Software Transactional Memory: a solution to concurrency problems

Duilio J Protti

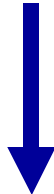
National University of Rosario, Argentina

December 6, 2005

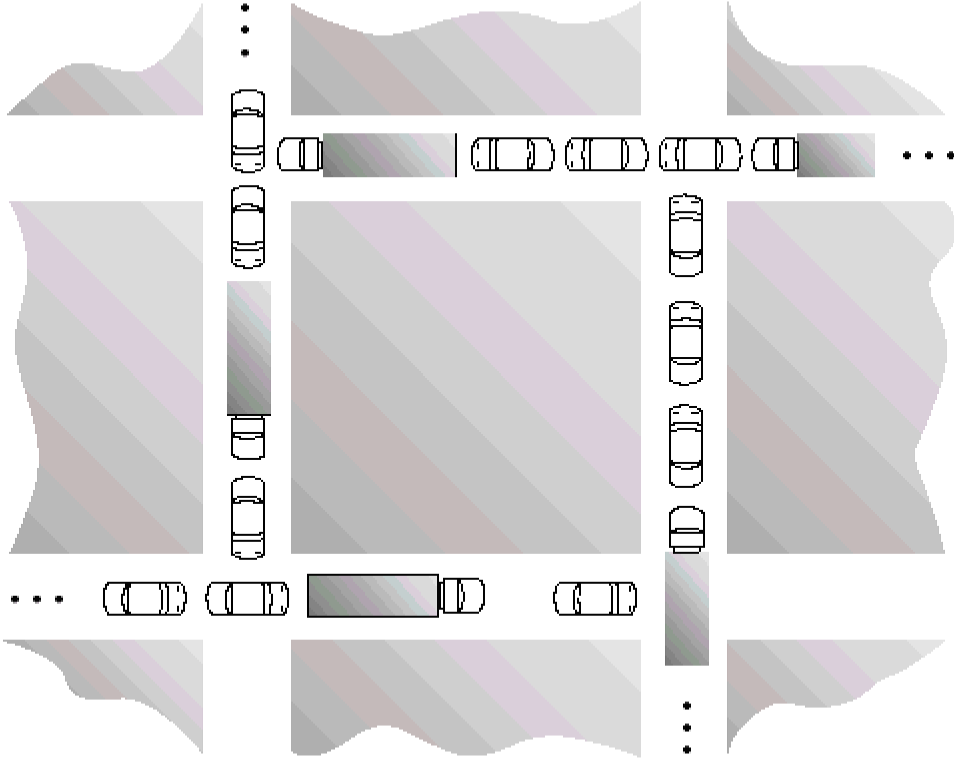
Deadlock: is not a problem of concurrent programming, but of one of its approaches

It appears in lock-based models

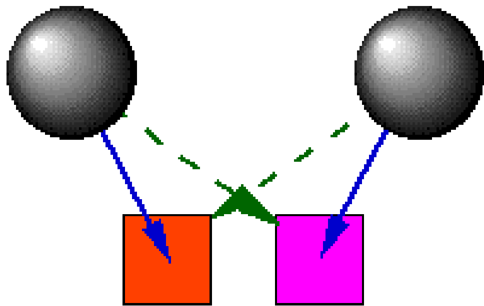
Deadlock can be avoided with careful usage



Or without using it at all...



Not using locks is the more insteresting choice because...

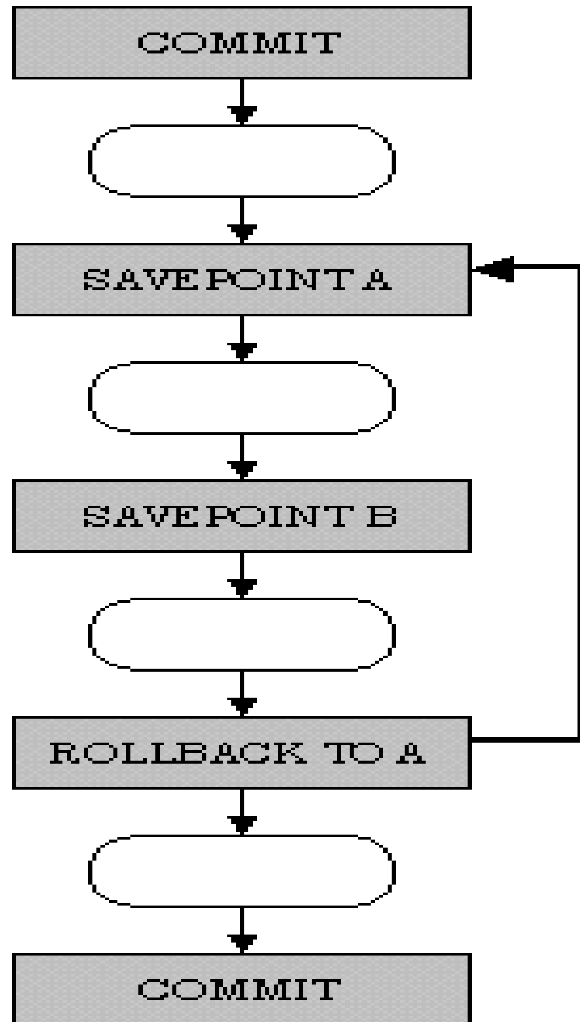


No locks, no deadlocks

But without locks, it can be handled concurrency?

Yes, it can, just another mechanism is needed

Commit/Rollback mechanism: propose changes, and drop it if they are not consistent



Participants involved does not "own" a shared resource

Every one works with a copy of the shared resource

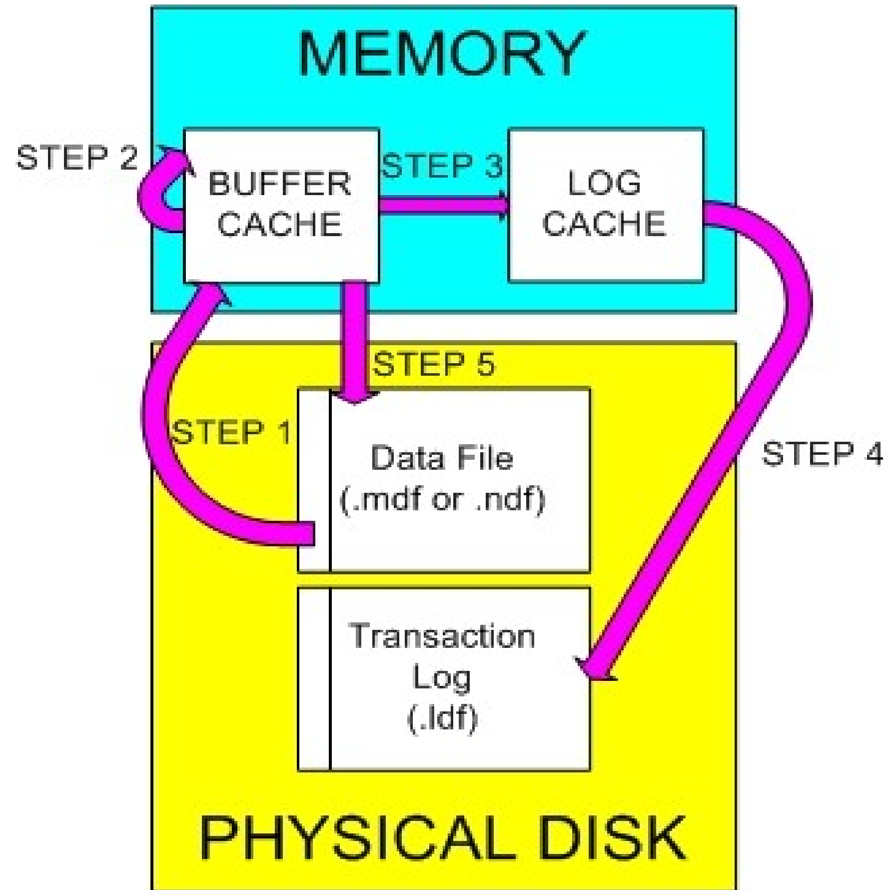
If at the end they have seen a consistent view of the resource, changes are made public

Lightweight Memory Transactions: different from the “heavy” counterparts found in DB's

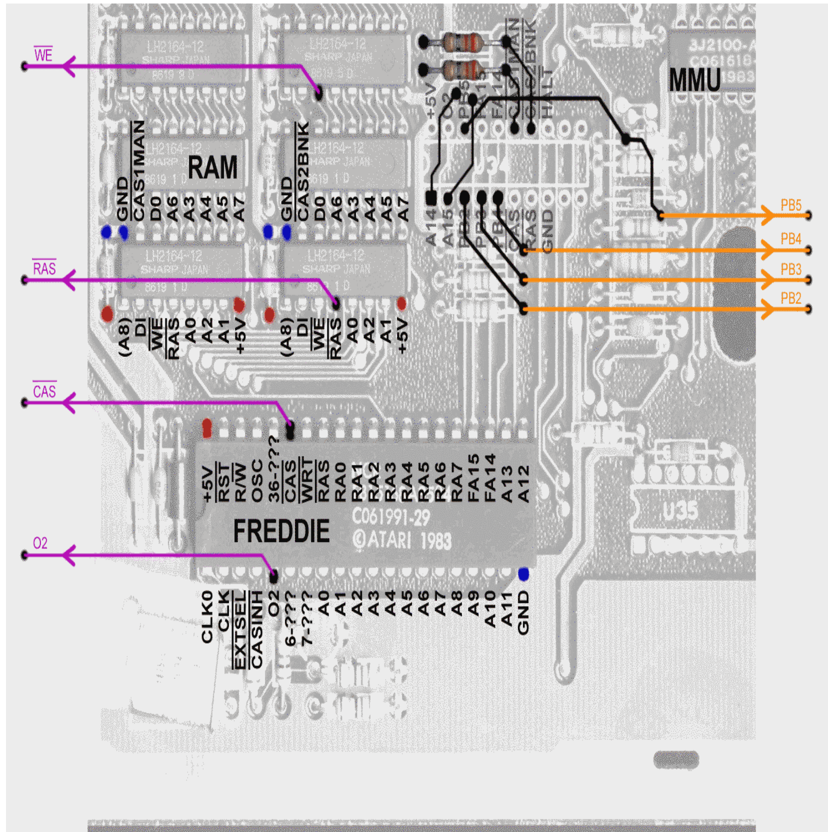
The *idea* of the commit/rollback is taken from the DB's

But the requirements are different

(Persistency, replication, fault tolerant, etc)



Software Transactional Memory: lightweight memory transactions in software

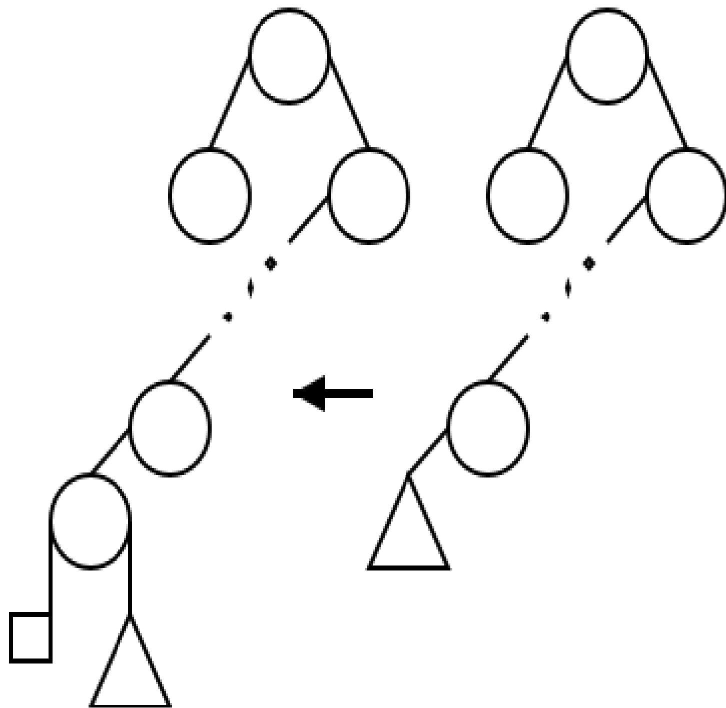


Lightweight memory transactions where first proposed as a hardware architecture

They didn't succeed

But with the current technology is practical to implement it in software

Limitation of previous STM implementations: lack of composability



I.e: two trees t1 and t2, with transactional operations insert() and remove()

It want to be done transaccionally:

{ x = remove(t1); insert(t2, x) }

Until recent, this does not scale well

Solution: Composable Memory Transactions, a composable STM model

Presented at the 2005 IBM Programming Languages Day

Implemented in Haskell (GHC 6.4)

It presents a modular (and composable) form to represent atomic actions, even in the presence of blocking ones

Composable Memory Transactions use a set of (composable) transaction combinators

```
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
atomic     :: STM a -> IO a
orElse     :: STM a -> STM a -> STM a
retry      :: STM a
```

atomic **gives a way to execute two transactions in sequence (as a transaction)**

orElse **allows to execute two transactions as alternatives**

Before to see CMT, let get a review of the **problems** associated with locks

1) Deadlock: due to locks acquired in wrong order
2) Lost wakeups: some conditional variable is not notified when changes occurs
3) Weak error recovery: lock's release in exception handlers
4) Tension between simplicity and scalability
5) They are not composable

Now let's take a look at how CMT works: optimistic concurrency

atomic

<body>

<body> is executed without acquiring any lock

Every read and write within <body> is made to a log which is private to the thread involved

In particular, writes are made to the log, not to shared memory

At the end, it try to commit. If it can't, the transaction is re-executed

A fundamental part of CMT: modular blocking

retry() causes to leave the actual attempt for the transaction, and the same will be executed again from the beginning (**but not immediately**)

The process will block waiting for any change in any of the variables readed by the transaction up to this point

When somebody commit to any of these variables, it wake up

```
atomic
    <... retry; ...>
```

Why this blocking is modular? Because of the orElse semantics

```
atomic
   <<body1>
      `orElse`
     <body2>>
```

<body1> is tried as a transaction, but if it blocks, <body2> is tried as a transaction

This allows to wait for many things at once

Is the dream of a composable select()

LibCMT, an implementation of Composable Memory Transactions

Implemented in C

Two datatypes are given: GTransaction and GTVar

Plus the operators of composition, atomic execution and modular blocking

It does not require garbage collector nor special memory allocators nor some particular thread model

http://libcmt.sourceforge.net

How to use CMT with LibCMT?

For every shared variable a transactional variable is created (GTVar) to represent it

Every thread creates the transactions (Gtransactions) in which it wants to be engaged

The actions of these transactions are performed by code written as if the program were sequential, with the difference that it access transactional variables instead of shared variables

Simple usage example: integer increment

```
void f (GTransaction *tr, gpointer user_data)
{
    int *i;

    i = g_transaction_read_tvar (tr, tvar_i);
    *i = ++(*i);
}

void worker_thread (void *data)
{
    ...

    tr = g_transaction_new (“Inc”, f, NULL);
    g_transaction_do (tr, NULL);
}
```

Example of composition in sequence: double integer increment

```
/* Same f() as previous */

void worker_thread (void *data)
{
    ...

    tr = g_transaction_new (“Inc”, f, NULL);
    tr2 = g_transaction_sequence (tr, tr);
    g_transaction_do (tr2, NULL);
    ...
}
```

A note about retry's implementation: non-local jumps

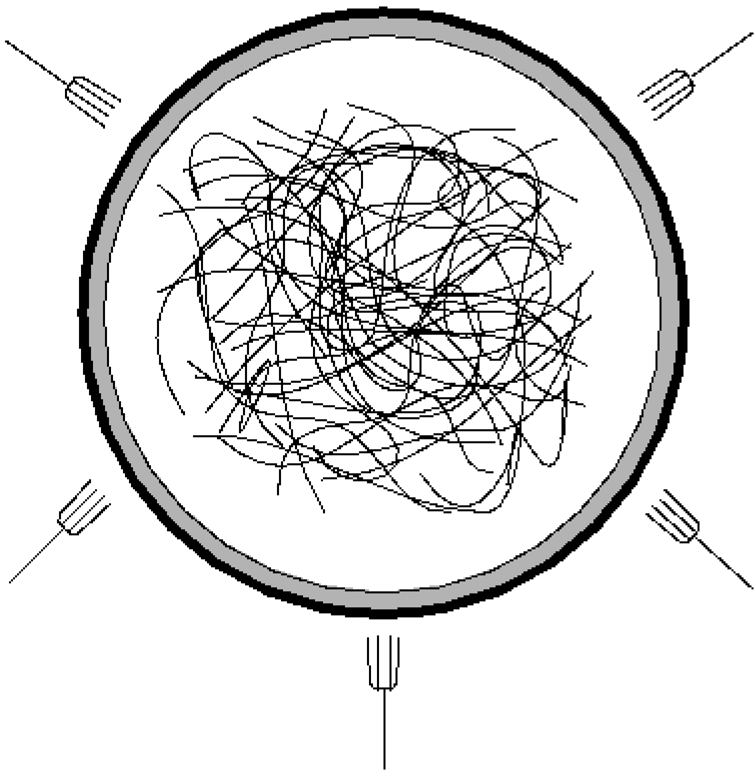
```
void h (GTransaction *tr, gpointer user_data)
{
    int *i;

    i = g_transaction_read_tvar (tr, tvar_i);
    if (*i < N)
        g_transaction_retry (tr);
    ...
}
```

It is not said what condition is expected, and blocking just **may happen** (but not always)

Make the condition explicit would break composability and ease of use. The library knows where to continue and who must be notified about changes

Example of composition with alternatives: The Dining Philosophers Problem



A transaction is created which try to take two adjacent forks

Then take_forks is created as the 'orElse' composition of five of that transactions

And that's all

Composition with alternatives: Dining Philosophers

```
void take_pair (GTransaction *tr, gpointer user_data)
{
    fork1 = g_transaction_read_tvar (tr, tforks[index]);
    fork2 = g_transaction_read_tvar (tr, tforks[(index+1)%NP]);
    if (fork1->in_use || fork2->in_use)
        g_transaction_retry (tr);
    fork1->in_use = fork2->in_use = TRUE;
}

void worker_thread (void *data)
{
    take_forks = take_pair_tr[0];
    for (i = 1; i < NP; i++)
        take_forks = g_transaction_or_else (take_forks,
                                            take_pair_tr[i]);
}
```

Coming back to the general STM, another advantage is: it offers the most parallelism

A concurrent program can be seen as a parcial order of elemental statements

Given a concurrent program, its “most parallel” version is that which can be executed in all the possible combinations given by that partial order (even the combinations which leaves to inconsistencies)

The ideal would be to have the “most parallel” version which is correct (from the point of view of concurrency)

STM offers, by design, the most parallelism possible

A lock-based mechanism works "serializing" portions of code (the *critical regions*)

This is done to avoid execution paths potentially incorrect. The problem is that this can impose more restrictions than needed. In particular when the critical regions are nested (like in compound transactions)

However, STM allows in principle all the possible executions, and then it forget the incorrect ones

It offers the most parallelism, always

In summary, lock-based mechanisms are not the only way to handle concurrency

In particular, deadlock is not an inherent problem of concurrency

Lesson learned: to note the difference between the problems of an issue, and the problems of its solutions

Questions?