Script **generated by TTT**

Title: Petter: Programmiersprachenh (21.11.2018)

Date: Wed Nov 21 14:17:05 CET 2018

Duration: 93:02 min

Pages: 29

Software Transactional Memory

Translation of `atomic`-Blocks

A TM system must track which shared memory locations are accessed:

- convert every read access `x` from a shared variable to `ReadTx(&x)`
- convert every write access `x=e` to a shared variable to `WriteTx(&x,e)`

Convert `atomic` blocks as follows:

```
atomic {
  // code
}
```

$\Longrightarrow$

```
do {
  StartTx();
  // code with ReadTx and WriteTx
} while (!CommitTx());
```

Translation of `atomic`-Blocks

A TM system must track which shared memory locations are accessed:

- convert every read access `x` from a shared variable to `ReadTx(&x)`
- convert every write access `x=e` to a shared variable to `WriteTx(&x,e)`

Convert `atomic` blocks as follows:

```
atomic {
  // code
}
```

$\Longrightarrow$

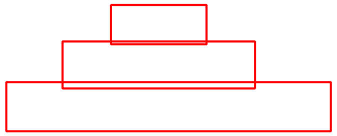
```
do {
  StartTx();
  // code with ReadTx and WriteTx
} while (!CommitTx());
```

- translation can be done using a pre-processor
  - determining a minimal set of memory accesses that need to be transactional requires a good static analysis
  - *idea*: translate all accesses to global variables and the heap as TM
  - more fine-grained control using manual translation
- an actual implementation might provide a `retry` keyword
  - when executing `retry`, the transaction aborts and re-starts
  - the transaction will again wind up at `retry` unless its *read set* changes
  - $\rightsquigarrow$ block until a variable in the read-set has changed
  - similar to condition variables in monitors ✓

A Software TM Implementation

A software TM implementation allocates a *transaction descriptor* to store data specific to each `atomic` block, for instance:

- *undo-log* of writes if writes have to be undone if a commit fails
- *redo-log* of writes if writes are postponed until a commit
- *read-* and *write-set*: locations accessed so far
- *read-* and *write-version*: time stamp when value was accessed

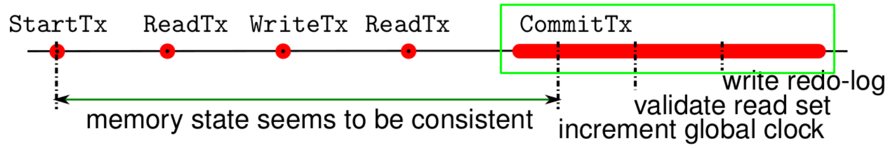


Principles of TL2

The idea: obtain a version from the global counter on starting the transaction, the *read-version*, and watch out for accesses to newer versions throughout the transaction.

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:



Other observations:

- read-only transactions just need to check that read versions are consistent (no need to increment the global clock)
- writing values still requires locks
  - deadlocks are still possible
  - since other transactions can be aborted, one can *preempt* transactions that are deadlocked
  - since lock accesses are generated, computing a lock order up-front might be possible
- there might be contention on the global clock

General Challenges when using STM

Executing `atomic` blocks by repeatedly trying to execute them non-atomically creates new problems:

- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
  - a TM implementation might impose restrictions:

```
// Thread 1                  // Thread 2
atomic { // clock=12
  ...                        atomic {
                               WriteTx(&x,0) = 42; // clock=13
                             }
  int r = ReadTx(&x,0);
} // tx.RV==12 != clock
```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object if the old object has not changed
  - $\rightsquigarrow$ idea of the original STM proposal
- TM system should figure out which memory locations must be logged
- danger of live-locks: transaction B might abort A which might abort B . . .

Integrating Non-TM Resources

Allowing access to other resources than memory inside an `atomic` block poses problems:

- storage management, condition variables, `volatile` variables, input/output
- semantics should be as if `atomic` implements SLA or TSC semantics

Integrating Non-TM Resources

Allowing access to other resources than memory inside an `atomic` block poses problems:

- storage management, condition variables, `volatile` variables, input/output
- semantics should be as if `atomic` implements SLA or TSC semantics

Usual choice is one of the following:

- *Prohibit It.* Certain constructs do not make sense. Use compiler to reject these programs.
- *Execute It.* I/O operations may only happen in some runs (e.g. file writes usually go to a buffer). Abort if I/O happens.
- *Irrevocably Execute It.* Universal way to deal with operations that cannot be undone: enforce that this transaction terminates (possibly before starting) by making all other transactions conflict.
- *Integrate It.* Re-write code to be transactional: error logging, writing data to a file, ...

Hardware Transactional Memory

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

$\rightsquigarrow$ limited by fixed hardware resources, a software backup must be provided

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is *eager* using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

⇝ limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:

1. Explicit Transactional Memory: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic
2. Implicit Transactional Memory: only the beginning and end of a transaction are marked
   - same instructions can be used, hardware interprets them as transactional
   - only instructions affecting memory that can be cached can be executed transactionally
   - hardware access, OS calls, page table changes, etc. all abort a transaction
   - ⇝ provides *strong isolation*

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel's TSX in Broadwell/Skylake microarchitecture (since $\boxed{\text{Aug 2014}}$):

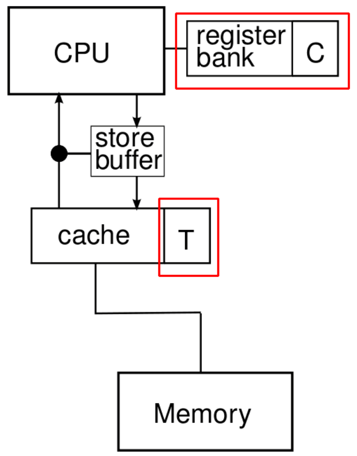
- *implicitely transactional*, can use normal instructions within transactions
- tracks read/write set using a single *transaction* bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources
- aborting in an inner transaction means aborting all of them

Restricted Transactional Memory

Implementing RTM using the Cache (Intel)

Supporting Transactional operations:

- augment each cache line with an extra bit T
- introduce a nesting counter C and a backup register set



$\rightsquigarrow$ additional transaction logic:

- `xbegin` increments C and, if $C = 0$, backs up registers and flushes buffer
  - subsequent read or write access to a cache line sets T if $C > 0$
  - applying an *invalidate* message from ~~*invalidate queue*~~ to a cache line with T flag issues `xabort`
  - observing a *read* for a *modified* cache line with T flag issues `xabort`
- `xabort` clears all T flags and the store buffer, invalidates the former TM lines, sets $C = 0$ and restores CPU registers
- `xend` decrements C and, if $C = 0$, clears all T flags, flushes store buffer

Restricted Transactional Memory

Provides new instructions `xbegin`, `xend`, `xabort`, and `xtest`:

- `xbegin` *on transaction start* skips to the next instruction or *on abort*
  - continues at the given address
  - implicitely stores an error code in `eax`
- `xend` commits the transaction started by the most recent `xbegin`
- `xabort` aborts the whole transaction with an error code
- `xtest` checks if the processor is executing transactionally

Considerations for the Fall-Back Path

Consider executing the following code concurrently with itself:

```
int data[100]; // shared
void update(int idx, int value) {
  if(_xbegin()==_XBEGIN_STARTED) {
      data[idx] += value;
      _xend();
    } else {
      data[idx] += value;
    }
}
```

Protecting the Fall-Back Path

Use a lock to prevent the transaction from interrupting the fall-back path:

```
int data[100]; // shared
int mutex;
void update(int idx, int value) {
  if(_xbegin()==_XBEGIN_STARTED) {

      data[idx] += value;
      _xend();
    } else {
      wait(mutex);
      data[idx] += value;
      signal(mutex);
    }
}
```

- the fall-back code does not execute racing itself ✓

Happened Before Diagram for Transactions

Augment MESI states with extra bit T. CPU A: d:E5 t:E0, CPU B: d:I

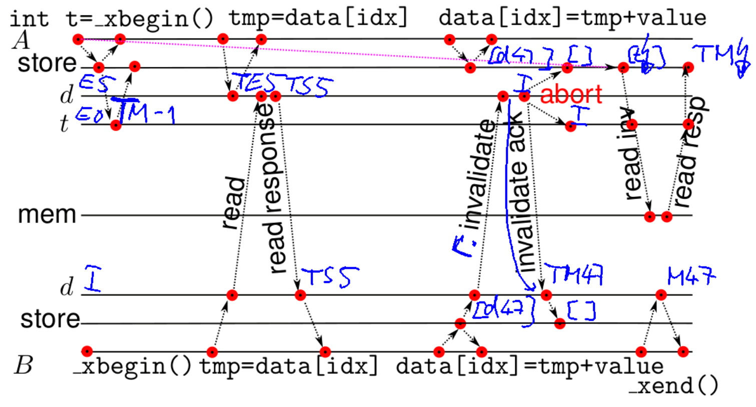
Thread A

```
int t = _xbegin();
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

Thread B

```
_xbegin();
int tmp = data[idx];
data[idx] = tmp+value;
_xend();
```

+2



Common Code Pattern for Mutexes

Using HTM in order to implement mutex:

```
int data[100]; // shared
int mutex;
void update(int idx, int val) {
  if(_xbegin()==_XBEGIN_STARTED)
  { if (!mutex>0) _xabort();
    data[idx] += val;
    _xend();
  } else {
    wait(mutex);
    data[idx] += val;
    signal(mutex);
  }
}
```

Common Code Pattern for Mutexes

Using HTM in order to implement mutex:

```
int data[100]; // shared
int mutex;
void update(int idx, int val) {
  if(_xbegin()==_XBEGIN_STARTED)
  { if (!mutex>0) _xabort();
    data[idx] += val;
    _xend();
  } else {
    wait(mutex);
    data[idx] += val;
    signal(mutex);
  }
}
```

```
void update(int idx, int val) {
  lock(&mutex);
  data[idx] += val;
  unlock(&mutex);
}
void lock(int* mutex) {
  if(_xbegin()==_XBEGIN_STARTED)
  { if (!*mutex>0) _xabort();
    else return;
  } wait(mutex);
}
void unlock(int* mutex) {
  if (!*mutex>0) signal(mutex);
  else _xend();
}
```

- critical section may be executed without taking the lock (the lock is *elided*)
- as soon as one thread conflicts, it aborts, takes the lock in the fallback path and thereby aborts all other transactions that have read `mutex`

Hardware Lock Elision

Hardware Lock Elision

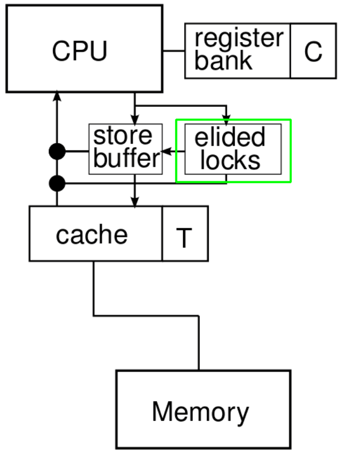
Observation: Using HTM to implement lock elision is a common pattern
$\rightsquigarrow$ provide special handling in hardware: HLE

- provides a way to execute a critical section without the need to immediately modify the cacheline in order to acquire and release the lock
- requires annotations:
  - instruction that increments the semaphore must be prefixed with `xacquire`
  - instruction setting the semaphore to 0 must be prefixed with `xrelease`
  - these prefixes are ignored on older platforms
- for a successful elision, all signal/wait operations of a lock must be annotated

Implementing Lock Elision

Transactional operation:

- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer



- `xacquire` of lock ensures *shared/exclusive* cache line state with *T*, issues `xbegin` and keeps the modified lock value in *elided lock* buffer
  - r/w access to other cache lines sets *T*
  - applying an *invalidate* message to a *T* cache line issues `xabort`, analogous for *read* message to a *TM* cache line
  - a *local CPU load* from the address of the elided lock accesses the buffer
- on `xrelease` on the same lock, decrement C and, if $C = 0$, clear *T* flags and elided locks buffer flush the store buffer

Transactional Memory: Summary

Transactional memory aims to provide `atomic` blocks for general code:

- frees the user from deciding how to lock data structures
- compositional way of communicating concurrently
- can be implemented using software (locks, atomic updates) or hardware

TM in Practice

Availability of TM Implementations:

- GCC can translate accesses in `__transaction_atomic` regions into `libitm` library calls
- the library `libitm` provides different TM implementations:
  1. On systems with TSX, it maps atomic blocks to HTM instructions
  2. On systems without TSX and for the fallback path, it resorts to STM
- C++20 standardizes `synchronized`/`atomic_XXX` blocks
- RTM support slowly introduced to OpenJDK Hotspot monitors

Outlook

Several other principles exist for concurrent programming:

1. non-blocking message passing (the actor model)
   - a program consists of actors that send messages
   - each actor has a queue of incoming messages
   - messages can be processed and new messages can be sent
   - special filtering of incoming messages
   - *example:* Erlang, many add-ons to existing languages
2. blocking message passing (CSP, π-calculus, join-calculus)
   - a process sends a message over a channel and blocks until the recipient accepts it
   - channels can be send over channels (π-calculus)
   - *examples:* Occam, Occam-π, Go
3. (immediate) priority ceiling
   - declare *processes* with priority and *resources* that each process may acquire
   - each resource has the maximum (ceiling) priority of all processes that may acquire it
   - a process' priority at run-time increases to the maximum of the priorities of held resources
   - the process with the maximum (run-time) priority executes

References

D. Dice, O. Shalev, and N. Shavit.
Transactional Locking II.
In *Distributed Coputing*, LNCS, pages 194–208. Springer, Sept. 2006.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.

Online resources on Intel HTM and GCC's STM:

1. `http://software.intel.com/en-us/blogs/2013/07/25/fun-with-intel-transactional-synchronization-extensions`
2. `http://www.realworldtech.com/haswell-tm/4/`
3. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf`