Transactional Memory – Implementation
Lecture 1

COS597C, Fall 2010
Princeton University

Arun Raman

Module Outline

Lecture 1 (THIS LECTURE)

Transactional Memory System Taxonomy

Software Transactional Memory

Hardware Accelerated STMs

Lecture 2 (Thursday)

Hardware based TMs

Hybrid TMs

When are transactions a good (bad) idea

Assignment

Compare lock-based and transaction-based implementations of a concurrent data structure

Resources

[0] TL2 and STAMP Benchmark Suite --- For ASSIGNMENT
http://stamp.stanford.edu/

[1] Transactional Memory, Concepts, Implementations, and Opportunities - Christos Kozyrakis
http://csl.stanford.edu/~christos/publications/2008.tm_tutorial.acaces.pdf

[2] Transactional Memory - Larus and Rajwar
Chapter 4: Hardware Transactional Memory

Lot of the material in this lecture is sourced from here

[3] McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime
http://portal.acm.org/ft_gateway.cfm?id=1123001&type=pdf&CFID=109418483&CFTOKEN=70706278

[4] Architectural Support for Software Transactional Memory (Hardware Accelerated STM) - Saha et al.
http://www.computer.org/portal/web/csdl/doi/10.1109/MICRO.2006.9

[5] Signature-Accelerated STM (SigTM)
http://portal.acm.org/citation.cfm?id=1250673

[6] Case Study: Early Experience with a Commercial Hardware Transactional Memory Implementation [ASPLOS '09]
http://delivery.acm.org/10.1145/1510000/1508263/p157-dice.pdf?key1=1508263&key2=2655478821&coll=GUIDE&dl=GUIDE&CFID=111387689&CFTOKEN=28498606

[7] The Transactional Memory / Garbage Collection Analogy - Dan Grossman
http://portal.acm.org/ft_gateway.cfm?id=1297080&type=pdf&coll=GUIDE&dl=GUIDE&CFID=111387606&CFTOKEN=67012128

[8] Subtleties of Transactional Memory Atomicity Semantics - Blundell et al.
http://www.cis.upenn.edu/acg/papers/cal06_atomic_semantics.pdf

Lecture Outline

1. Recap of Transactions
2. Transactional Memory System Taxonomy
3. Software Transactional Memory
4. Hardware Accelerated STM

Lecture Outline

1. Recap of Transactions
2. Transactional Memory System Taxonomy
3. Software Transactional Memory
4. Hardware Accelerated STMs

Parallel Programming

nd independent tasks in the algorithm
ap tasks to execution units (e.g. threads)
efine and implement synchronization among tasks
void races and deadlocks, address memory
 model issues, …

4. Compose parallel tasks
5. Recover from errors
6. Ensure scalability
7. Manage locality
8. ….

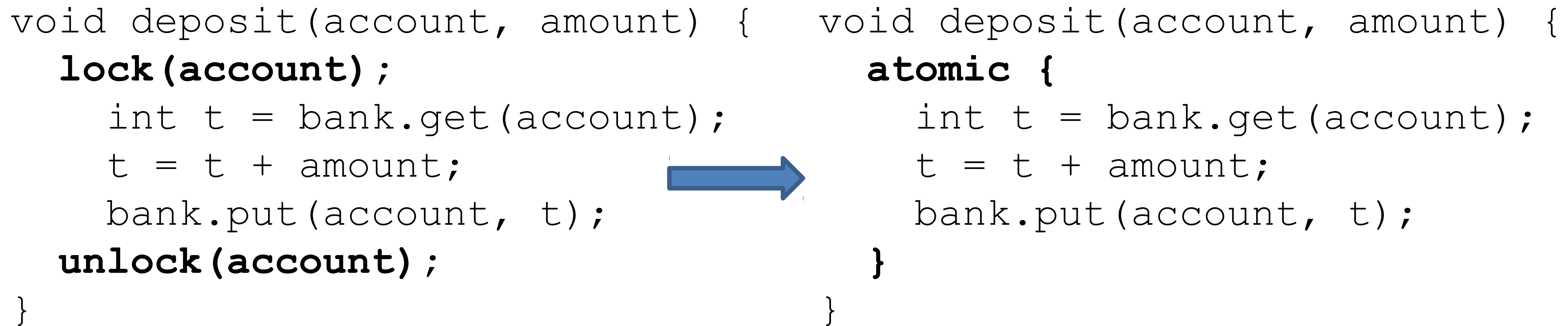
Parallel Programming

nd independent tasks in the algorithm
ap tasks to execution units (e.g. threads)
efine and implement synchronization among tasks
void races and deadlocks, address memory
 model issues, ...
4. Compose parallel tasks
5. Recover from errors

Transactional Memory

6. Ensure scalability
7. Manage locality
8. ....

Transactional Programming



- Declarative Synchronization – What, not How
- System implements Synchronization transparently

Transactional Memory

Memory Transaction - An atomic and isolated sequence of memory accesses

Transactional Memory – Provides transactions for threads running in a shared address space

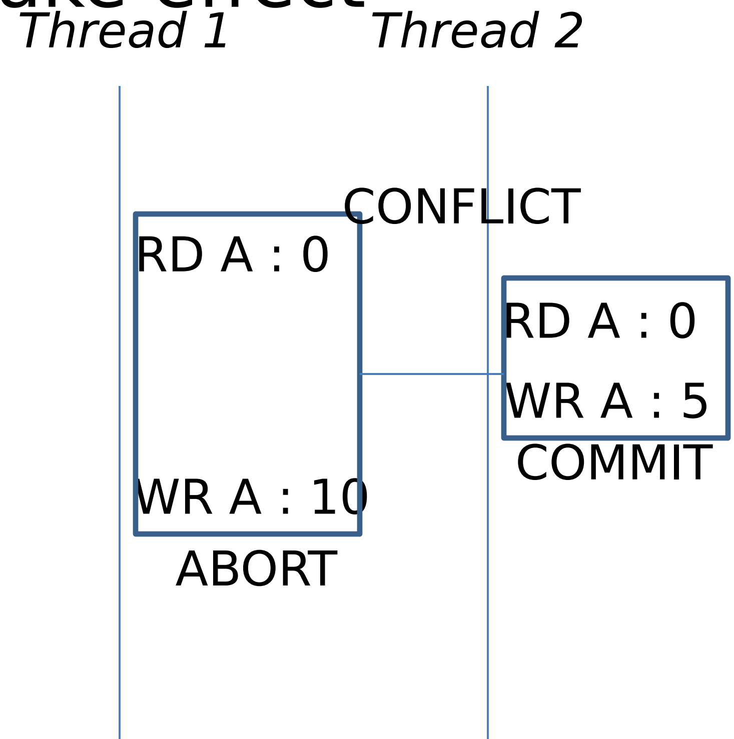
Transactional Memory - Atomicity

Atomicity – On transaction commit, all memory updates appear to take effect at once; on transaction abort, none of the memory updates appear to take effect

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

RD

WR



Transactional Memory - Isolation

Isolation – No other code can observe updates before commit

Programmer only needs to identify operation sequence that should appear to execute atomically to other, concurrent threads

Transactional Memory - Serializability

Serializability – Result of executing concurrent transactions on a data structure must be identical to a result in which these transactions executed serially.

Some advantages of TM

- ase of use (declarative)
- omposability
- xpected performance of fine-grained locking

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);
  }
  }
}
```

1. Fine grained locking Can lead to deadlock
2. Need some global locking discipline now

Composability : Locks

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

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

1. Fine grained locking Can lead to deadlock
2. Coarse grained 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);
  }
}
```

Serialization for transfer(A,B,100) and transfer(B,A,
Concurrency for transfer(A,B,100) and transfer(C,D,

Some issues with TM

- /O and unrecoverable actions
- Atomicity violations are still possible
- nteraction with non-transactional code

Atomicity Violation

Thread 1

```
atomic {
  …
  ptr = A;
  …
}

atomic {
  B = ptr->field;
}
```

Thread 2

```
atomic {
  …
  ptr = NULL;
}
```

nteraction with non-transactional code

Thread 1

```
lock_acquire(lock);
  obj.x = 1;
  if (obj.x != 1)
    fireMissiles();
lock_release(lock);
```

Thread 2

```
obj.x = 2;
```

nteraction with non-transactional code

Thread 1

```
atomic {
  obj.x = 1;
  if (obj.x != 1)
    fireMissiles();
}
```

Thread 2

```
obj.x = 2;
```

nteraction with non-transactional code

```
       Thread 1                  Thread 2
atomic {
  obj.x = 1;                     obj.x = 2;
  if (obj.x != 1)
    fireMissiles();
}
```

Weak Isolation – Transactions are serializable only against other transactions

Strong Isolation – Transactions are serializable against all memory accesses (Non-transactional LD/ST are 1-instruction TXs)

Nested Transactions

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

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

Semantics of Nested Transactions

- Flattened
- Closed Nested
- Open Nested

Nested Transactions - Flattened

```
int x = 1;
atomic {
   x = 2;
   atomic flatten {
      x = 3;
      abort;
   }
}
```

Nested Transactions - Closed

```
int x = 1;
atomic {
   x = 2;
   atomic closed {
      x = 3;
      abort;
   }
}
```

Nested Transactions - Open

```
int x = 1;
atomic {
  x = 2;
  atomic open {
    x = 3;
  }
  abort;
}
```

lested Transactions – Open – Use Case

```
int counter = 1;
atomic {
    …
    atomic open {
        counter++;
    }
}
```

ested Transactions ≠ Nested Parallelism!

```
1  atomic {
2    int *results =
3      get_results(&n);
4    sort(results, n);
5    for (i = 0; i < 10; i++)
6      sum += results[i];
7  }
```

(a) Application code.

```
1  void sort(int *list, int n) {
2    if (n == 1) return;
3    atomic {
4      sort(list, n/2);
5      sort(list + n/2, n - n/2);
6      merge(list, n/2, n - n/2);
7    }
8  }
```

(b) Sequential library implementation.

```
1  void sort(int *list, int n) {
2    if (n == 1) return;
3    atomic {
4      tid = spawn(sort, list, n/2);
5      sort(list + n/2, n - n/2);
6      wait(tid);
7      merge(list, n/2, n - n/2);
8    }
9  }
```

(c) Parallel library implementation.

Intelligent Speculation for Pipelined Multithreading
– Neil Vachharajani, PhD Thesis, Princeton University

ransactional Programming - Summary

ansactions do not generate parallelism

ansactions target performance of fine-grained lockin

@ effort of coarse-grained locking

Various constructs studied previously (atomic, retry

orelse,...)

ifferent semantics (Weak/Strong Isolation, Nesting)

Lecture Outline

1. Recap of Transactions
2. Transactional Memory System Taxonomy
3. Software Transactional Memory
4. Hardware Accelerated STMs

TM Implementation

Data Versioning

- Eager Versioning
- Lazy Versioning

Conflict Detection and Resolution

- Pessimistic Concurrency Control
- Optimistic Concurrency Control

Conflict Detection Granularity

- Object Granularity
- Word Granularity
- Cache line Granularity

TM Implementation

Data Versioning

- Eager Versioning
- Lazy Versioning

Conflict Detection and Resolution

- Pessimistic Concurrency Control
- Optimistic Concurrency Control

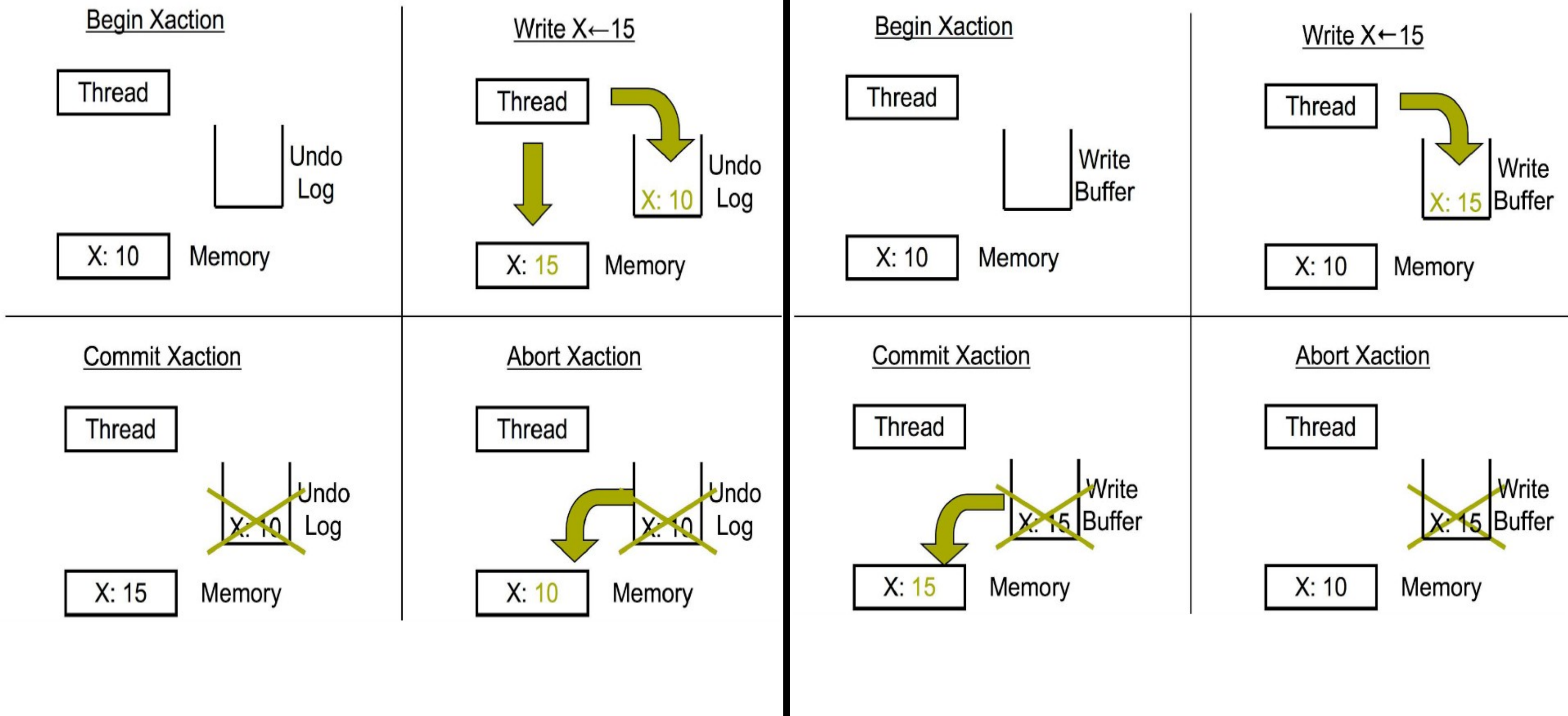
Conflict Detection Granularity

- Object Granularity
- Word Granularity
- Cache line Granularity

Data Versioning

Eager Versioning (Direct Update)

Lazy Versioning (Deferred Updat



TM Implementation

Data Versioning

- Eager Versioning
- Lazy Versioning

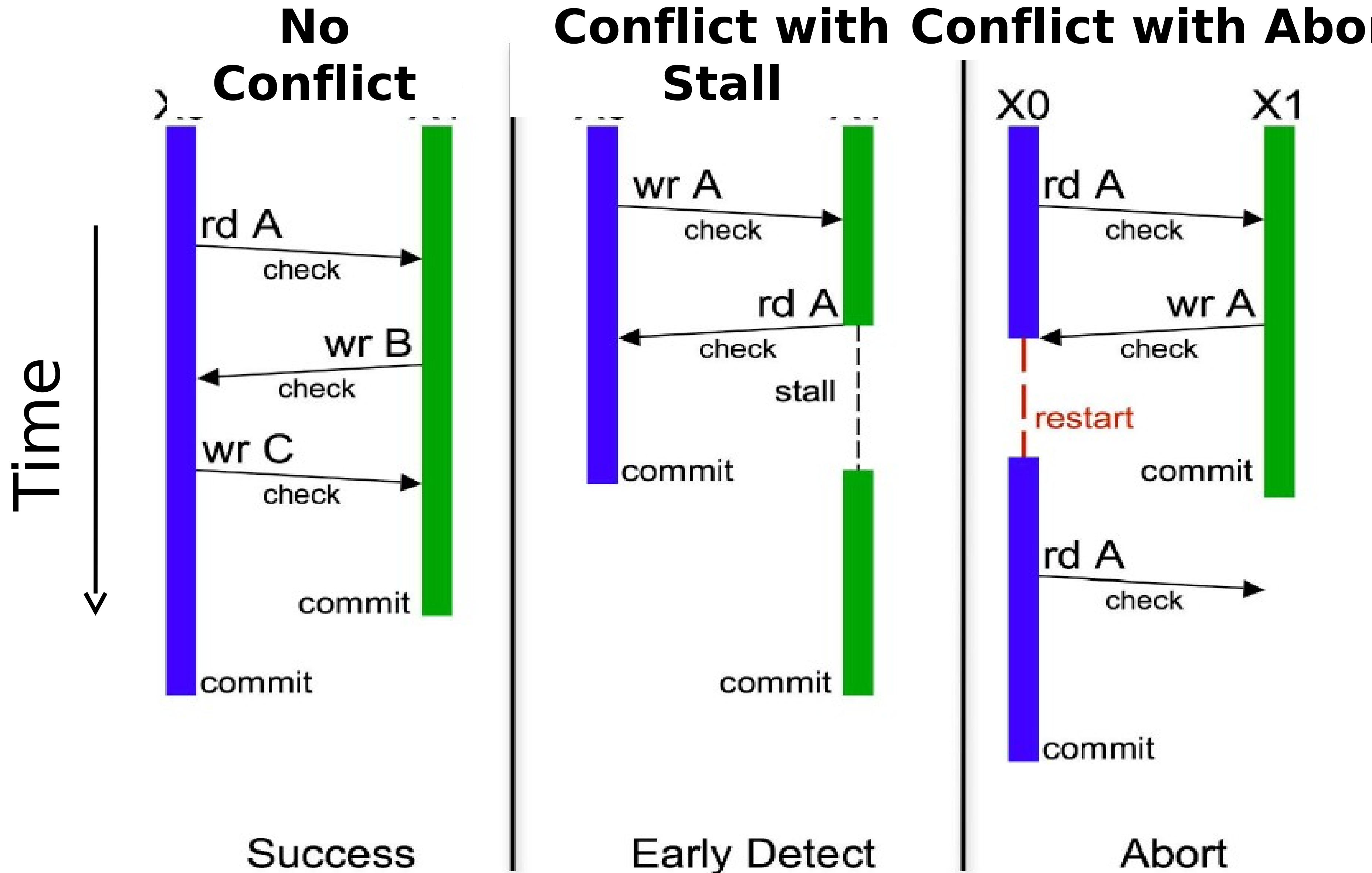
Conflict Detection and Resolution

- Pessimistic Concurrency Control
- Optimistic Concurrency Control

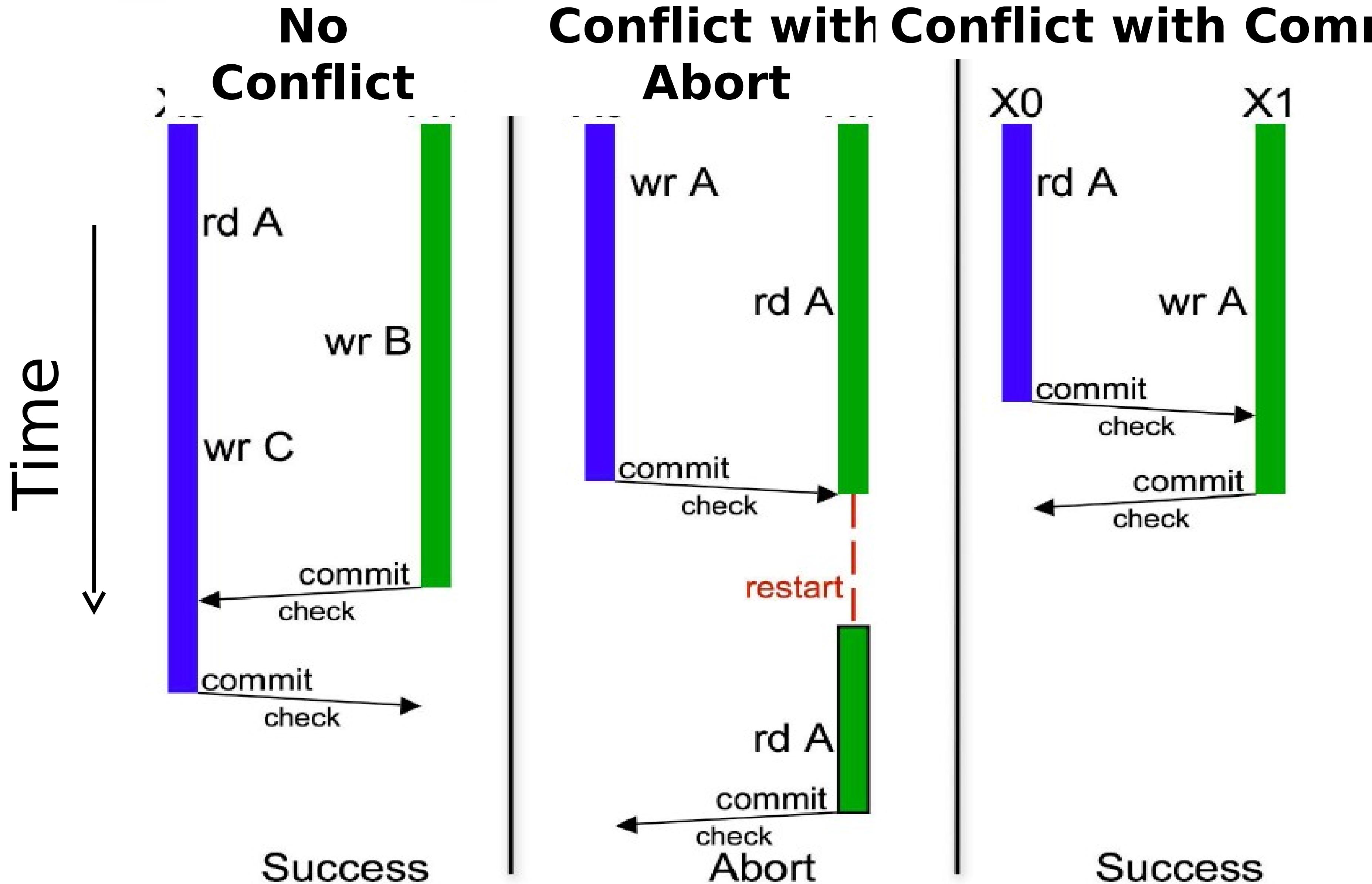
Conflict Detection Granularity

- Object Granularity
- Word Granularity
- Cache line Granularity

onflict Detection and Resolution - Pessimistic



onflict Detection and Resolution - Optimistic



TM Implementation

Data Versioning
- Eager Versioning
- Lazy Versioning

Conflict Detection and Resolution
- Pessimistic Concurrency Control
- Optimistic Concurrency Control

Conflict Detection Granularity
- Object Granularity
- Word Granularity
- Cache line Granularity

Examples

Hardware TM

- Stanford TCC: Lazy + Optimistic
- Intel VTM: Lazy + Pessimistic
- Wisconsin LogTM: Eager + Pessimistic
- UHTM
- SpHT

Software TM

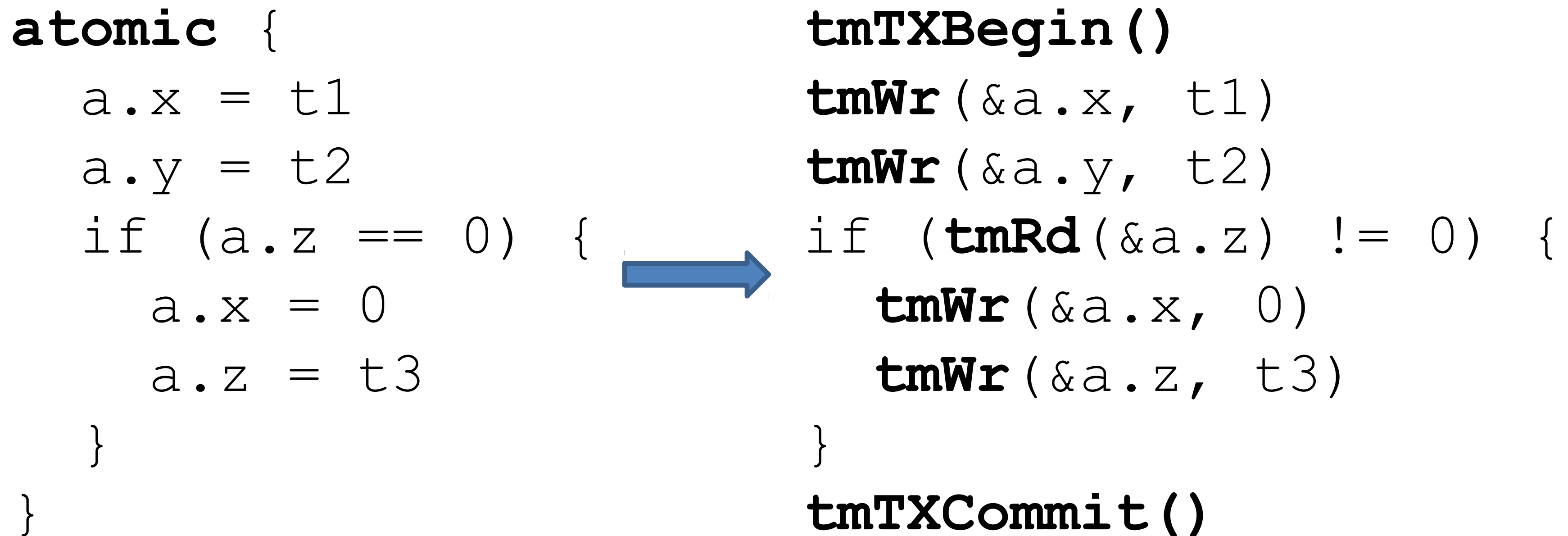
- Sun TL2: Lazy + Optimistic (R/W)
- Intel STM: Eager + Optimistic (R)/Pessimistic (W)
- MS OSTM: Lazy + Optimistic (R)/Pessimistic (W)
- Draco STM
- STMLite
- DSTM

nd many more at http://www.dolcera.com/wiki/index.php?title=Transactional_m

Lecture Outline

1. Recap of Transactions
2. Transactional Memory System Taxonomy
3. Software Transactional Memory (Intel McRT-STM)
4. Hardware Accelerated STMs

Software Transactional Memory (STM)



Intel McRT-STM

Strong or Weak Isolation	Weak
Transaction Granularity	Word or Object
Lazy or Eager Versioning	Eager
Concurrency Control	Optimistic read, Pessimistic Write
Nested Transaction	Closed

McRT-STM Runtime Data Structures

Transaction Descriptor (per thread)

- Used for conflict detection, commit, abort, ...
- Includes read set, write set, undo log or write buffer

Transaction Record (per datum)

- Pointer-sized record guarding shared datum
- Tracks transactional state of datum

  Shared: Read-only access by multiple readers

    Value is odd (low bit is 1)

  Exclusive: Write-only access by single owner

    Value is aligned pointer to owning transaction's descriptor

McRT-STM: Example

```
Class Foo {
   int x;
   int y;
};
Foo bar, foo;
```

T1

```
atomic {
   t = foo.x;
   bar.x = t;
   t = foo.y;
   bar.y = t;
}
```

T2

```
atomic {
   t1 = bar.x;
   t2 = bar.y;
}
```

- **T1 copies foo into bar**
- **T2 reads bar, but should not see intermediate values**

McRT-STM: Example

T1

```
stmStart();
  t = stmRd(foo.x);
  stmWr(bar.x,t);
  t = stmRd(foo.y);
  stmWr(bar.y,t);
stmCommit();
```

T2

```
stmStart();
  t1 = stmRd(bar.x);
  t2 = stmRd(bar.y);
stmCommit();
```

- **T1 copies foo into bar**
- **T2 reads bar, but should not see intermediate values**

McRT-STM Operations

STM read (Optimistic)

- Direct read of memory location (eager versioning)
- Validate read data
  - Check if unlocked and data version <= local timestamp
  - If not, validate all data in read set for consistency

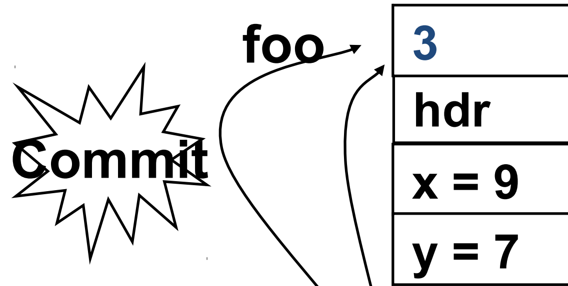
validate() {for <txnrec,ver> in transaction’s read set, if (*txnrec != ver) abort();}

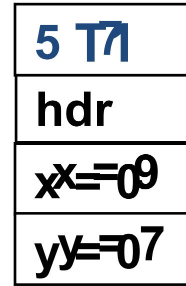
- Insert in read set
- Return value

STM write (Pessimistic)

- Validate data
  - Check if unlocked and data version <= local timestamp
- Acquire lock
- Insert in write set
- Create undo log entry
- Write data in place (eager versioning)

McRT-STM: Example





bar

Abort

T1

```
stmStart();
  t = stmRd(foo.x);
  stmWr(bar.x,t);
  t = stmRd(foo.y);
  stmWr(bar.y,t);
stmCommit;
```

T2

```
stmStart();
  t1 = stmRd(bar.x);
  t2 = stmRd(bar.y);
stmCommit();
```

T2 waits

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

T2 should read [0, 0] or should read [9,7]

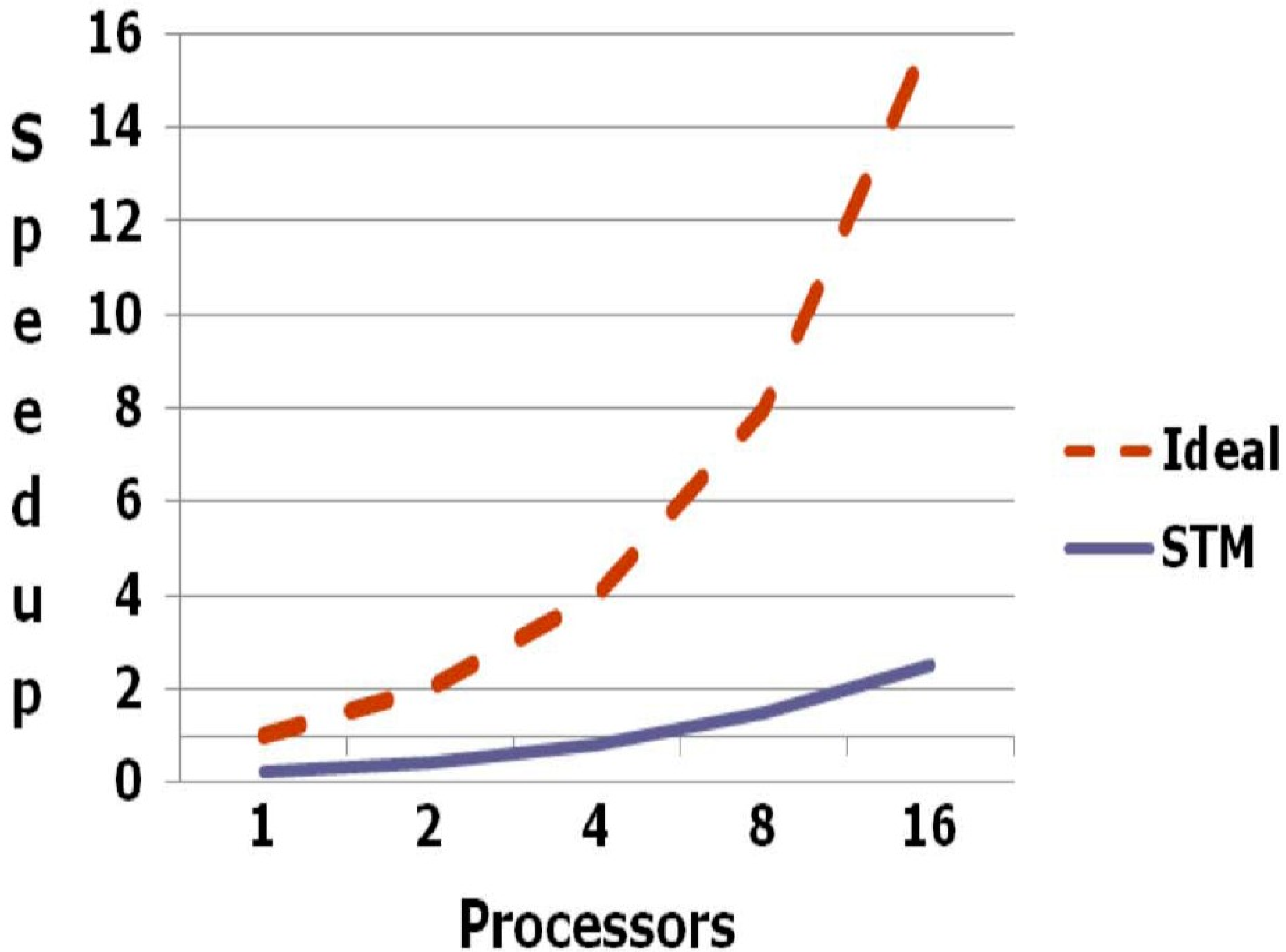
Undo <bar.x, 0> <bar.y, 0>

Reads <bar, 5> <bar, 7>

Lecture Outline

1. Recap of Transactions
2. Transactional Memory System Taxonomy
3. Software Transactional Memory (Intel McRT-STM)
4. Hardware Accelerated STMs (HASTM)

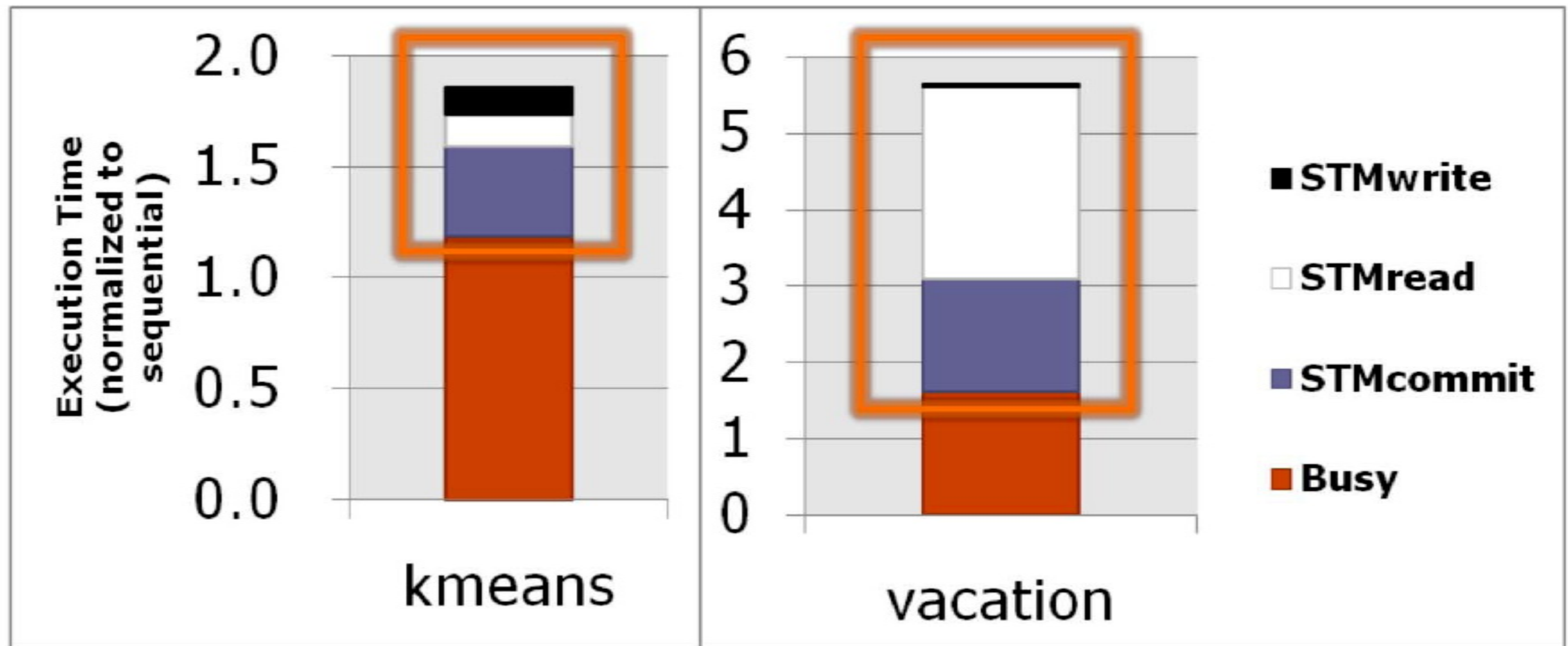
Hardware Support?



Types of Hardware Support

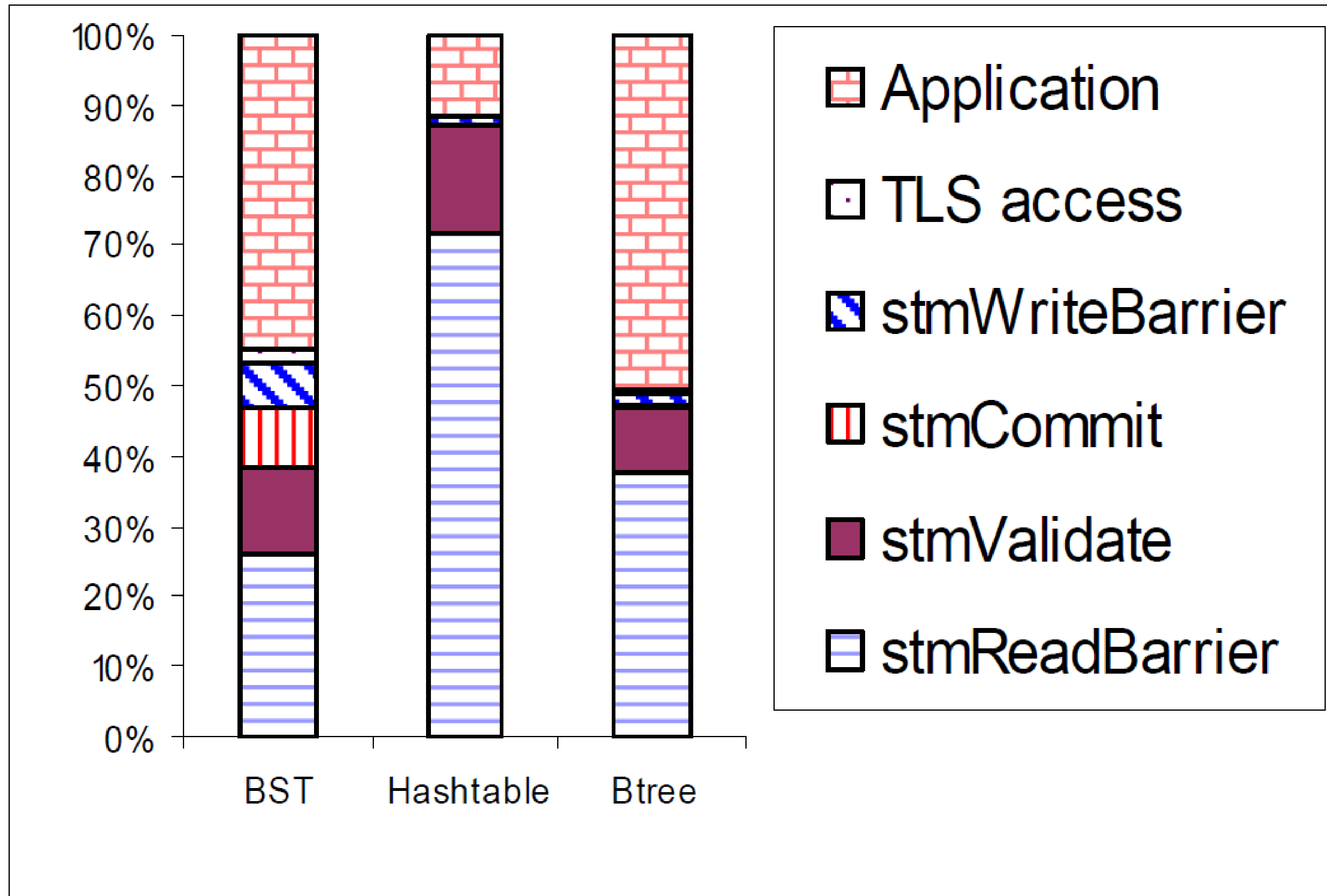
- Hardware-accelerated STM systems (HASTM, SigTM, …)
- Hardware-based TM systems (TCC, LTM, VTM, LogTM, …)
- Hybrid TM systems (Sun Rock)

Hardware Support?



1. 1.8x – 5.6x slowdown over sequential
2. Most time goes in read barriers and validation

Hardware Support?



chitectural Support for Software Transactional Memory – Saha et al.

Hardware-accelerated STM (HASTM)

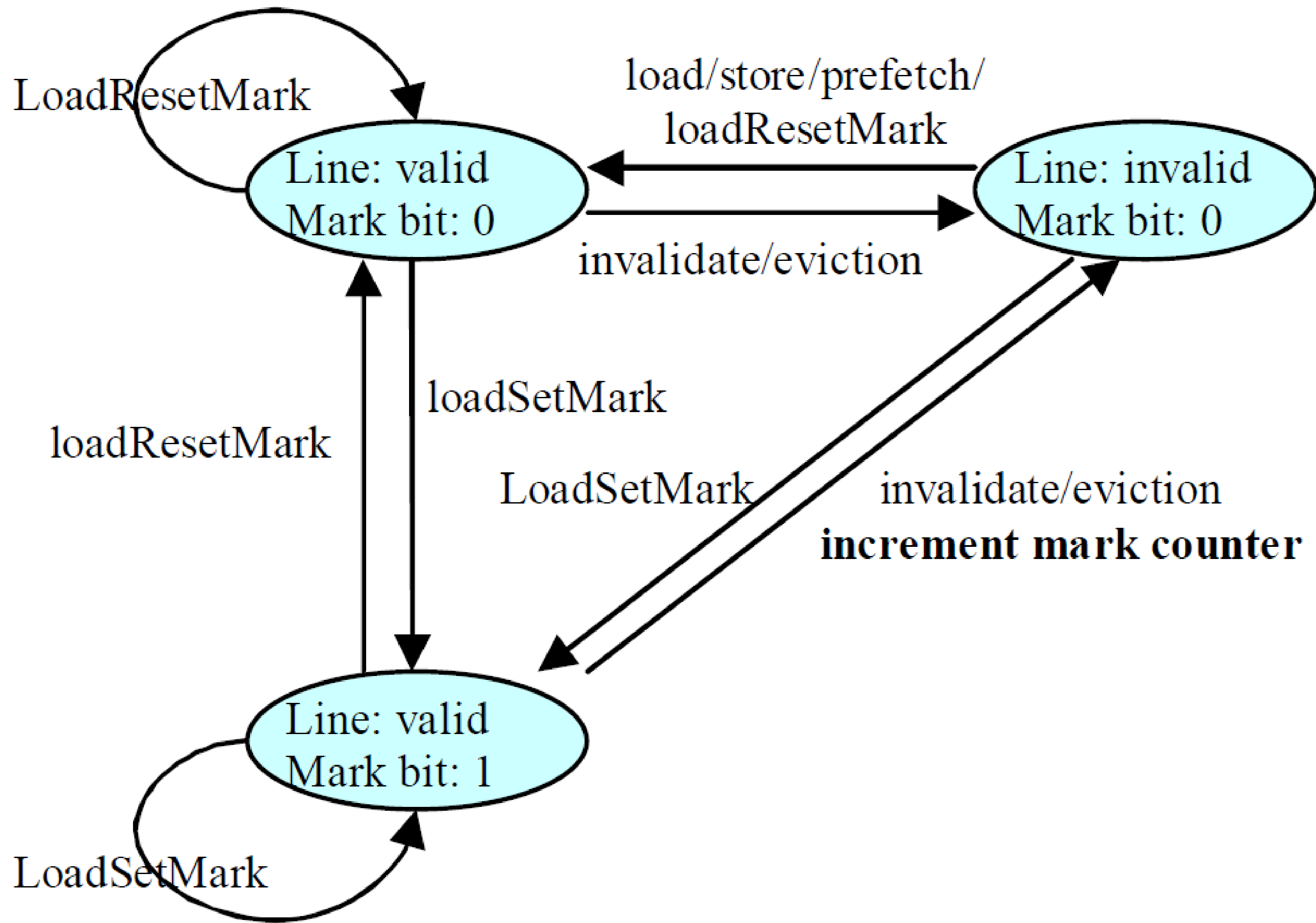
Hardware Support

- Per-thread mark bits at granularity of cache lines
- New register (*mark counter*)
- Used to build fast filters to speed up read barriers

ISA Extensions

- **loadSetMark(addr) –** Loads value at *addr* and sets mark bit associated with addr
- **loadResetMark(addr) –** Loads value at *addr* and clears mark bit associated with addr
- **loadTestMark(addr)** – Loads the value at *addr* and sets carry flag to value of mark bit
- **resetMarkAll()** – Clears all mark bits in cache and increments mark counter
- **resetMarkCounter()** – Resets mark counter
- **readMarkCounter()** – Reads mark counter value

HASTM – Cache line transitions



McRT-STM Operations

stmRd = stmRdBar + Rd

```
stmRdBar(TxnRec* rec) {
  void* recval = *rec;
  void* txndesc = getTxnDesc();
  if (recval == txndesc) return;
  if (isVersion(recVal) == false)
    recval =
handleContention(rec);
  logRead(rec,recval);
}
```

McRT-STM Operations

stmWr = stmWrBar + LogOldValue + Wr

```
stmWrBar(TxnRec* rec) {
  void* recval = *rec;
  void* txndesc = getTxnDesc();
  i f (recval == txndesc) return;
  if (isVersion(recVal) == false)
    recval = handleContention(rec);
  while (CAS(rec,recval,txndesc) == false)
    recval = handleContention(rec);
  logWrite(rec,recval);
}
```

McRT-STM Operations

stmRd = stmRdBar + Rd

```
mov eax, [rec] /* load TxRec */
cmp eax, txndesc /* do I own exclusive */
jeq done
test eax, #versionmask /* is a version no. */
jz contention
mov ecx, [txndesc + rdsetlog] /*get log ptr*/
test ecx, #overflowmask
jz overflow
add ecx, 8 /*inc log ptr*/
mov [txndesc + rdsetlog], ecx
mov [ecx – 8], rec /* logging */
mov [ecx – 4], eax /* logging */
done:
...
```

Manage Contention

Manage Overflow

Fast Path

Slow Path

HASTM Operations stmRd = stmRdBar + Rd

```
loadTestMark eax, [rec] /* check 1st access */
jnae done
loadSetMark eax, [rec]

test eax, #versionmask /* is a version no. */
jz contention

mov ecx, [txndesc + rdsetlog] /*get log ptr*/
test ecx, #overflowmask
jz overflow

add ecx, 8 /*inc log ptr*/
mov [txndesc + rdsetlog], ecx
mov [ecx – 8], rec /* logging */
mov [ecx – 4], eax /* logging */

done:
...
```

Manage Contention

Manage Overflow

Fast Path

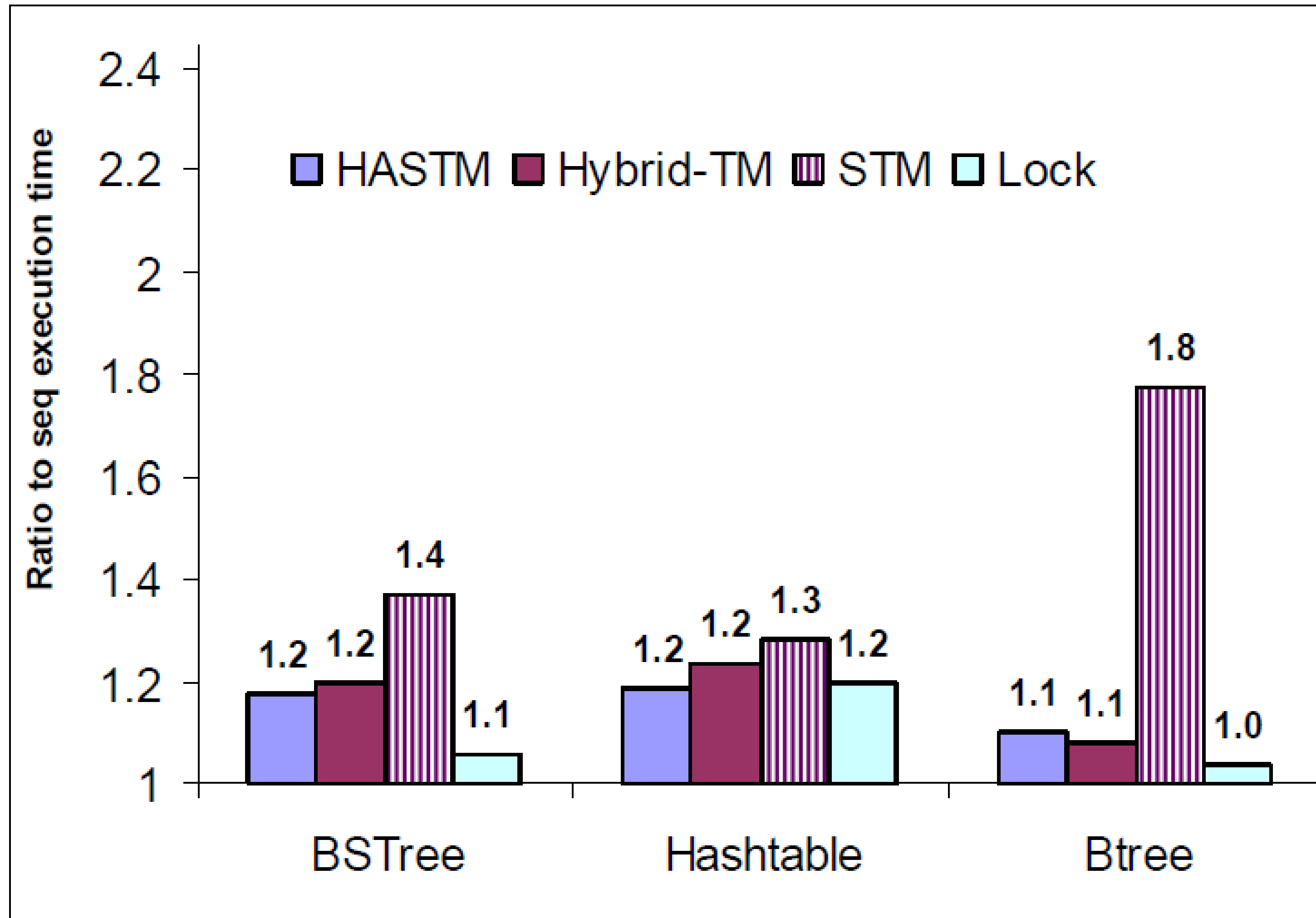
Slow Path

HASTM Operations

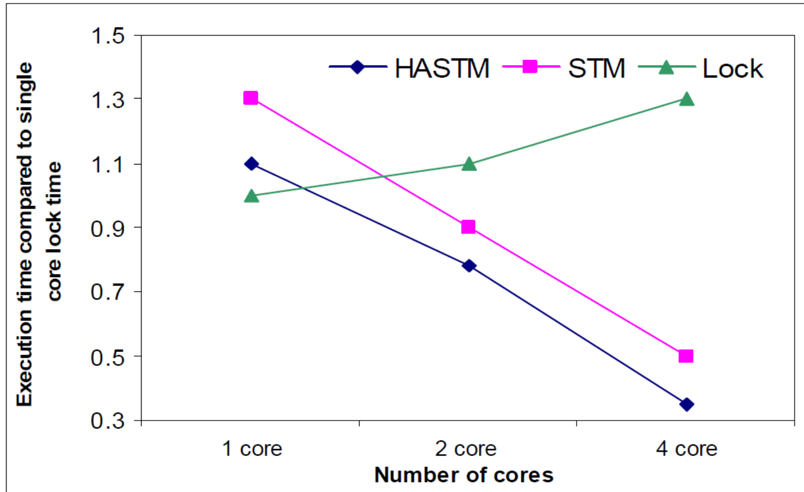
validate

validate() {
  markCount = ***readMarkCounter();***
  resetMarkAll();
  if (markCount == 0) /*no snoop or eviction*/
    return;
  /* perform full read set validation */
  for <txnrec,ver> in transaction’s read set
    if (*txnrec != ver)
      abort();

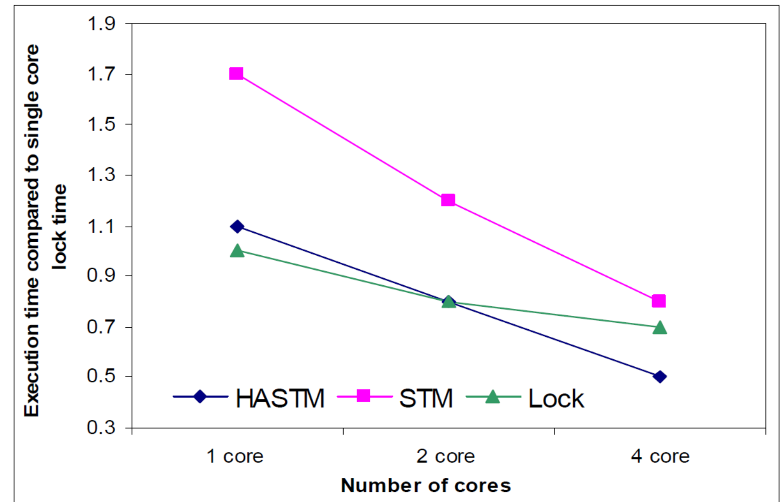
HASTM Performance Improvement



HASTM Performance Improvement – Multicore



Binary Search Tree



B-Tree

HASTM - Issues

Insufficient Cache Capacity

- Mark bits are only an acceleration mechanism
- Cache evictions, cause mark bits to be lost – revert to slow software validation

Weak Isolation

Other Slides

Intel McRT-STM

STM Characteristics

McRT-STM	
Strong or Weak Isolation	Weak
Transaction Granularity	Word or object
Direct or Deferred Update Lazy/Eager	Direct Eager
Concurrency Control	Optimistic read, Pessimistic write
Synchronization	Blocking
Conflict Detection	Early write–write conflict Late write–read conflict
Inconsistent Reads	Inconsistency toleration
Conflict Resolution	Abort
Nested Transaction	Closed
Exceptions	

ther Transactional Programming Primitives

User-triggered abort: abort

```
void transfer(A,B,amount)
  atomic{
    try{
      work();
    }
    catch(error1) {fix_code();}
    catch(error2) {abort;}
  }
```

2.Conditional synchronization: retry

```
Object blockingDequeue(q)
  atomic{
    if(q.isEmpty()) retry;
    return dequeue();
  }
```

3.Composing code sequences: orelse

```
atomic{q1.blockingDequeue();
}orelse{q2.blockingDequeue();
}
```