Script generated by TTT

Title: Petter: Programmiersprachen (20.11.2019)

Date: Wed Nov 20 12:21:28 CET 2019

Duration: 90:55 min

Pages: 17

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
  - ⇝ 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, ...

⇝ currently best to use TM only for memory; check if TM supports irrevocable transactions

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

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

Intel's TSX in Broadwell/Skylake microarchitecture (since 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

Intel provides two software interfaces to TM:

1. Restricted Transactional Memory (RTM)
2. Hardware Lock Elision (HLE)

Implementing RTM using the Cache (Intel)

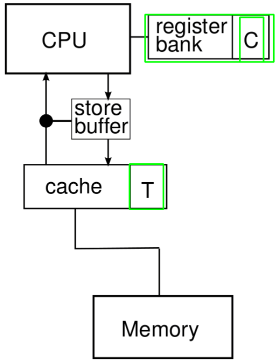
Supporting Transactional operations:

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

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



⇝ 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 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

The instruction `xbegin` is made accessible via library function `_xbegin()`:

_xbegin()

```
move   eax, 0xFFFFFFFF
xbegin _txnL1
_txnL1:
move   retval, eax
```

```
if(_xbegin()==_XBEGIN_STARTED) {
  // transaction code
  _xend();
} else {
  // non-transactional fall-back
}
```

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 ✓

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;
    }
}
```

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

The instruction `xbegin` is made accessible via library function `_xbegin()`:

_xbegin()

```
move   eax, 0xFFFFFFFF
xbegin _txnL1
_txnL1:
move   retval, eax
```

```
if(_xbegin()==_XBEGIN_STARTED) {
  // transaction code
  _xend();
} else {
  // non-transactional fall-back
}
```

⇝ user must provide *fall-back code*

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;
    }
}
```

⚠ Several problems:

- the fall-back code may execute racing itself
- the fall-back code is not isolated from the transaction

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) {
      if (!mutex>0) _xabort();
      data[idx] += value;
      _xend();
    } else {
      wait(mutex);
      data[idx] += value;
      signal(mutex);
    }
}
```

- the fall-back code does not execute racing itself ✓
- the fall-back code is now isolated from the transaction ✓

Happened Before Diagram for Transactions

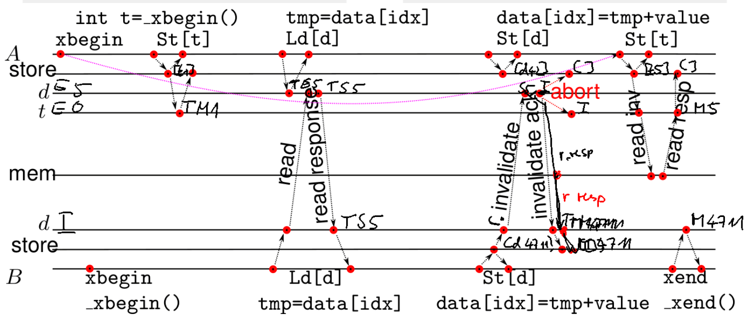
Augment MESI states with extra bit T. CPU A: d:E5 t:E0, CPU B: d:I, tmp/value registers

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();
```



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