully commits it will crash the sytem and the $!\overline{k}$ will be able to internally synchronize with $k.x_1$ or $k.x_2$. This is quite embarrassing as we do not want the committing transaction to receive an exception! We are therefore going to modify the DEM to

$$DEM :=!p(x,cancel).(k.x + cancel)$$

allowing to cancel a specified deferred exception call on a special dedicated channel *cancel*. This added

possibility shall be used in a different way

$$G := b(o,g,a,c,x).(\nu cancel)\,(\overline{p}\langle x,cancel\rangle ... \\ (a + c ... (...|\overline{cancel}.!\overline{k})))$$

which will therefore cancel the successfully committing transaction's deferred exception call and then allow the DEM to work as it did before.

Another problem we may want to address in an implementation is allowing exceptions as soon as a transaction has started a transaction. Indeed, in the present system exceptions are only taken into account after a commit. Yet it would be better to allow exceptions to be received at any time.

9 π-Calculus modeling - proofs

Atomicity This property requires that if a transaction aborts it does not modify the overall situation and that if it commits Atomicity This property requires that if a transaction aborts it does not modify the overall situation and that if it commits the changes are made atomically, in undividable steps as seen by an outside observer. Atomic here means that once a successful commit process has started and until it has finished the outside observers cannot modify the system. First of all, let's notice that when the system starts (in the state described by S) the memory cell M is visible only to the TM system. Indeed, the restrictions ensure that only the TM system can communicate with it. Furthermore, the TM system is only aware of the new (n) channel and therefore cannot write or read the memory. The only memory cells that are actually read and written are the copies (the transactional objects). Thus, when a transaction aborts (a) the internal memory, which reflects the systems state, is left unchanged. When a transaction commits (c) a copy of the transactional object owned by the committing transaction is made and a new instance of a TM system is made. These steps are all internal to the "transaction-TM" couple and therefore all the other transactions cannot observe a change. Note here that the commit confirmation ($\overline{c}$) is made asynchronously so that the system can continue running even if the distant process does not respond (has failed). In addition, the commit process cannot be preempted because the system is crashed and thus no transaction can pass the "*crash*" action of the G process. Overall, this ensures that a successful commit is atomic.

Consistency In our context, we have not defined what a legal state of the system is and it is therefore difficult to argue about Consistency In our context, we have not defined what a legal state of the system is and it is therefore difficult to argue about whether it is in one before and after a transaction.

Isolation This property is trivially respected in our system due to the cloning operation. In fact, a transaction works on Isolation This property is trivially respected in our system due to the cloning operation. In fact, a transaction works on a copy of a previously committed value. On-the-fly modifications are local to transactions.

Durability When a transaction commits, the transactional object it had is copied so that the internal memory cell managed by the TM system is always private (known only by the TM system). Therefore, the changes are durably recorded internally and cannot be modified without a begin/commit cycle.

Références

[1] L. Bocci, C. Laneve and G. Zavattaro. A Calculus for Long Running Transactions. In *In Proc. of Sixth IFIP Int. Conf. on Formal Methods for Open-Object Based Distributed Systems (FMOODS'03)*, pages 124-138, 2003.

[2] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA03, October 2630*, 2003.

[3] M. Herlihy and J. Wing. Linearizability A Correctness Condition for Concurrent Objects. In *ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3*, pages 463-492, 1990.

[4] M. Herlihy, V. Luchangco, P. Martin and M. Moir. Dynamic-sized Lockfree Data Structures. *Technical Report TR-2002-110, Sun Microsystems Laboratories*, 2002.

[5] M. Herlihy, V. Luchangco, M. Moir and W. Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *PODC 2003, July 1316*, 2003.

[6] M. Herlihy, V. Luchangco and M. Moir. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *Technical Report TR-2002-112, Sun Microsystems Laboratories*, 2002.

[7] R. Milner. Communicating and mobile systems: the π-Calculus . Cambridge University Press, 1999.