The Case for Hardware Support for Transactional Memory

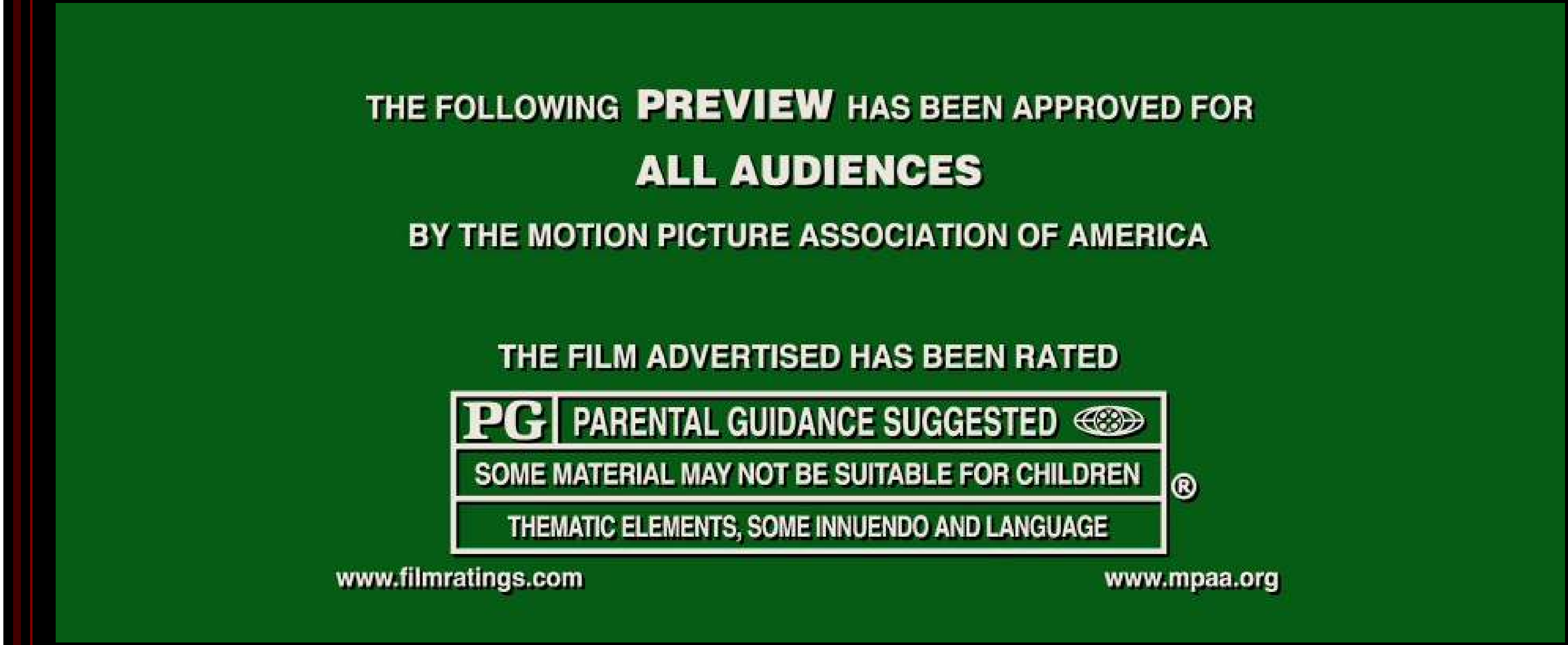
Christos Kozyrakis

Computer Systems Lab
Stanford University

http://csl.stanford.edu/~christos

MIT Seminar – February 2008

Coming Attractions…



THE FOLLOWING **PREVIEW** HAS BEEN APPROVED FOR

ALL AUDIENCES

BY THE MOTION PICTURE ASSOCIATION OF AMERICA

THE FILM ADVERTISED HAS BEEN RATED

PG PARENTAL GUIDANCE SUGGESTED

SOME MATERIAL MAY NOT BE SUITABLE FOR CHILDREN ®

THEMATIC ELEMENTS, SOME INNUENDO AND LANGUAGE

www.filmratings.com www.mpaa.org

Raksha: A Flexible Architecture for Software Security

- HW support for Dynamic Information Flow Tracking
  - Multiple, programmable policies; user-level exceptions
  - Full-system prototype: Sparc V8 core + Linux 2.6
- 1st DIFT system to
  - Detect low & high level attacks on unmodified binaries
    - From buffer overflows to SQL injections
  - Robust BOF detection by prohibiting pointer injection
    - Bypasses the problem of input validation within the program
  - Protects unmodified Linux kernel from BOF
    - Prevents buffer overflows and user/kernel pointer dereferences
    - No false positives
- The details at http://raksha.stanford.edu

The Case for Hardware Support for Transactional Memory

Christos Kozyrakis

Computer Systems Lab
Stanford University

http://csl.stanford.edu/~christos

The Parallel Programming Crisis

- Multi-core chips ⇒ inflection point for SW development
  - Scalable performance now requires parallel programming
- Parallel programming up until now
  - Limited to people with access to large parallel systems
  - Using low-level concurrency features in languages
    - Thin veneer over underlying hardware
  - Too cumbersome for mainstream software developers
    - Difficult to write, debug, maintain and even get some speedup
- We need better concurrency abstractions
  - Goal = easy to use + high performance
  - 90% of the speedup with 10% of the effort

The Difficulties with Parallel Programming

1. Finding independent tasks in the algorithm
2. Mapping tasks to execution units (e.g. threads)
3. Defining & implementing synchronization
   - Race conditions
   - Deadlock avoidance
   - Interactions with the memory model
4. Composing parallel tasks
5. Recovering from errors
6. Portable & predictable performance
7. Scalability
8. Locality management

- And, of course, all the sequential issues…

This Talk

1. Transactional Memory is a great, high-level construct for concurrency
2. Hardware support for Transactional Memory is necessary and practical
3. Transactional Memory hardware has beneficial uses beyond mutual exclusion

Transactional Memory (TM)

- Memory transaction [Knight'86, Herlihy & Moss'93]
  - An atomic & isolated sequence of memory accesses
  - Inspired by database transactions
- Atomicity (all or nothing)
  - At commit, all memory updates take effect at once
  - On abort, none of the memory updates appear to take effect
- Isolation
  - No other code can observe memory updates before commit
- Serializability
  - Transactions seem to commit in a single serial order

Programming with TM

```
void deposit(account, amount)
    atomic {
    synchronized(account) {
        int t = bank.get(account);
        t = t + amount;
        bank.put(account, t);
        return (1);
    }
```

```
void withdraw(account, amount)
    atomic {
    synchronized(account) {
        int t = bank.get(account);
        t = t - amount;
        if (t<0) return (0);
        bank.put(account, t);
        return (1);
    }
```

- Declarative synchronization
  - Programmers says what but not how
  - No explicit declaration or management of locks
- System implements synchronization
  - Typically with optimistic concurrency [Kung'81]
  - Slow down only on true conflicts (R-W or W-W)

Advantages of TM

- Easy to use synchronization construct
  - As easy to use as coarse-grain locks
  - Programmer declares, system implements
- Performs as well as fine-grain locks
  - Automatic read-read & fine-grain concurrency
  - No tradeoff between performance & correctness
- Failure atomicity & recovery
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart
- Composability
  - Safe & scalable composition of software modules

Composability: Locks

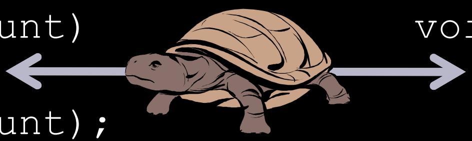
```
void transfer(A, B, amount)
   synchronized(A) {
   synchronized(B) {
        withdraw(A, amount);
        deposit(B, amount);
   }
   }
```

```
void transfer(B, A, amount)
   synchronized(B) {
   synchronized(A) {
        withdraw(B, amount);
        deposit(A, amount);
   }
   }
```

- Composing lock-based code is tough
    - Goal: hide intermediate state during transfer
    - Need global locking methodology now…
- Between the rock & the hard place
    - Fine-grain locking: can lead to deadlock

Composability: Locks

```
void transfer(A, B, amount)
   synchronized(bank){
        withdraw(A, amount);
        deposit(B, amount);
   }
```



```
void transfer(C, D, amount)
   synchronized(bank){
        withdraw(C, amount);
        deposit(A, amount);
   }
```

- Composing lock-based code is tough
  - Goal: hide intermediate state during transfer
  - Need global locking methodology now…
- Between the rock & the hard place
  - Fine-grain locking: can lead to deadlock
  - Coarse-grain locking: no concurrency

Composability: Transactions

```
void transfer(A, B, amount)
   atomic{
        withdraw(A, amount);
        deposit(B, amount);
   }
```

```
void transfer(B, A, amount)
   atomic{
        withdraw(B, amount);
        deposit(A, amount);
   }
```

- Transactions compose gracefully
  - Programmer declares global intend (atomic transfer)
    - No need to know of a global implementation strategy
  - Transaction in transfer subsumes those in withdraw & deposit
- System manages concurrency as well as possible
  - Serialization for transfer(A, B, x) & transfer(B, A, y)
  - Concurrency for transfer(A, B, x) & transfer(C, D, y)

Implementing Memory Transactions

- Data versioning for updated data
  - Manage new & old values for memory data
  - *Deferred updates* (write-buffer) vs *direct updates* (undo-log)

- Conflict detection for shared data
  - Detect R-W and W-W for concurrent transactions
  - Track the *read-set* and *write-set* of each transaction
  - Check during execution (*pessimistic*) or at the end (*optimistic*)

- Ideal implementation
  - Software only: works with current & future hardware
  - Flexible: can modify, enhance, or use in alternative manners
  - High performance: faster than sequential code & scalable
  - Correct: no incorrect or surprising execution results

Software Transactional Memory

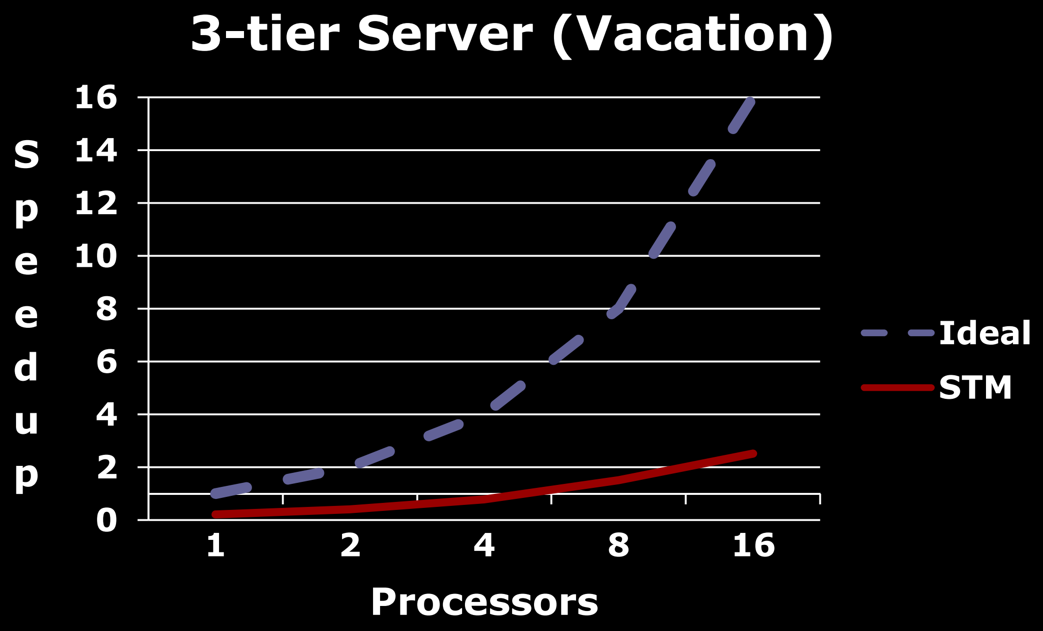
High-level → STM Compiler → Low-level

```
ListNode n;
atomic {
  n = head;
  if (n != NULL) {

    head = head.next;

  }
}
```

```
ListNode n;
STMstart();
  n = STMread(&head);
  if (n != NULL) {
    ListNode t;
    t = STMread(&head.next);
    STMwrite(&head, t);
  }
STMcommit();
```

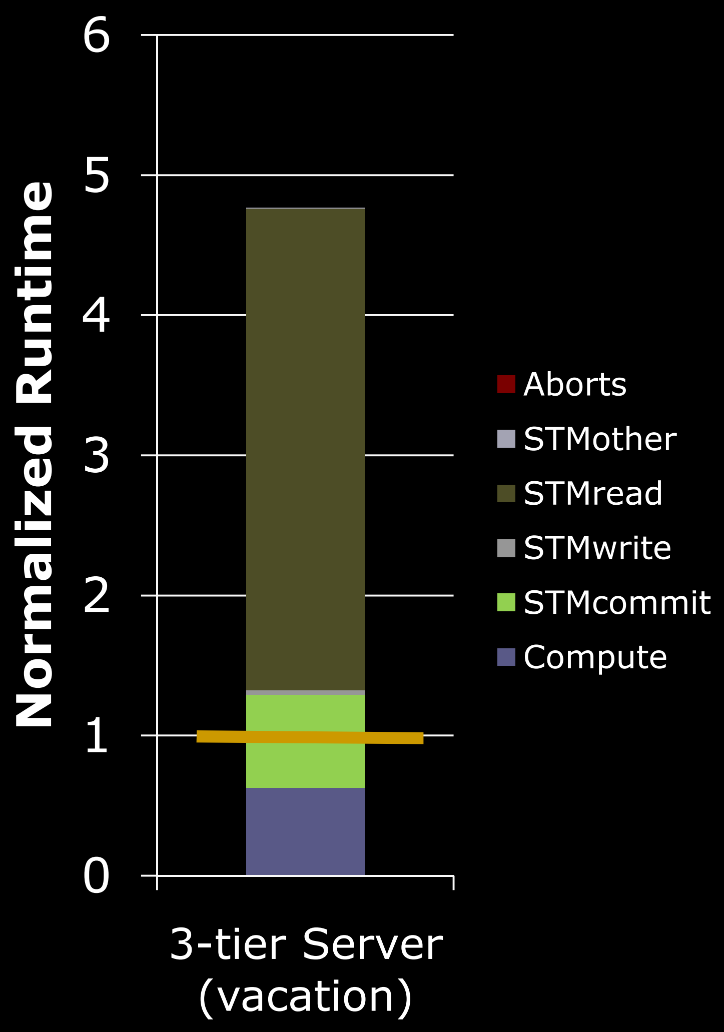
- Software barriers for TM bookkeeping
  - Versioning, read/write-set tracking, commit, …
  - Using locks, timestamps, object copying, …
- Can be optimized by compilers [Adl-Tabatabai'06, Harris'06]
- Requires function cloning or dynamic translation

STM Performance Challenges



- 2x to 8x overhead due to SW barriers
  - After compiler optimizations, inlining, …
- Short term: demotivates parallel programming
  - TM coding easier than locks but harder than sequential…
- Long term: energy wasteful

STM Runtime Breakdown



- STM challenges
  - Read barriers
    - Validate input data
    - Track read-set
  - Commit
    - Revalidate all input data
    - Detect conflicts
- Optimize away?
  - ≥1 barrier per object
  - Fine-grain concurrency ⇒ higher STM overheads
  - Frequent use of xactions ⇒ higher STM overhead

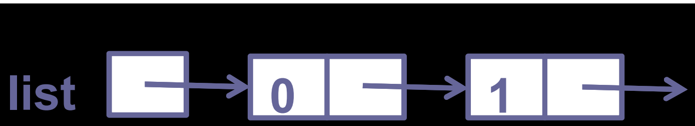
Is STM Correct?

Thread 1

```
atomic{
 if (list != NULL) {
      e = list;
      list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 != r2);
```

Thread 2

```
atomic{
  if (list != NULL) {
      p = list;
      p.x = 9;
}
```



- The privatization example
  - T1 removes a head; T2 increments head
  - Correctly synchronized code with locks
- Inconsistent results with all STMs
  - T1 assertion may fail from time to time

Weak Vs Strong Isolation

- STMs offer weak isolation
  - Xactions not isolated from non-xaction code
  - Privatization & publication become challenging
- Strong isolation is expensive in SW
  - Requires barriers in non-transaction code
  - Further performance losses
- Alternative: segregate transactional data
  - Error-prone if done by the programmer
  - Difficult if done by the compiler or runtime
  - Analogy: relaxed consistency models…

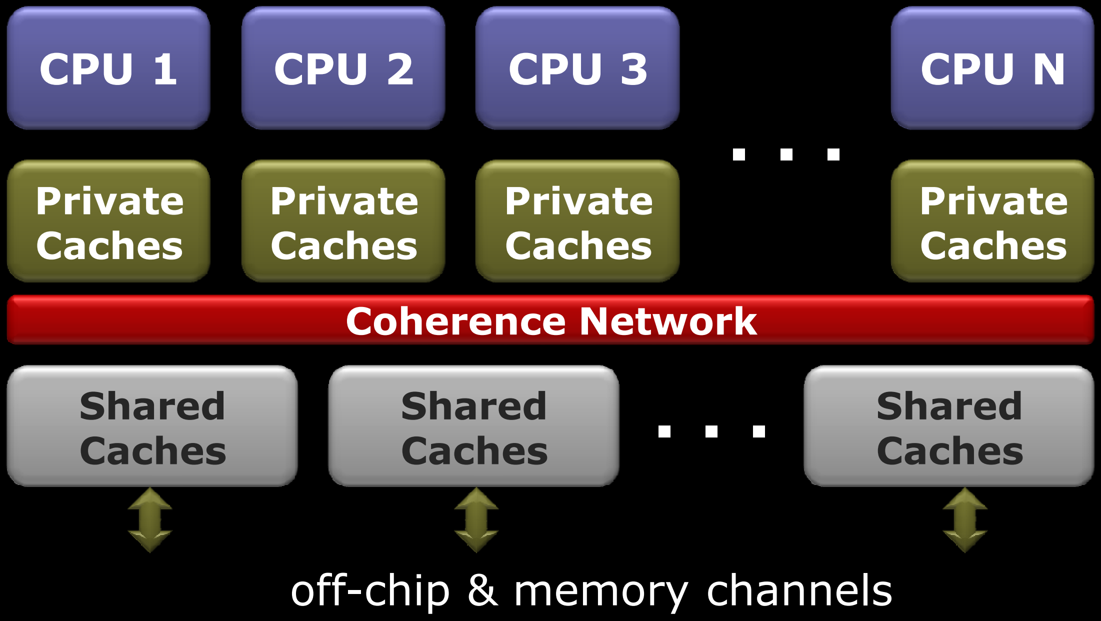
This Talk

1. Transactional Memory is a great, high-level construct for concurrency
2. Hardware support for Transactional Memory is necessary and practical
3. Transactional Memory hardware has beneficial uses beyond mutual exclusion

Hardware TM (HTM)

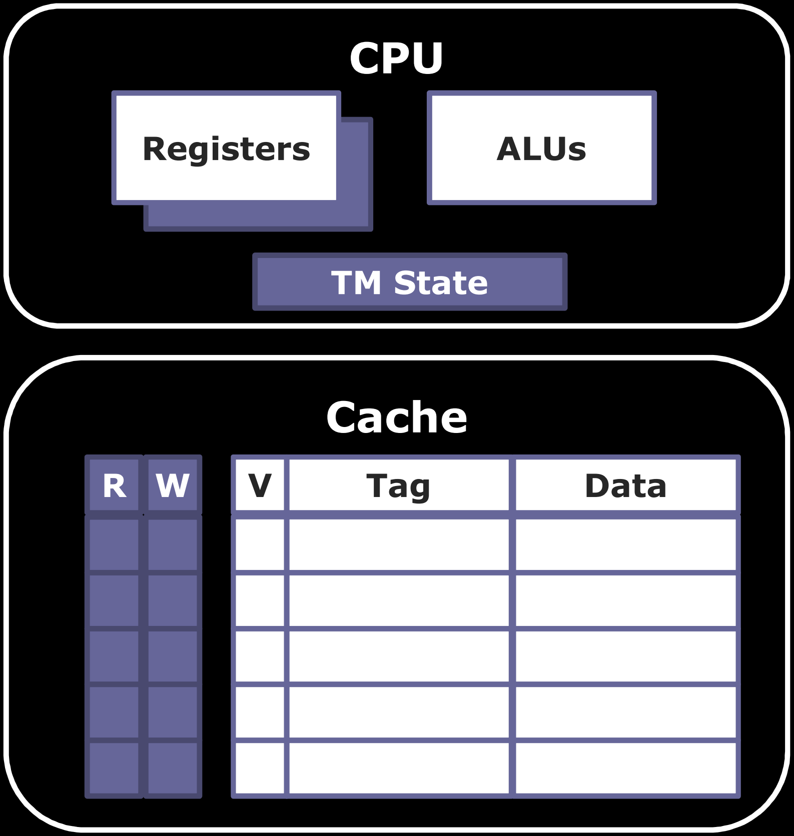
- HW support for common case TM behavior
  - Initial TMs used hardware [Knight’86, Herlihy & Moss’93]
  - All HTMs include software too…
- Rationale
  - HW can track all loads/stores transparently, w/o overhead
  - HW is good at fine-grain operations within a chip
  - We have transistors to spare in multi-core designs
    - Thanks to Moore’s law…
- Basic HW mechanisms
  - Cache metadata track read-set & write-set
  - Caches buffer deferred updates
  - Coherence protocol does conflict detection

Multi-core Chip



- HTM works with bus-based & scalable networks
- HTM works with private & shared caches

HTM Design

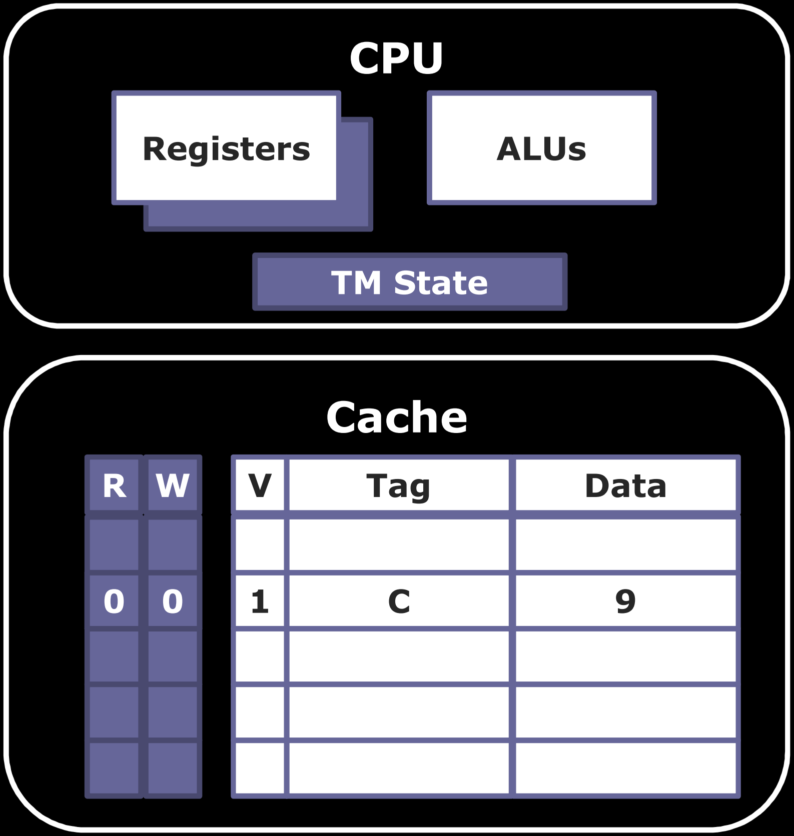


- The details in
  [ISCA’04, PACT’05, HPCA’07]

- CPU changes
  - Registers checkpoint (available in many CPUs)
  - TM state registers (status, pointers to handlers, …)

- Cache changes
  - R bit indicates membership to read-set
  - W bit indicates membership to write-set

HTM Transaction Execution

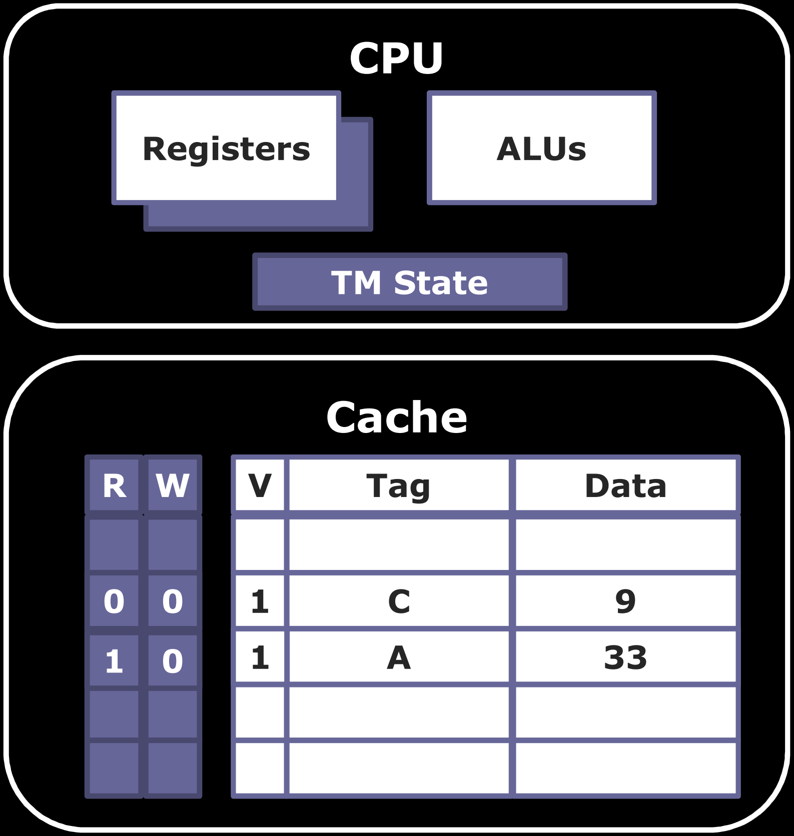


R	W	V	Tag	Data
0	0	1	C	9

```
Xbegin ⇦
    Load A
    Store B ⇐ 5
    Load C
Xcommit
```

- Transaction begin
  - Initialize CPU & cache state
  - Take register checkpoint

HTM Transaction Execution

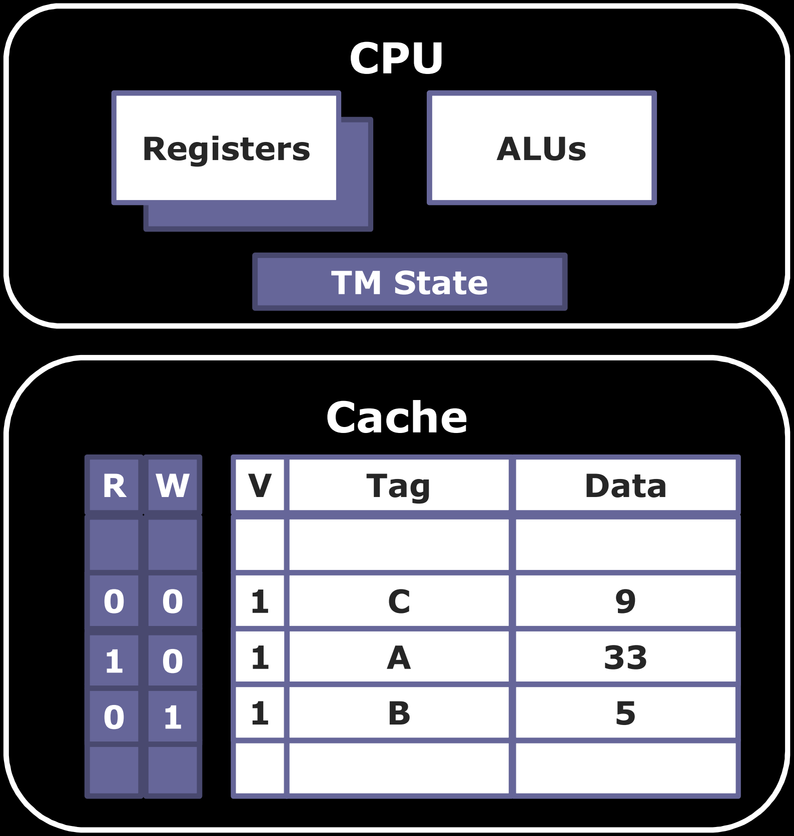


R	W	V	Tag	Data
0	0	1	C	9
1	0	1	A	33

```
Xbegin
    Load A ⇦
    Store B ⇐ 5
    Load C
Xcommit
```

- Load operation
  - Serve cache miss if needed
  - Mark data as part of read-set

HTM Transaction Execution



CPU

Registers

ALUs

TM State

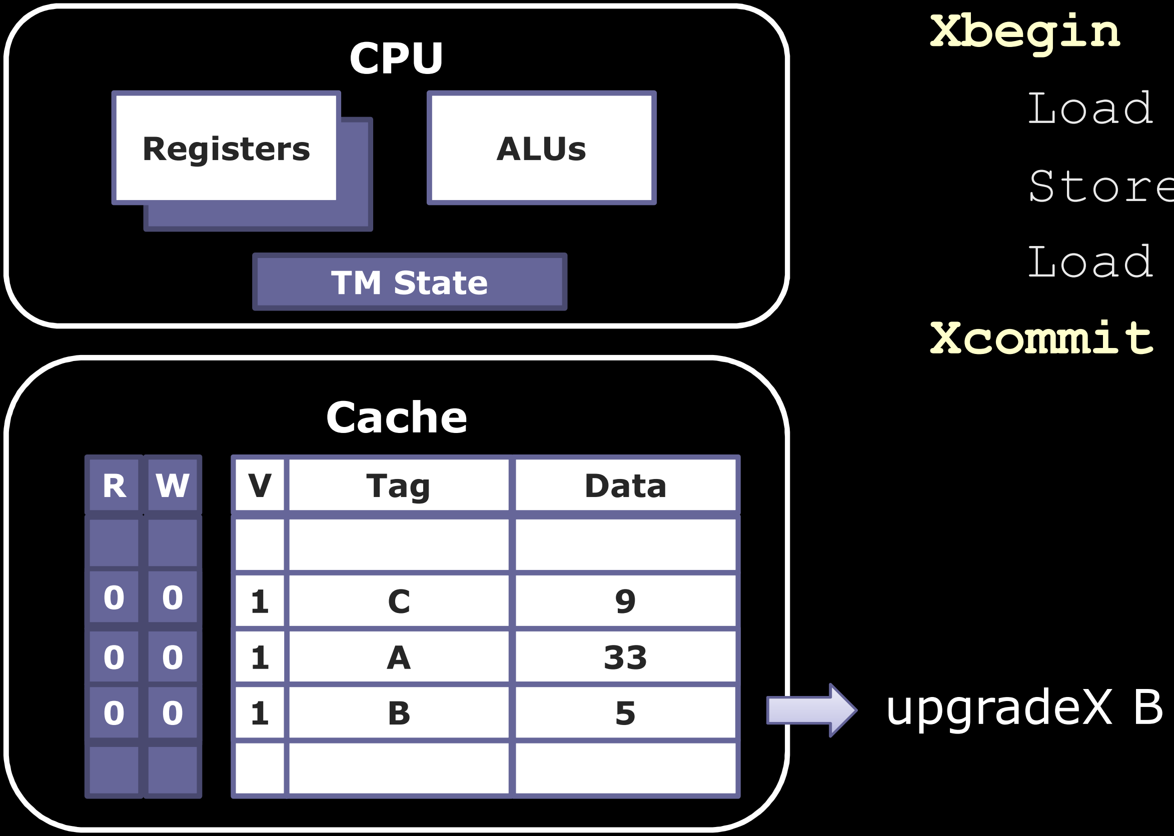
Cache

R	W	V	Tag	Data
0	0	1	C	9
1	0	1	A	33
0	1	1	B	5

```
Xbegin
   Load A
   Store B ⇐ 5
   Load C
Xcommit
```

- Store operation
  - Serve cache miss if needed (eXclusive if not shared, Shared otherwise)
  - Mark data as part of write-set

HTM Transaction Execution

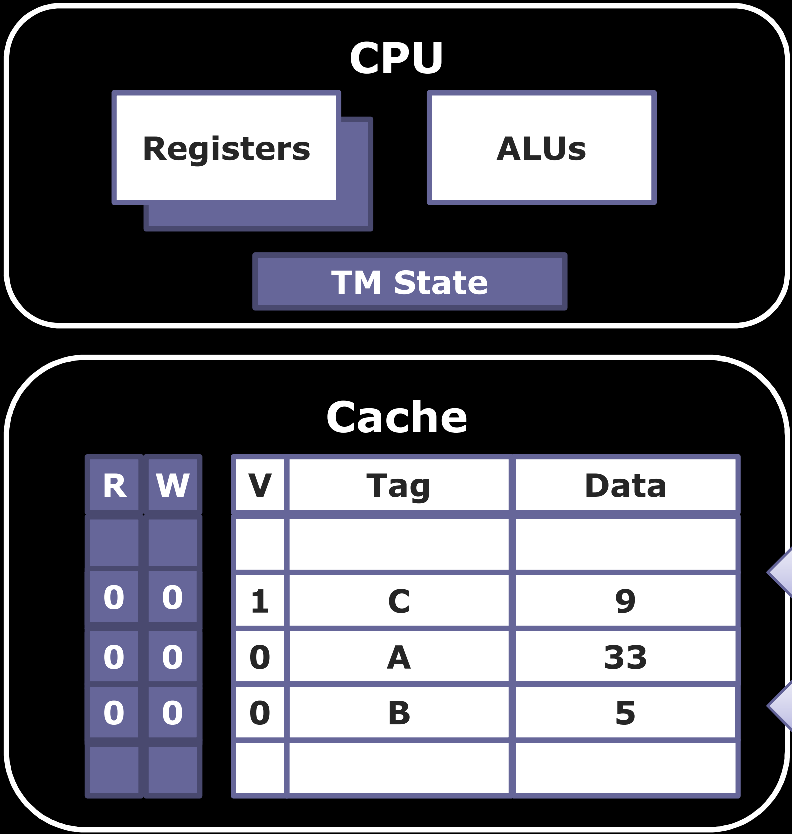


R	W	V	Tag	Data
0	0	1	C	9
0	0	1	A	33
0	0	1	B	5

```
Xbegin
    Load A
    Store B ⇐ 5
    Load C
Xcommit
```

- Fast, 2-phase commit
  - Validate: request exclusive access to write-set lines (if needed)
  - Commit: gang-reset R & W bits, turns write-set data to valid (dirty) data

HTM Conflict Detection



R	W	V	Tag	Data
0	0	1	C	9
0	0	0	A	33
0	0	0	B	5

```
Xbegin
    Load A
    Store B ⇐ 5
    Load C ⇐
Xcommit
```

upgradeX D ☑

upgradeX A ☒

- Fast conflict detection & abort
  - Check: lookup exclusive requests in the read-set and write-set
  - Abort: invalidate write-set, gang-reset R and W bits, restore checkpoint

HTM Advantages

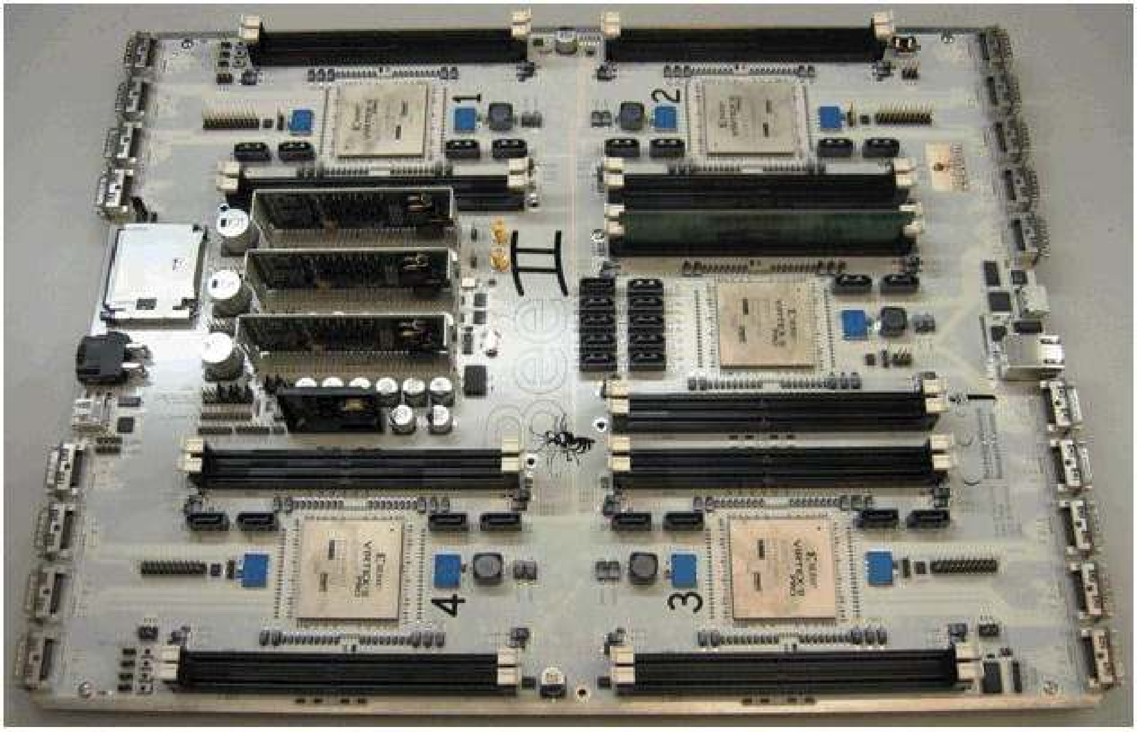
- Transparent
  - No need for barriers, function cloning, DBT, …
- Fast common case
  - Zero-overhead tracking of read-set & write-set
  - Zero-overhead versioning
  - Continuous validation of read-set
  - Fast commit without data movement
  - Fast abort without data movement
- Strong isolation
  - Conflicts detected on non-xaction stores too
  - Xactions isolated from non-xaction code
- Can simplify multi-core hardware [ISCA'04]
  - Replace existing coherence with transactional coherence

Questions about HTM

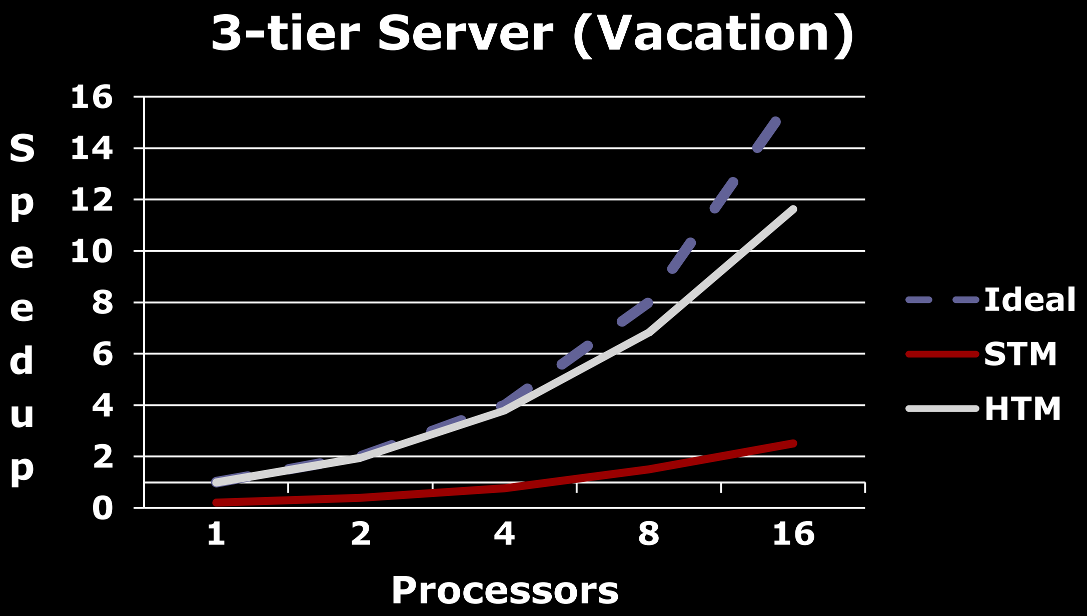
- Can it be built?
- Is it fast enough?
- Can you reduce the HW cost?
- Is the HW virtualizable?
- Is the HW flexible enough?

ATLAS HTM Prototype [DATE'07,FPGA'07]

- 9-way CMP with HTM support on the BEE2 board
  - Full-system prototype: PowerPC cores + Linux 2.6
  - 100MHz but still ≥100x faster than software simulator
  - OpenMP+TM, deterministic replay, perf. tuning tools, …

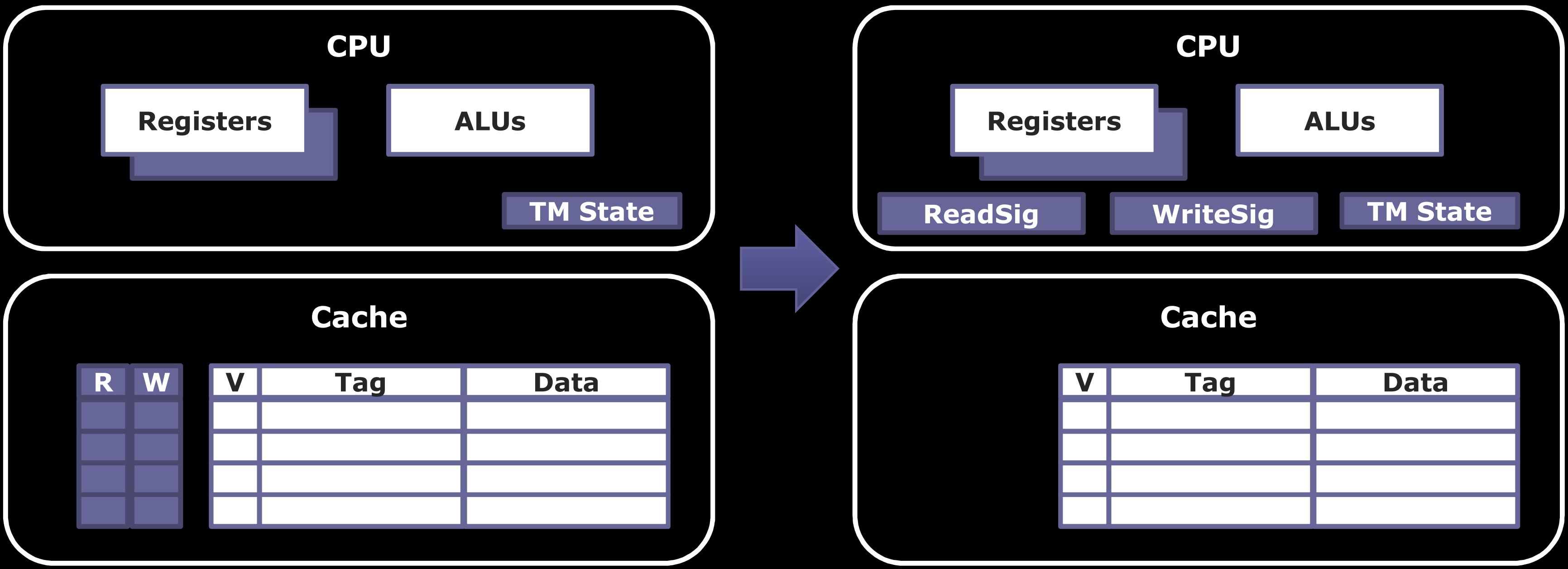


HTM Performance



- 2x to 7x over STM performance [ISCA'07]
  - Within 10% of sequential for one thread
  - Scales efficiently with number of processors
  - Uncommon cases not a performance challenge

Reducing HTM Cost with Signatures



- Signatures track read-set & write-set [ISCA’07]
  - Signatures = hardware Bloom filters
  - Conflict detection by HW, versioning by SW
- Pros: cost effective; easy to manipulate; flexible placement
- Cons: 2x performance penalty, false conflicts

Virtualizing HTM Capacity

- Problem: loss of metadata on cache evictions
  - Programs should be limited by HW details

- Common evictions: associativity misses
  - Eliminated with a simple victim cache [PACT’05]

- Uncommon evictions: capacity misses
  - Evict metadata to virtual memory [Rajwar’05]
    - Signatures to avoid VM search for conflict detection
    - Supported by HW [Chuang’06] or SW [ASPLOS’06]
  - Conceptually similar to VM paging
    - Functionally correct, but slower if used often

Virtualizing HTM Time

- Problem: interrupting a transaction
  - Interrupts, context switches, …
- Rethink scheduling for multi-core [ASPLOS’06]
  - Status-aware interrupt assignment
    - Prefer CPUs that don’t run transactions
    - Defer interrupt until a CPU commits its transaction
    - If needed, abort a young transaction & use its CPU
    - Use space virtualization when all else fails (rare)
  - Similar approach needed for locality management

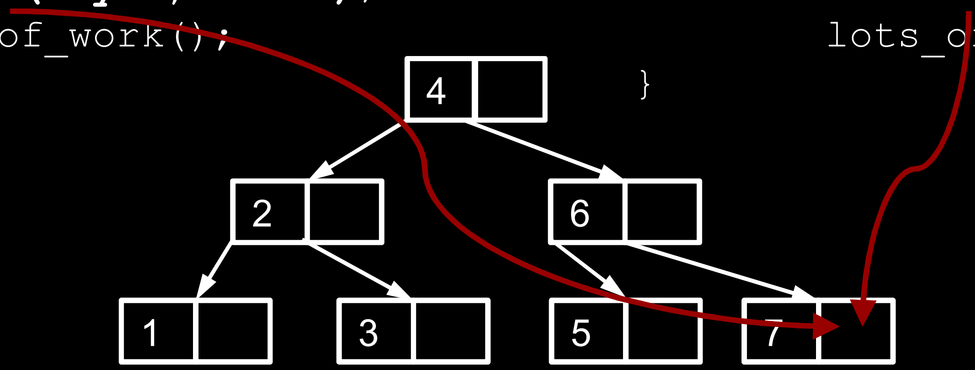
A Flexible Interface for HTM

- How does SW control an HTM?
  - To support common & new SW features
- Features for flexible HTM [ISCA’06]
  - Architecturally visible 2-phase commit
  - Support for transactional handlers
  - Support for nested transactions
  - Instructions for private or idempotent accesses
- Implementation notes
  - HW: metadata support for nested transactions
  - SW: transaction begin/end similar to function call/return
  - SW: transactional handlers similar to user-level exceptions

Example: Semantic Concurrency Control

```
Thread 1:
atomic{
        lots_of_work();
        insert(key=8, data1);
        lots_of_work();
}
```

```
Thread 2:
atomic{
        lots_of_work();
        insert(key=9, data2);
        lots_of_work();
}
```



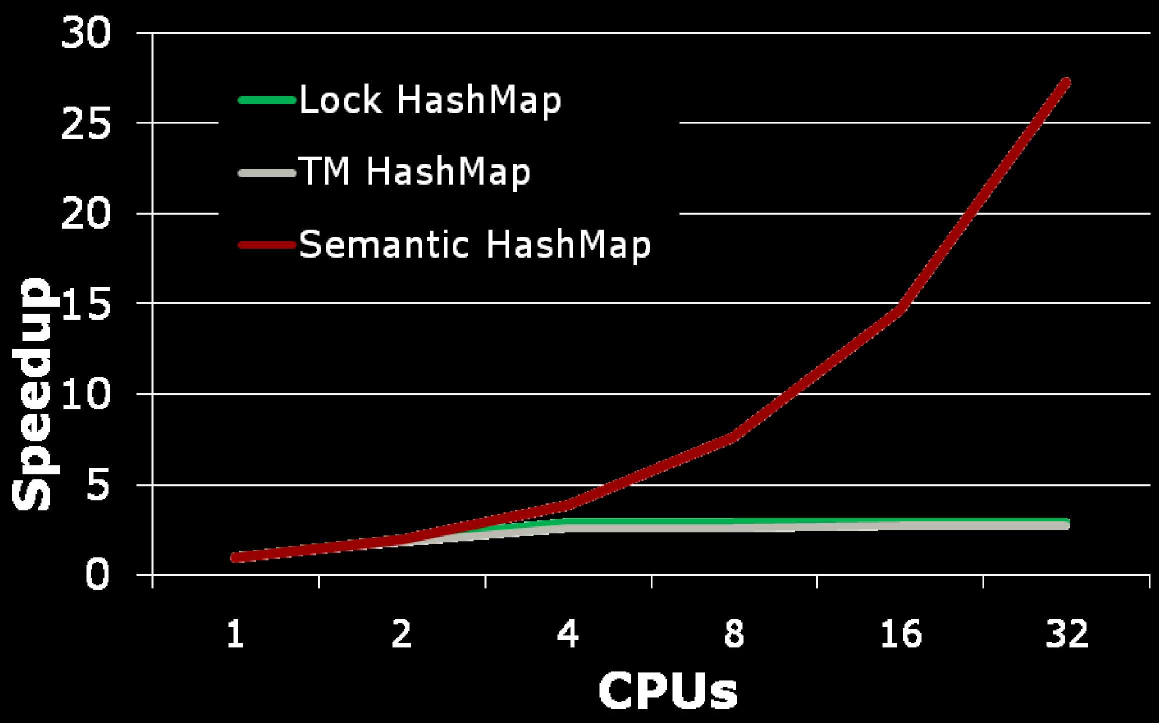
- Is there a conflict?
  - TM: yes, W-W conflict on a memory location
  - App logic: no, operation on different keys
- Common performance loss in TM programs
  - Large, compound transactions

Example: Semantic Concurrency Control

- Semantic concurrency in Atomos [PLDI’06]
  - From memory to semantic dependencies
  - Similar to multi-level transactions from DBs

- Transactional collection classes [PPoPP’06]
  - Read ops acquire semantic dependency
    - Using open nested transactions
  - Write ops deferred until commit
    - Using open nested transactions
  - Commit handler checks for semantic conflicts
  - Commit handler performs write ops
  - Commit/abort handlers clear dependencies

Example: Semantic Concurrency Control



- TestCompound
  - Long transaction with 2 map ops
  - Semantic concurrency ⇒ scalable performance

This Talk

1. Transactional Memory is a great, high-level construct for concurrency

2. Hardware support for Transactional Memory is necessary and practical

3. Transactional Memory hardware has beneficial uses beyond mutual exclusion

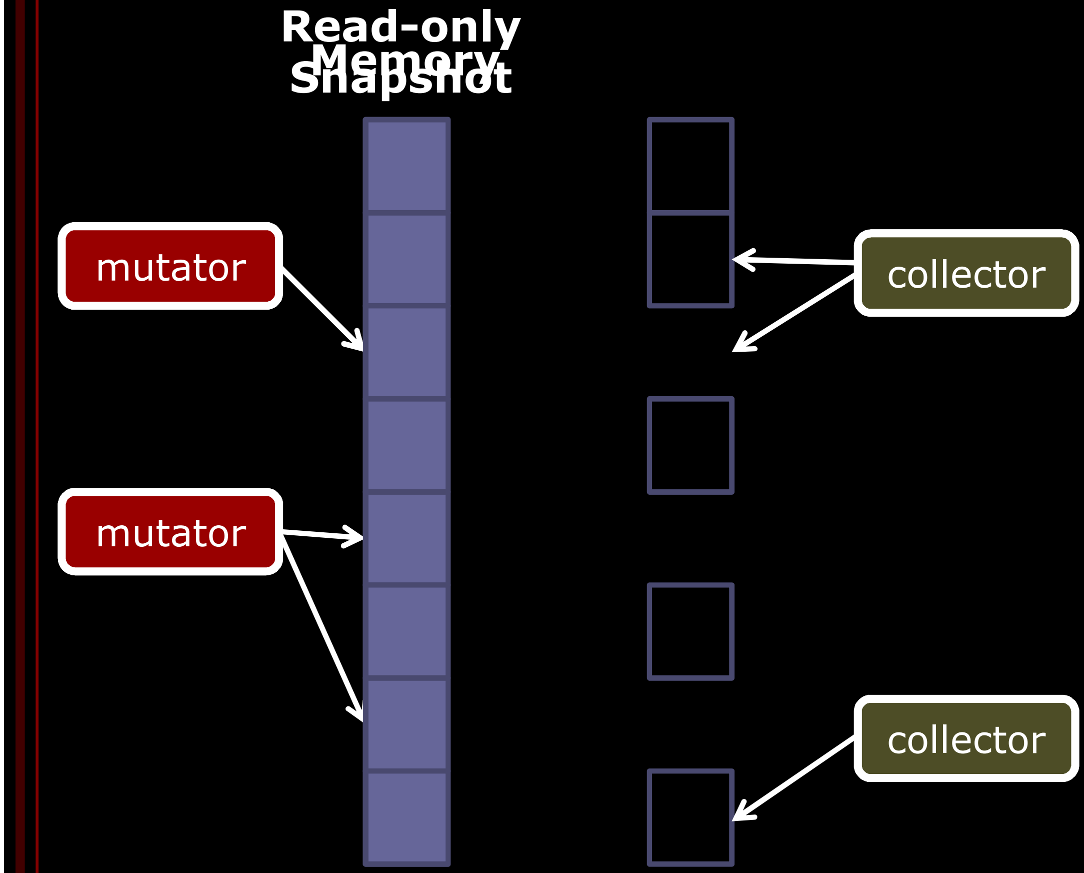
Beyond Concurrency Control

- TM hardware consists of
  - Memory versioning HW
  - Fine-grain access tracking HW
  - HW to enforcing ordering
  - Fast exception handlers
- Can use such HW for other purposes
  - Security, fault-tolerance, debugging, performance tuning, …
- The benefits for SW
  - Finer granularity (compared to VM-based approach)
  - User-level handling (compared to VM-based approach)
  - No instrumentation overhead (compared to DBT-based approach)
  - Simplified code
  - Automatic handling of interactions with other programs/tools

Applying TM Hardware

- Availability
  - Global & local checkpoints (versioning, order)
- Security
  - Fine-grain read/write barriers (tracking)
  - Isolated execution (versioning)
  - Thread-safe dynamic binary translation (all) [HPCA’08]
- Debugging
  - Deterministic replay (order)
  - Parallel step-back (versioning)
  - Infinite, fast watchpoints (tracking)
  - Atomicity violation detectors (tracking, order)
  - Performance tuning tools (tracking)
- Snapshot-based services (versioning)
  - Concurrent garbage collector
  - Dynamic memory profiler
  - User-level copy-on-write

Memory Snapshot



- Snapshot
  - Read-only image
  - Multiple regions
  - Access by ≥ 1 threads
- Snapshot GC
  - Stop-the-word⇒ concurrent
    - + 100 lines of code
    - For Boehm GC
  - No SW barriers or synch in mutator or collector code

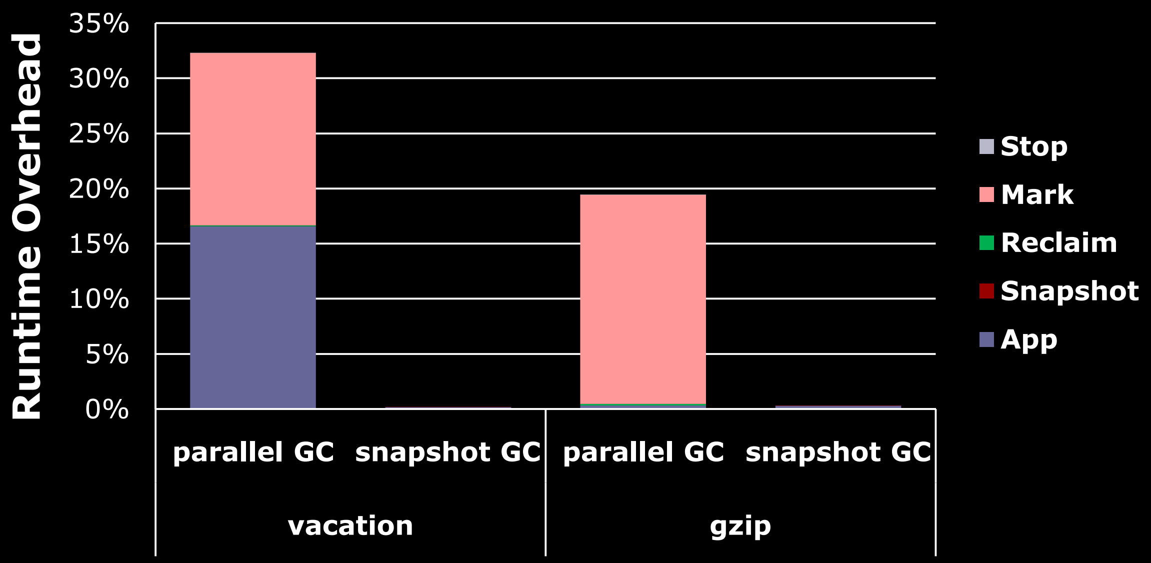
TM Hardware ⇒ Snapshot

- Feature correspondence
  - TM metadata ⇒ track data written since or read from snapshot
  - TM versioning ⇒ storage for progressive snapshot
    - Including virtualization mechanism
  - TM conflict detection ⇒ catch errors
    - Writes to read-only snapshot

- Differences & additions
  - Single-thread Vs. multithread versioning
  - Table to describe snapshot regions

- Resulting snapshot system
  - Scan (create) snapshot in O(# CPUs)
  - Update (write) and read in O(1)
  - Memory overhead up to O(# memory locations written)

GC Overhead



- Parallel GC ⇒ noticeable overhead
  - Stop app & use all available CPUs for GC
- Snapshot GC ⇒ GC is essentially free
  - Stop app, take snapshot, the run GC & app concurrently
  - App runs in parallel using TM; GC uses one core only
- Snapshot GC ⇒ simple (+100 lines over base GC)

Conclusions

1. Transactional Memory is a great, high-level construct for concurrency

2. Hardware support for Transactional Memory is necessary and practical

3. Transactional Memory hardware has beneficial uses beyond mutual exclusion

Questions?

- The Stanford TCC group
  - Faculty: C. Kozyrakis, K. Olukotun
  - Students: W. Baek, N. Bronson, C. Cao Minh, B. Carlstrom, J. Casper, J. Chung, A. McDonald, N. Njoroge, T. Oguntebi, S. Wee
- More info, papers, tutorials, tools at:
  - http://tcc.stanford.edu
  - http://csl.stanford.edu/~christos