Virtualizing Transactional Memory

Ravi Rajwar[†] Maurice Herlihy[§] Konrad Lai[†]

[†]Intel Labs
[§]Brown University/Microsoft Research

ISCA-32

Wednesday, June 8, 2005

Problem

- Transactional memory
  - A promising concurrency abstraction
- Current hardware well suited
  1. Buffering (cache)
  2. Conflict detection (coherence)
  3. Atomic commit (in cache)
- Resource limitations fundamental
  - Space: cache, page faults
  - Time: context switches

Place significant roadblocks to acceptability of hardware transactional memory

Why are these limitations serious?

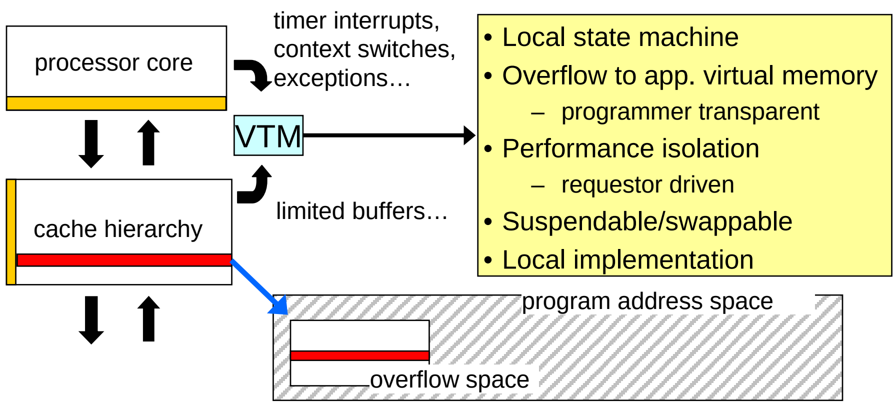
- Affects functionality, not just performance
  - Some transactions will never commit
- A fundamental limitation of space and time
  - More hardware only delays the inevitable
    - Non-scalable in a multi-programmed world
    - There will always be an n+1 case…
  - Time slice
    - Programmers have no control over time
    - Cannot determine when this would occur

Implementation artifacts must be functionally hidden from the user

3

Solution: Virtual TM

Seamless hardware/software integration



1. Common-case performance unaffected
2. No radical changes to the architecture

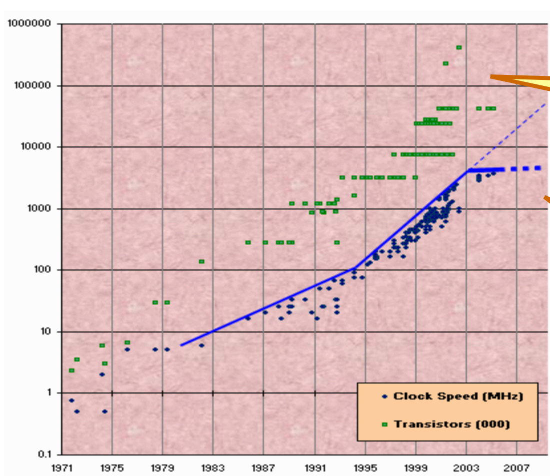
4

Outline

- Introduction
- Motivation for Transactional Memory
- Transactional Memory
- Virtualizing Transactional Memory
- Summary

5

Moore's law: not about clock



Transistor count still rising

Clock speed flattening sharply

http://www.gotw.ca/publications/concurrency-ddj.htm

Use transistors for concurrency through multi/many core

Concurrency models and locks

- Time no longer cures software
  - Cannot just wait 6 months for a faster processor
  - Must exploit increasing hardware concurrency
- Lock based concurrency popular
  - Common
  - Fundamentally limited
    - Performance
    - Software engineering

Locks rely on conventions

- Which locks protect which data
  - Software convention
  - 15% of Linux comments concern locking protocols
- Usage rules embedded in comments

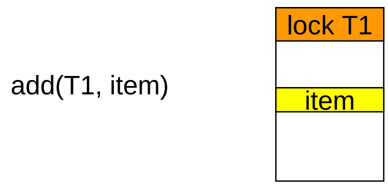
```
/*
* When a locked buffer is visible to the I/O layer BH_Launder
 * is set. This means before unlocking we must clear BH_Launder,
 * mb() on alpha and then clear BH_Lock, so no reader can see
* BH_Launder set on an unlocked buffer and then risk to deadlock.
 */
```

(ack: Brad Kuszmaul)

Expensive & dangerous to maintain code

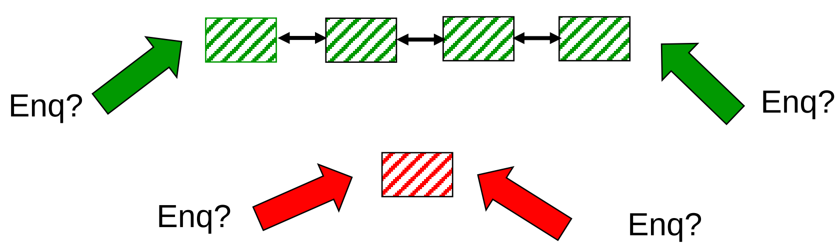
Locks do not compose

Consider thread-safe hash table modules



Exposing lock internals breaks abstraction

Locks are plain difficult



- Double ended queue (Herlihy's sadistic homework)
  – Concurrent updates to both ends when far apart
  – Interference ok if close but must not deadlock
- Appears very easy…

Locks are plain difficult

- Solution was publishable
  - Michael & Scott PODC 1996

Solving simple, easy-to-state synchronization problems should not be a publishable result

Need a working concurrency model

11

Outline

- Introduction
- Motivation for Transactional Memory
- Transactional Memory
- Virtualizing Transactional Memory
- Summary

12

What is Transactional Memory?

- Atomic sequence of reads, writes, computation
  1. All-or-nothing (failure atomicity)
  2. One-at-a-time order
  - Simpler than a database transaction (No durability)

```
start_transaction
    remove(T1, item)
    add(T2, item)
end_transaction
```

Atomic

- Perfect abstraction
  - Simple interface hiding a complex/subtle machinery
  - Usage model focused on "use"
    - Programmer declares intent, unconcerned with "how"

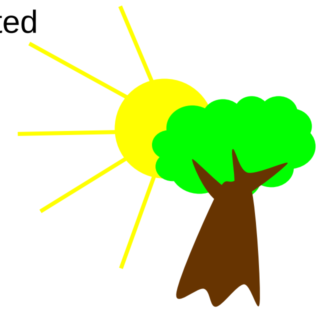
Benefits of Transactional Memory

- Eases writing correct concurrent programs
  - Composability and modularity
  - Non-blocking behavior
    - Gracefully deal with thread failures
  - Eliminates deadlocks, priority inversion, data races
  - Allows for sequential reasoning of programs
- Extracts high performance
  - Removes tension between lock granularity/concurrency
    - Automatically extract fine-grain locking behavior
  - Eliminates serialization limitations of locks

14

Implementing Transactional Memory

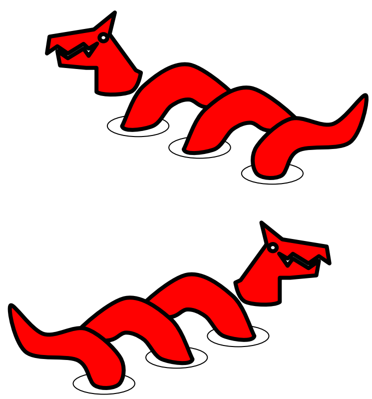
- All schemes basically
  1. Buffer updates within transaction
  2. Ensure no conflicts occur with other transactions
  3. Commit all updates instantaneously
- Hardware accelerations
  – Current hardware very well suited
    - Buffer: in cache
    - Conflicts: coherence
    - Commits: in cache



15

Fundamental hardware limitations

- Space
  – Cache evictions, collisions
- Time
  – pre-emption



Place significant roadblocks to acceptability of hardware transactional memory

16

Why is this serious?

- Impacts functional correctness
  - Today: limited resources ➔ performance degrades
  - With transactions
    - Some transactions will never commit…
- Throw more hardware at it
  - More caches, buffers,…
  - Non-scalable in a multi-programmed world
- What about time?

1. Always an n+1
2. De-scheduling happens

17

How about…

- Making the hardware limitation explicit?
  - Unrealistic for high-level languages
  - Don't know what a library call may do…
- Dual path coding?
  - Try in hardware, fall back to software
  - Unacceptable to write two versions
    - Worsens software engineering
    - Open question: can this be hidden in libraries?

Bottom line: Must virtualize limitation artifacts

Just like Virtual Memory virtualized Physical Memory

18

Minimum entry level for solution…

- High-performance hardware-only mode
  - Virtualization should not affect common case
- Performance isolation
  - Conflict detection, commits and unrelated threads
- Program isolation
  - Multi-programmed world, security, denial-of-service
- Transactional/Non-transactional interactions
  - Maintain atomicity at all times
- Must be transparent
  - Like virtual memory
- Must be implementable!

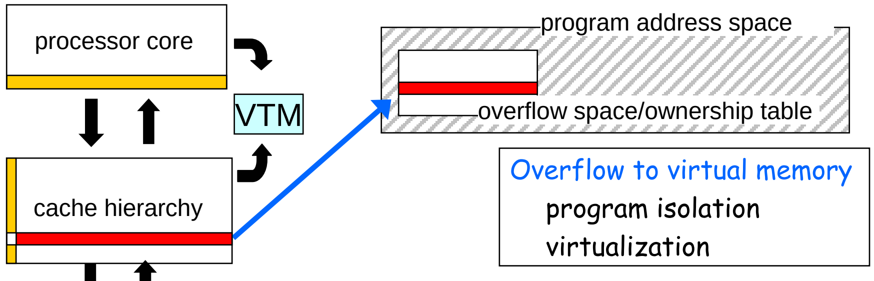
19

Outline

- Introduction
- Motivation for Transactional Memory
- Transactional Memory
- Virtualizing Transactional Memory
  - Key idea
  - Components
  - Working details
- Summary

20

VTM: key idea



When hardware resources exhausted:

Local state machine overflows to software space

Ownership "transfers" to shared software structure (XADT)

-- XADT controls access to this block

-- All accesses (all threads) to block gated through XADT

Make the common case fast (hardware)

Make the fast case common (Moore's law)

21

VTM: key components

1. Transaction Status Word (XSW)
2. Transaction Address/Data Table (XADT)
3. XADT Filter (XF)

XSW

Per-thread software structure

Determines transaction state

Compare&Swap operations

running → committed

running → aborted

XADT

Shared

Lives in virtual memory

Persistent across switches

Logical ownership/data table

Pointers to owning XSW

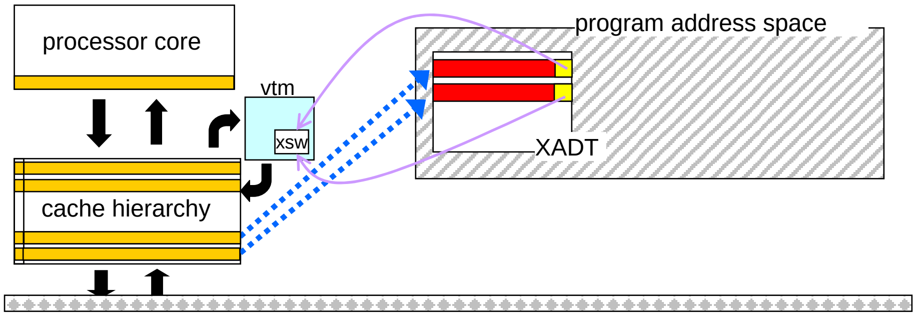
Stores overflowed data

Final arbiter of overflowed blocks

XF is a Bloom filter summarizing the XADT

22

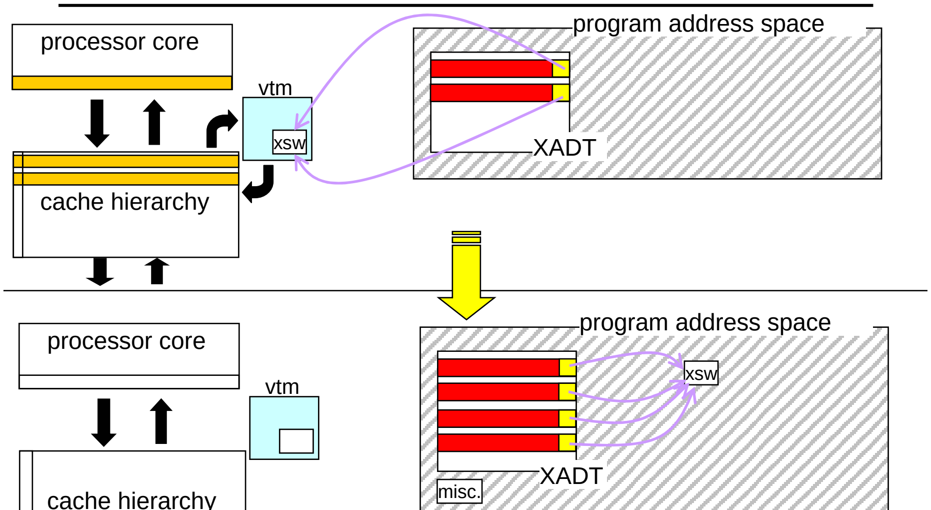
VTM: overflows



VTM machinery intercepts:
- Gets free entry in XADT
- Install pointer to XSW
- Move data and information to XADT
- XADT now owns these blocks
- Single XSW ties all blocks together

23

VTM: context switches



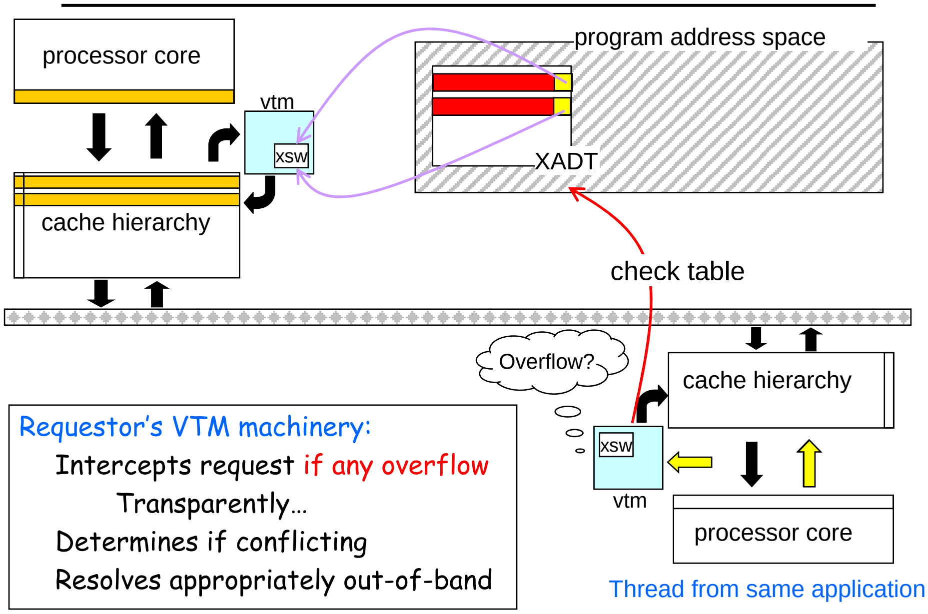
Persistent across time-slices
XADT guards access to all overflow data

VTM: conflict detection

- Conflict with cached blocks
  - Coherence protocol
  - Snooping processor detects conflicts immediately
- Conflict with overflowed blocks
  - Requires accessing the ownership table
    - Hardware no longer controls this block
  - Requestor detects this conflict before making request
    - Performance isolation
    - Design simplification

25

VTM: conflict detection

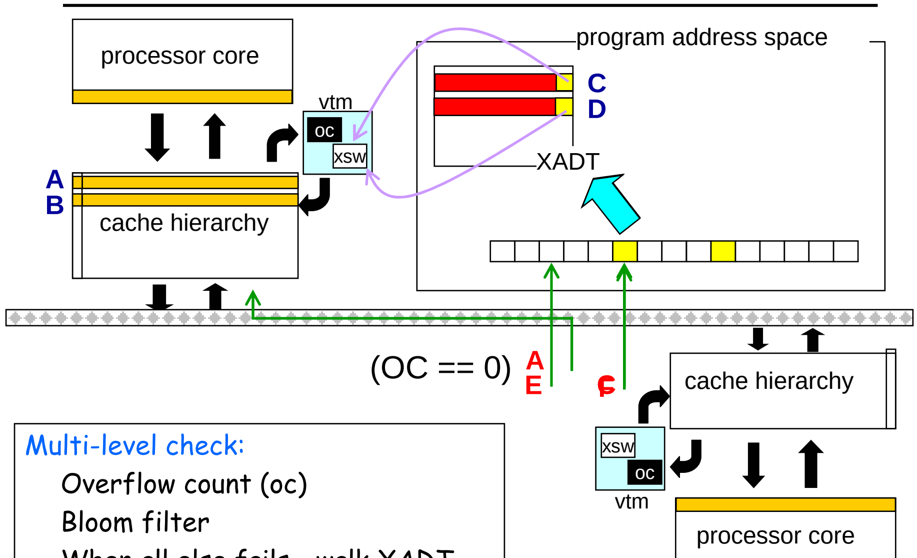


VTM: conflict detection

Overflows are rare by design
Conflict check must be done on every cache miss
Must ensure doesn't slow down hardware TM

27

VTM: conflict detection



Multi-level check:
- Overflow count (oc)
- Bloom filter
- When all else fails… walk XADT

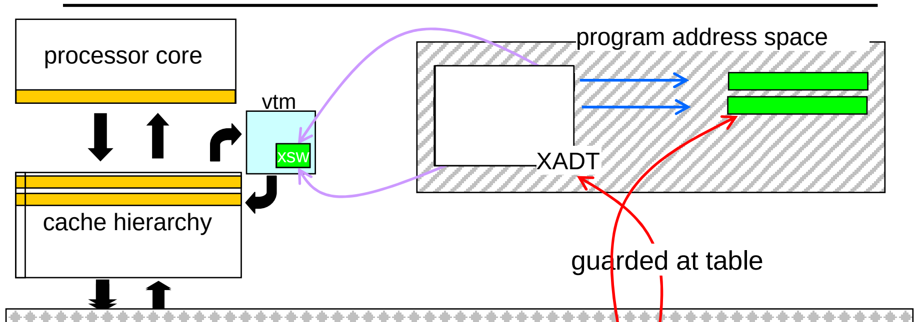
Thread from same application

VTM: conflict detection

- Overflow count
  - Move along, nothing to see
  - Typically
    - Zero, Locally-cached
    - Normal case unaffected
- Filter
  - Summarizes ownership table
  - Fast testing, possible false positives
  - Mostly clean and shared
  - Various implementations: arrays/hash/hybrids…
- Table Walker
  - When all else fails…

29

VTM: logical commits



To commit:

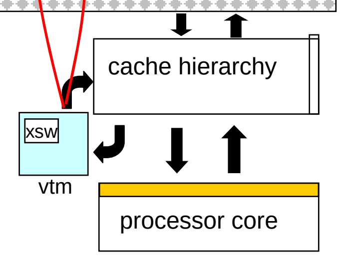
Set XSW running → committed (CAS)

This transaction now cannot abort

Commit hardware state

Update memory (incrementally)

Access protected at all times



Thread from same application

VTM: logical aborts

To abort:
- Aborter accesses XADT on possible conflict
- Detect conflict, resolves in its favor
- Set losing xaction XSW aborted (CAS)
- Losing transaction:
  - If active: detects abort right away
  - If swapped: detects abort on rescheduling

31

Did we meet the requirements? (1)

- High performance
  - → Virtualization doesn't impact hardware only mode…
  - → Overflow count, XF
- Program isolation
  - → Virtual memory
- Performance isolation
  - → Requestor does everything for overflows/conflicts
  - → No asynchronous events from outside
  - paper talks about an optional design, but bottom line:
    VTM allows requestor to make all overflow conflict detection decisions for both, transactional and non-transactional requests

32

Did we meet the requirements? (2)

- Transactional & non-transactional interactions
  - → Transactional data always guarded!
- Should be implementable
  - → All changes localized on processor…
  - → Requestor initiated actions simplify overall design

33

Summary

- Demonstrated transactional memory virtualization
  - Without radical changes to architecture
  - Without hurting common-case performance
- Ensure isolation
  - Performance
  - Program
- Open challenges
  - Many software usage model challenges
  - Software semantics
  - Operating systems interactions

34

BACKUP

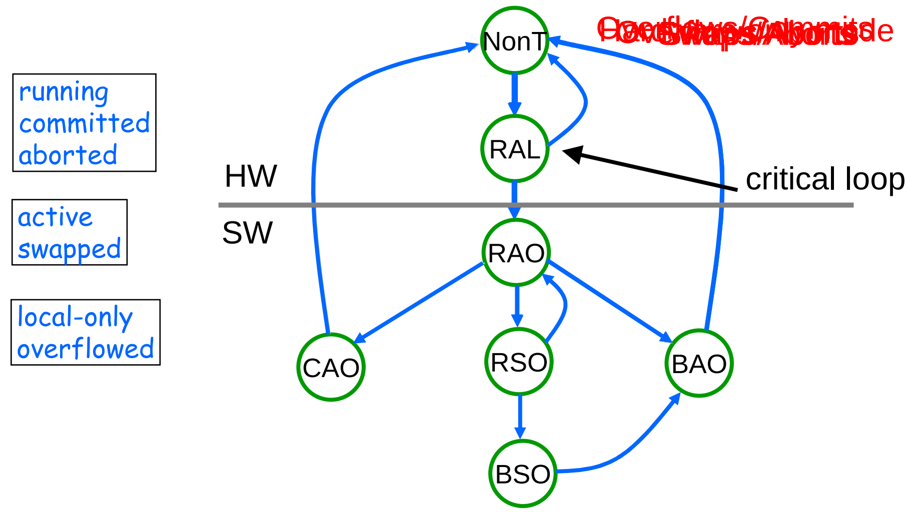
35

Hardware/Software communication

- Transfer of ownership needs coordination
- After XADT gains ownership
  - Send coherence invalidations
  - Informs all hardware transactional modes
  - Any cached copies invalidated
  - Remote processors, if required, re-access block
    - local VTM machinery intercepts this re-execution

36

Lifecycle of a transaction XSW



Transactions access other XSWs via the XADT

37