Hardware Transactional Memory without the Hardware

Zach Bischof, Marcel Flores, Maciej Swiech

Northwestern University

June 7, 2012

Table of Contents

Transactions

Transactions are taken from the world of databases, where units of work are performed in isolation from the system, and can be aborted so that from the point of the view of the rest of the system, it looks like no changes ever happened.
A database transaction must be:

- Atomic
- Consistent
- Isolated
- Durable

Why Transactions?

- Locks are complicated, expensive, and ...

Why Transactions?

- Locks are complicated, expensive, and ... pessimistic

Why Transactions?

- Locks are complicated, expensive, and ... pessimistic
- Transactions are optimistic - very few (if any) memory locations in a locked region actually get altered by other processess, so locking them becomes an overhead that need not be paid.

Why Transactions?

- Locks are complicated, expensive, and ... pessimistic
- Transactions are optimistic - very few (if any) memory locations in a locked region actually get altered by other processess, so locking them becomes an overhead that need not be paid.
- Locks require a thread to wait for an entire region to be released, which does not scale well in increasingly parallel programming design.

Why Transactions?

- Locks are complicated, expensive, and ... pessimistic
- Transactions are optimistic - very few (if any) memory locations in a locked region actually get altered by other processess, so locking them becomes an overhead that need not be paid.
- Locks require a thread to wait for an entire region to be released, which does not scale well in increasingly parallel programming design.
- Transactions allow actions to go on until they necessarily would interfere or be interfered with, with a abort handler scenario.

Why Transactions?

So by using transactions we are afforded:

- Cheap critical sections of code

Why Transactions?

So by using transactions we are afforded:

- Cheap critical sections of code
- Freedom from having to manage tons of locks

Why Transactions?

So by using transactions we are afforded:

- Cheap critical sections of code
- Freedom from having to manage tons of locks
- Code that can operate as if no other threads were running

Intel Implementation

Definition

A **transaction** is a sequence of memory operations that are performed by a process such that, to all other processes, the operations seem to have been performed atomically. Furthermore, the operations must appear to the process performing the transaction as having happened one after another, with no interference from other processes

Intel Implementation (Haswell)

XBEGIN rel32

Starts off a transactional block, specifying a relative location to jump to in the case of an XABORT

Intel Implementation (Haswell)

XBEGIN rel32

Starts off a transactional block, specifying a relative location to jump to in the case of an XABORT

XABORT imm8

Aborts a transaction, placing the code in EAX and ignoring all changes made during the transaction

Intel Implementation (Haswell)

XBEGIN rel32

Starts off a transactional block, specifying a relative location to jump to in the case of an XABORT

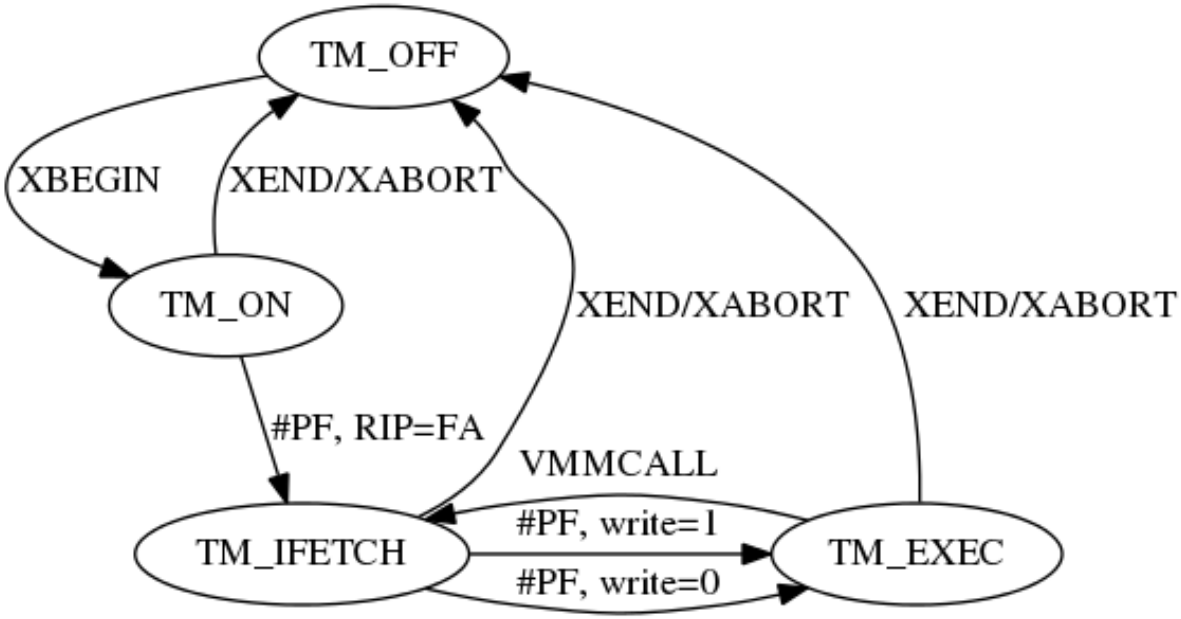
XABORT imm8

Aborts a transaction, placing the code in EAX and ignoring all changes made during the transaction

XEND

Ends a transational block, committing all changes made during the transaction

Palacios Implementation



The finite state machine we use in our implementation of Transactional Memory

XBEGIN

When the Host OS sees an XBEGIN, it throws an Undefined Opcode Exception, which we catch. We now enter the TM_ON portion of our FSM, and set up the following:

- Allocate necessary data structures to keep track of Guest OS state
- Blast away the VTLB

IFETCH

We come into the VMM having seen an ifetch Page Fault, and let the VTLB do its thing

- Store the next instruction and replace it with a hypercall

Data Operations

The next Page Fault we receive is a read or write, we need to handle it accordingly

- Check The TLB error flag to see if read or write
- Record it in the appropriate data structures

hypercall

When we see our hypercall, we know that the 'current' instruction has finished, we need to clean up and restore the next instruction

- Restore the stored next instruction
- Blast away the VTLB
- The cycle starts over

Abort

Either the code or a condition has signalled an abort

- Turn off TM state
- Restore the next instruction
- Free all of our data structures
- Change RIP to point to abort handler

XEND

The transactional block has come to a successful end, we need to commit the changes

- Turn off TM state
- Go through our data structures, copying entries into memory
- Free all of our data structures

Current Assumptions

- Instructions in transaction can be decoded
- Entire transaction will fit on a page
- All instructions occur on the same page
- Interrupts can be ignored
- Single core environment

Future Work

- Fix all assumptions
- Add Multicore Support
- Add Intel Support (?)

Future Work

- Fix all assumptions
- Add Multicore Support
- Add Intel Support (?)
- Optimizations...