Transactional Memory

- Transactional memory promises to substantially reduce difficulty in writing correct, efficient, and scalable concurrent programs

Problems with Transactional Memory

- Problems
  - Bounded* and Best-effort* HTM proposals impose unreasonable constraints on programmers
  - More flexible STM implementations are too slow
    - Too much overhead
Hybrid Transactional Memory (HyTM)

- An approach to implementing transactions in software that uses best-effort HTM to boost performance but doesn’t depend on HTM

- Goal:
  - Programmers can design & test transactional programs in existing systems today, and still enjoy benefits of HTM support when it becomes available in the future
  - Inspire processor designers to consider TM 🐻
Hybrid Transactional Memory

- Hybrid Transactional Memory (HyTM) System
  - Library
  - Compiler
    1. Transaction is attempted using best-effort HTM
    2. Retried using software if it fails

Background &
Other Relevant Definitions
Review

**Transactional memory**
- Supports code sections that are executed *atomically*
  - Appear to be executed one at a time with no interleaving between their steps
- Allows programmers to express what SHOULD be executed atomically not HOW it should be done
  - Reduces difficulty of writing correct concurrent programs
- Can significantly improve performance/scalability making them:
  - Easier to write, understand and maintain

---

**Transactional Memory**

- Review – Transactional Memory
  - Allows sections of code to be designated as transactional
  - Transaction commits:
    - Appears to have executed atomically
  - Transaction aborts:
    - Has no effect on shared state
  - Transactional section is attempted/retried until transaction commits
More on Bounded HTM

- Significant progress towards practical/sufficient STM
  - Hardware support for TM is desirable/beneficial

- Herlihy & Moss – Hardware Transactional Memory
  - Bounded-size atomic transactions that are short enough to be completed without context switching
    - Supported by simple additions to cache mechanisms of current processors
      - Programmers must be aware of specific limitations of HTM

Bounded & Best-effort HTM

- Herlihy & Moss (1993)
  - Fixed-size, fully-associative transactional cache
  - Exploits existing cache coherence protocols to enforce atomicity (up to the size of the transactional cache)
    - Larger transactions fail
    - Transactions that are interrupted by context switch fail
Bounded & Best-effort HTM

- Drawbacks of Bounded & Best-effort HTM
  - Transactions can only succeed if it fits in cache
    - Ability to commit depends on size and layout w.r.t. cache geometry
  - Doesn't guarantee to handle every transaction

Introduction – Unbounded HTM

- Overcoming disadvantages of bounded HTM using “unbounded” HTM
  - Allow transactions to commit even if they exceed on-chip resources or too long duration
  - Too complex and risky to be used in near future 😞
Unbounded STM

- **UTM – Ananian et al.**
  - Supports transactions that can survive context switches and whose size is only limited by virtual memory
  - Requires additional hardware support that seems too complicated for near future commercial processors 😞

Related Work – Moore et al.

- **Thread-Level TM (TTM)**
- **LogTM**
  - Stores tentative new values “in place”
  - Maintain logs to undo changes made by transaction (abort)
  - Transactions can’t survive context switches
Related Work – Hybrid Transactional Memory

- Kumar et al.
  - Use HTM to optimize DSTM of Herlihy
    - Specific HTM design
  - Key Differences:
    - Aims to optimize object-based DSTM
      - HyTM uses low-level, word-based TM
    - Requires new hardware support
      - HyTM doesn’t
    - No Support for preserving transactions across context switches

Introduction – Hybrid Transactional Memory

- HyTM
  - New approach to TM that works in existing systems but can boost performance and scalability using future hardware support
    1. Exploits HTM support if available
      - For transactions that don’t exceed HTM limitations
    2. Executes other transactions in software (STM)
  - Any transaction can be executed in software
    - HTM doesn’t need to be able to
      - Build best-effort HTM
        - Avoid risk/complexity of unbounded
Introduction - HyTM

- Built prototype compiler & STM library
- Compiler
  - Produces code for executing transactions in HTM/STM
- Key design challenge
  - Ensure that HTM conflicts & STM conflicts are detected/resolved appropriately
    - Augment HTM transactions with code to look up structures maintained by STM

Hybrid Transactional Memory - Presented by: Armand R. Burks - Fall 2008

Hybrid Transactional Memory
HyTM Prototype

The “Nitty Gritty”
HyTM Design Overview

- **Compiler & Library**
- Compiler produces 2 code paths for each transaction
  - One attempts to use HTM
  - The other uses STM by invoking calls to the library
- **HYTM_SECTION_BEGIN** - where hybrid transactional code section begins
- **HYTM_SECTION_END** - where it ends
  - Compiler translates code between to execute transactionally

HTM Interface

- Transaction starts with `txn_begin` and ends with `txn_end`
  - `txn_begin` specifies an address to branch to in case transaction aborts
- If transaction executes `txn_end` without aborting, continues past `txn_end`
  - Appeared to have executed atomically
- Otherwise branches off to the address specified
- Transaction aborts with `txn_abort`
**HTM Interface**

- First attempt each transaction in HTM
  - Expect most transactions to be able to complete in hardware
  - These are faster than software transactions

- If HTM fails, use library
  - Call method in HyTM library to decide whether to retry with HTM or STM
    - Method also implements contention control policies
  - Transaction that fails repeatedly should be attempted in STM

- All transparent to the programmer

**Augmenting Hardware Transactions**

- Augment hardware transactions to insure they interact correctly with transactions executed in software library

- Key observation:
  - Location’s logical value only differs from its physical value if a software transaction has modified it
    - If no such transaction exists, can apply transaction to location using HTM
    - HyTM augments HTM transactions to detect these conflicts with STM at orecs
      - Looks up orec of location accessed to detect conflict with STM
      - HTM aborts if orec changes before it commits
Augmenting Hardware Transactions

**Straightforward HTM**

```plaintext
txn_begin handler-addr
    tmp = X;
    Y = tmp + 5;
txn_end
```

**Augmented HTM (produced by compiler)**

```plaintext
txn_begin handler-addr
    if (!canHardwareRead(&X))
        txn_abort;
    tmp = X;
    if (!canHardwareWrite(&Y))
        txn_abort;
    Y = tmp + 5;
txn_end;
```

**HyTM Software Library**

```plaintext
bool canHardwareRead(a)
{
    return (OREC_TABLE[h(a)].o_mode != WRITE);
}

bool canHardwareWrite(a)
{
    return (OREC_TABLE[h(a)].o_mode == UNOWNED);
}
```

Check orec to see if the mode is not set to WRITE

Check orec to see if the mode is set to UNOWNED
HyTM Data Structures

- Two key data structures:
  - Transaction Descriptor
  - Ownership Record (orec)

HyTM Data Structures

- Transaction Descriptor
  - Prototype maintains only one for each thread

- Ownership record
  - Maintains table of oreces
  - Each location in memory maps to an orec in table
    - Multiple locations map to same orec
    - Uses hashing to do this
How are Software Transactions Implemented?

1. Transaction begins with empty read and write sets
2. Status is set to ACTIVE
3. Executes the user code (transactional section)
4. Make calls to STM library for each memory access
Writing in Software Transactions

- **Writing**
  - **Before writing a location:**
    - Transaction acquires exclusive ownership (WRITE mode) of orec for that location
    - Stores descriptor identifier (tdid) & version # in the orec
    - Subsequent writes (*by same transaction*) finds entry in orec & overwrites value
    - Creates an entry in write set to record new value
  - Example follows...

Transaction Descriptor

<table>
<thead>
<tr>
<th>Tdid: 0</th>
<th>ver/status: 27/ACTIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Set</td>
<td>orecIdx</td>
</tr>
<tr>
<td>Write Set</td>
<td>(0x100, 1)</td>
</tr>
</tbody>
</table>

Address space: 0x100

Orec Table

<table>
<thead>
<tr>
<th>tdid</th>
<th>ver</th>
<th>mode</th>
<th>rdcnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>27</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Hybrid Transactional Memory - Presented by: Armand R. Burks - Fall 2008
**Reading in Software Transactions**

- **Reading**
  - **Before reading a location:**
    - Transaction acquires ownership of orec (READ mode) for that location
    - Set mode field to READ and rdcnt to 1
    - If already owned in READ mode:
      - Increment rdcnt field
    - Records index of orec and snapshot of orec’s contents into write set
  - **After every read:**
    - Transaction validates entire read set
    - Ensure values read is consistent with values previously read

**Transaction Descriptor**

- Tdid: 0
- Ver/status: 27/ACTIVE

**Read Set**
- orecIdx: 0
- orecSnapshot: (0, 26, R, 1)

**Write Set**

**Orec Table**

<table>
<thead>
<tr>
<th>Address space</th>
<th>Orec Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>tid</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Hybrid Transactional Memory - Presented by: Armand R. Burks - Fall 2008
Validation in Software Transactions

- **Validation**
  - Determines that none of the locations its read have changed
    - Iterating over read set
    - Compare each orec owned in READ mode with snapshot
      - Nothing should change except possibly rdcnt field

Fast Read Validation

- **Avoids iterating over a transaction’s read set for validation**
  - Maintain a counter for # of times orec owned in READ mode is stolen by a transaction in WRITE mode
    - If counter hasn’t changed since last validation
      - All snapshots are still valid
    - If counter changes since last validation
      - Must iterate over entire read set
Committing Software Transactions

- Completing a transaction:
  - Transaction attempts to commit
    - Validates read set
      - If succeeds, atomically change status from ACTIVE to COMMITTED
    - Copies values in its write set back to appropriate memory locations
    - Releases ownership of those locations
      - Maintains exclusive ownership of locations until values are copied
        - Prevents others from viewing out-of-date values

Implementing Software Transactions

- Read after write
  - $T_o$ has write ownership of orec
  - $T_o$ needs to read from orec
    - $T_o$ searches write set to see if value was already stored in location
      - If not, get value from memory (virtual)
        - Logical value would only change by another transaction

- Write after read
  - $T_o$ has read ownership of orec
  - $T_o$ needs to write to location that maps to orec
    - Use snapshot to upgrade to WRITE mode
      - Has to insure another transaction doesn’t get WRITE ownership in between
    - Entry in read set is discarded – no longer in READ mode
Transaction Descriptor

Tdid: 0
ver/status: 27/APACTIVE

Read Set
orecldx oreclSnapshot

Write Set
(0x100, 1) (0x108, 12)

Orec Table

Address space

0x100
0x108

Orec Table

tdid ver mode rdcnt

0 27 W
1 1
2 ...
...
...

Other transactions

Read after Write:
Needs to read location 0x100

Write after Read:
Needs to write to location 0x100
Resolving Conflicts (Software Transactions)

- $T_0$ requires ownership of location
- $T_1$ already owns in WRITE mode

**Case 1:**
- $T_1$'s status is ABORTED
  - $T_1$ won’t be able to commit
  - $T_0$ can steal ownership from $T_1$

**Case 2:**
- $T_1$'s status is ACTIVE
  - $T_0$ can abort $T_1$ (change its status from ACTIVE to ABORTED)
  - $T_0$ can wait and give $T_1$ a chance to complete
    - Decision is left to the contention manager

**Examples follow...**
**CASE 1:**
Another transaction owns in WRITE mode but status is ABORTED

**CASE 3:**
- T₀’s status is COMMITTED
  - Not safe to steal because T₁ may not have copied values
  - T₀ waits for T₁ to release ownership of the location

- T₀ requires ownership of location (WRITE)
- T₁ already owns in READ mode
  - T₁ can steal by acquiring in WRITE mode
    - This may cause other read validations to fail
  - T₁ can wait and allow reads to complete
    - T₁ asks contention manager
Contention Management

- HyTM design allows interface for separable contention managers
  - Library uses interface to inform contention manager of various events
    - Asks advice for decisions in situations discussed earlier
  - HyTM uses the following:
    - Polka contention manager
    - Variant of Greedy manager
  - Managers are not optimized
    - Possible improvement of performance

Conclusions

- Hybrid Transactional Memory allows us to use transactional programs today, while taking advantage of hardware support of the future
- Software-only mode is much more scalable than using locking techniques
- HyTM needs BETTER contention management
- Processor designers should consider supporting Transactional programming
That's all Folks!