Designing an Effective Hybrid Transactional Memory System

Chí Cao Minh

28 May 2008

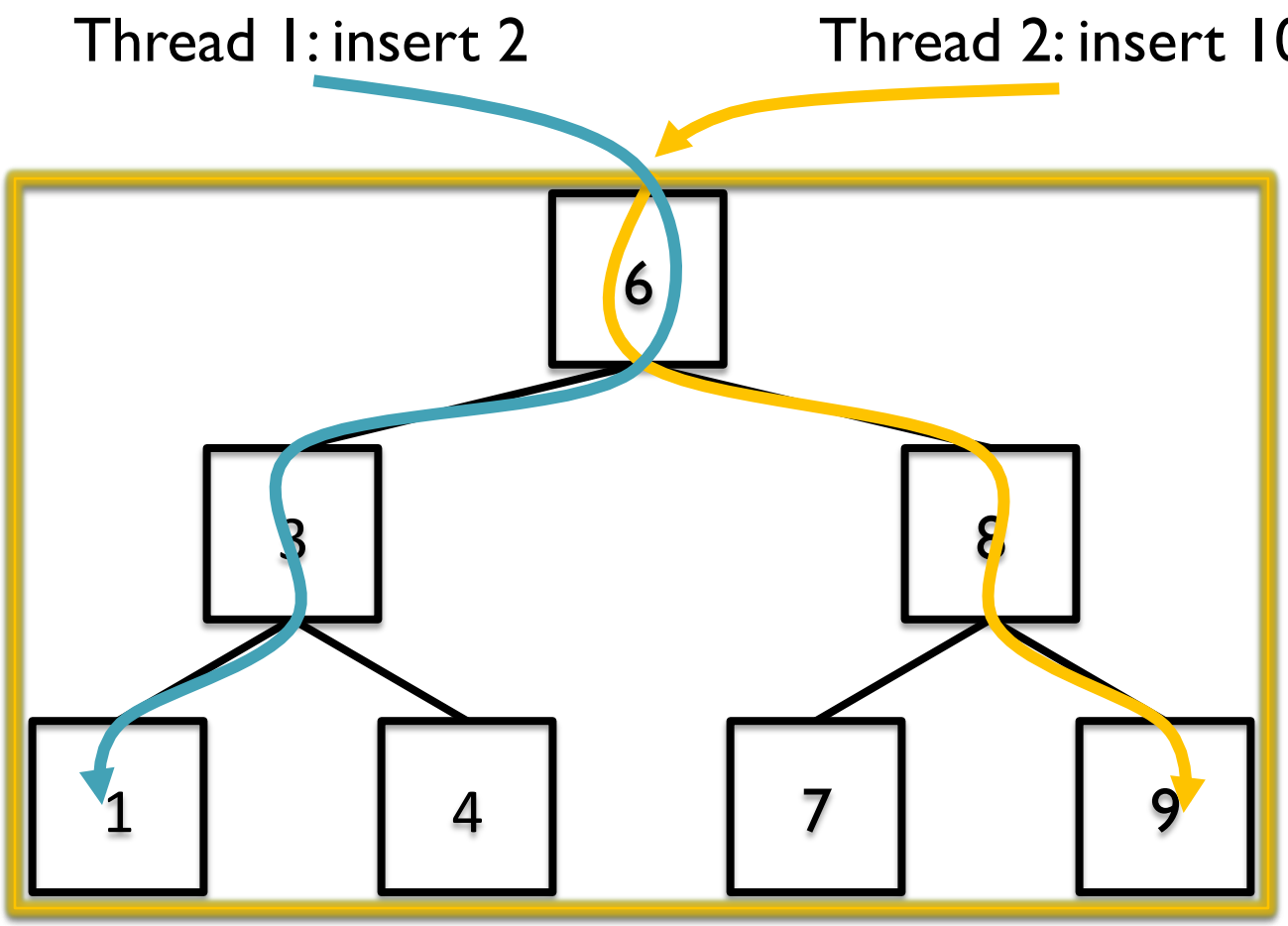
The Need for Multiprocessors

- Uniprocessor systems hitting limits
  - Design complexity overwhelming
  - Power consumption increasing dramatically
  - Instruction-level parallelism exhausted
- Solution is multiprocessor systems
  - Simpler processor design (but many of them)
  - Reduce power requirements
  - Expose opportunity for thread-level parallelism

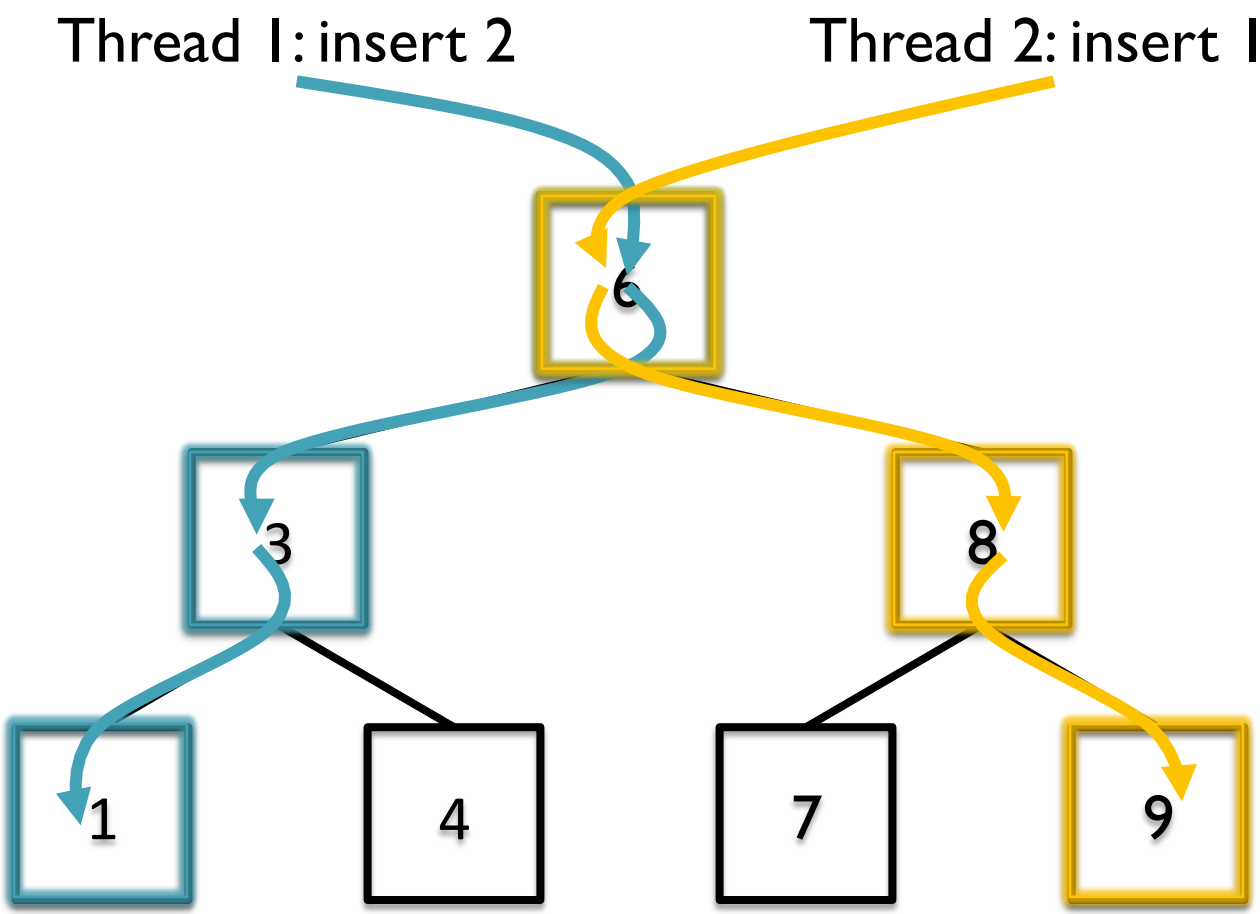
Programming Multiprocessors

- Commonly achieved via lock-based parallel programs
- Unfortunately, parallel programming with locks is hard
  - Option 1: Coarse-grain locks
    - Simplicity at less concurrency
  - Option 2: Fine-grain locks
    - Better performance (maybe) at more complexity

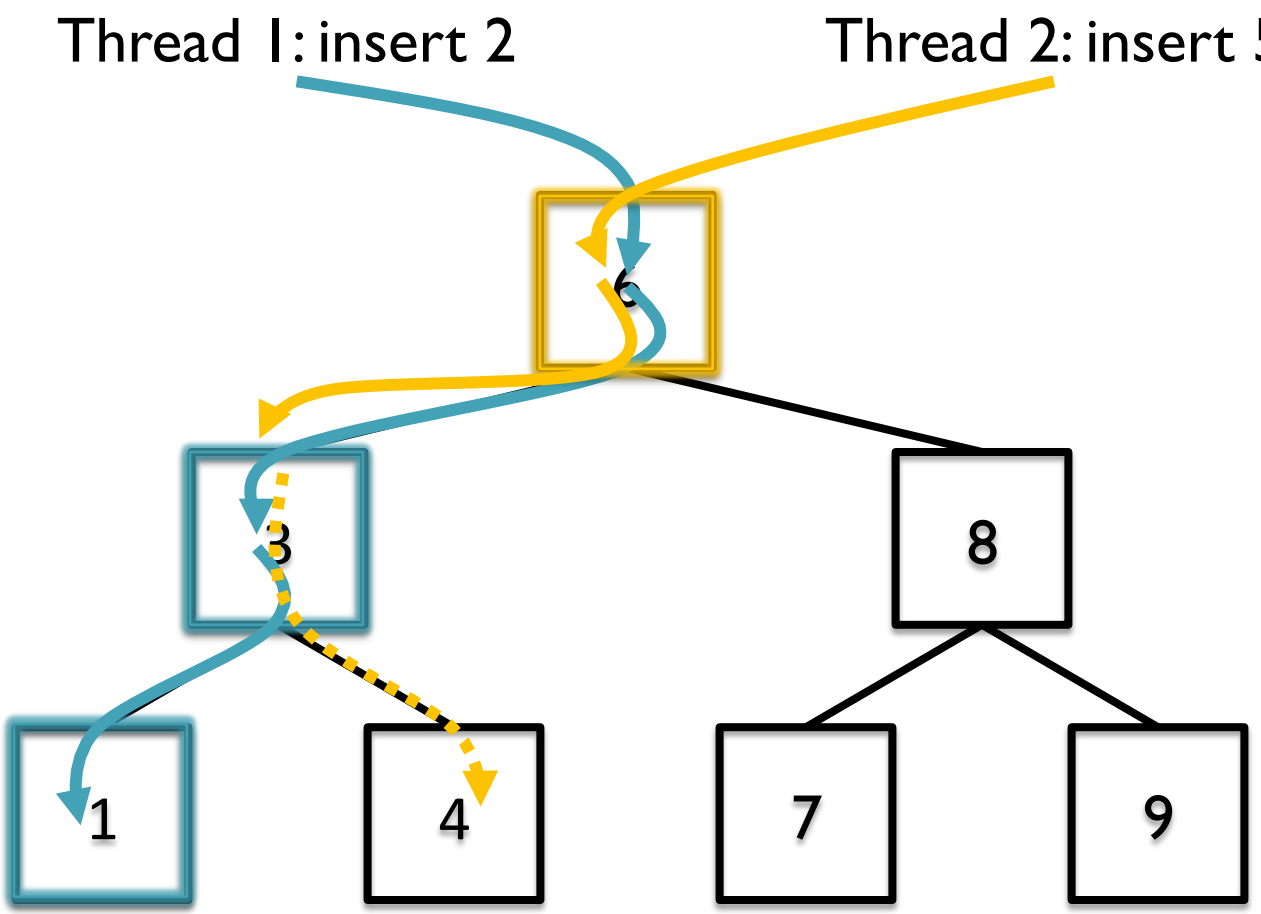
Parallel Programming With Locks



Parallel Programming With Locks



Parallel Programming With Locks



Transactional Memory (TM)

- What is a transaction?
  - Group of instructions in computer program:

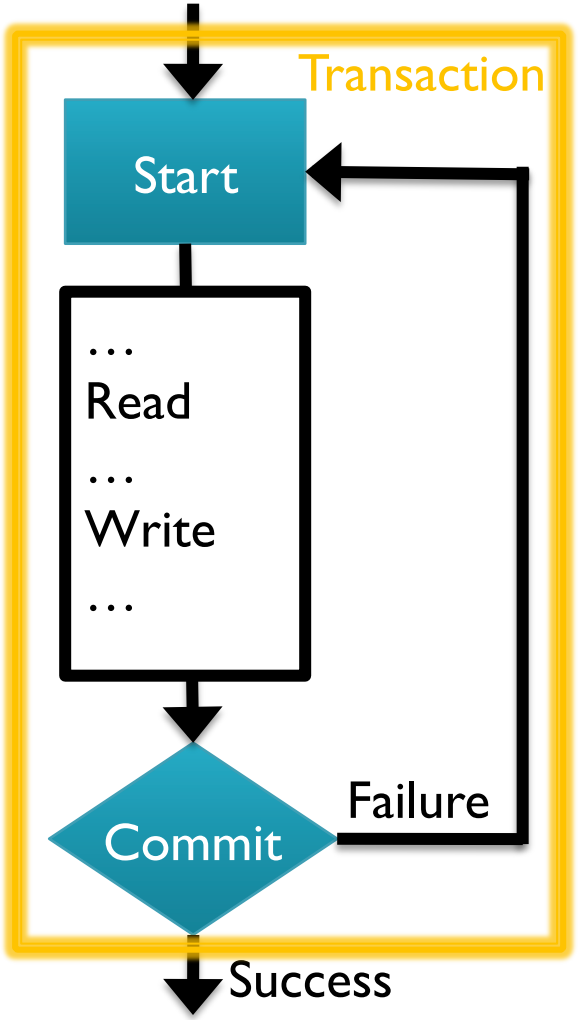
```
atomic {
  if (x != NULL) x.foo();
  y = true;
}
```

  - Required properties: Atomicity, Isolation, Serializability

- Key idea: Use transactions to build parallel programs
  - Large atomic blocks simplify parallel programming
  - Simplicity of coarse-grain locks with speed of fine-grain locks

Optimistic Concurrency Control

- Life cycle of a transaction:
  - Start
  - Speculative execution (optimistic)
  - Build read-set and write-set
    - Write-set manages write versioning
  - Commit
    - Fine-grain R-W & W-W conflict detection
  - Abort & rollback



Parallel Programming With TM

Thread 1: insert 2

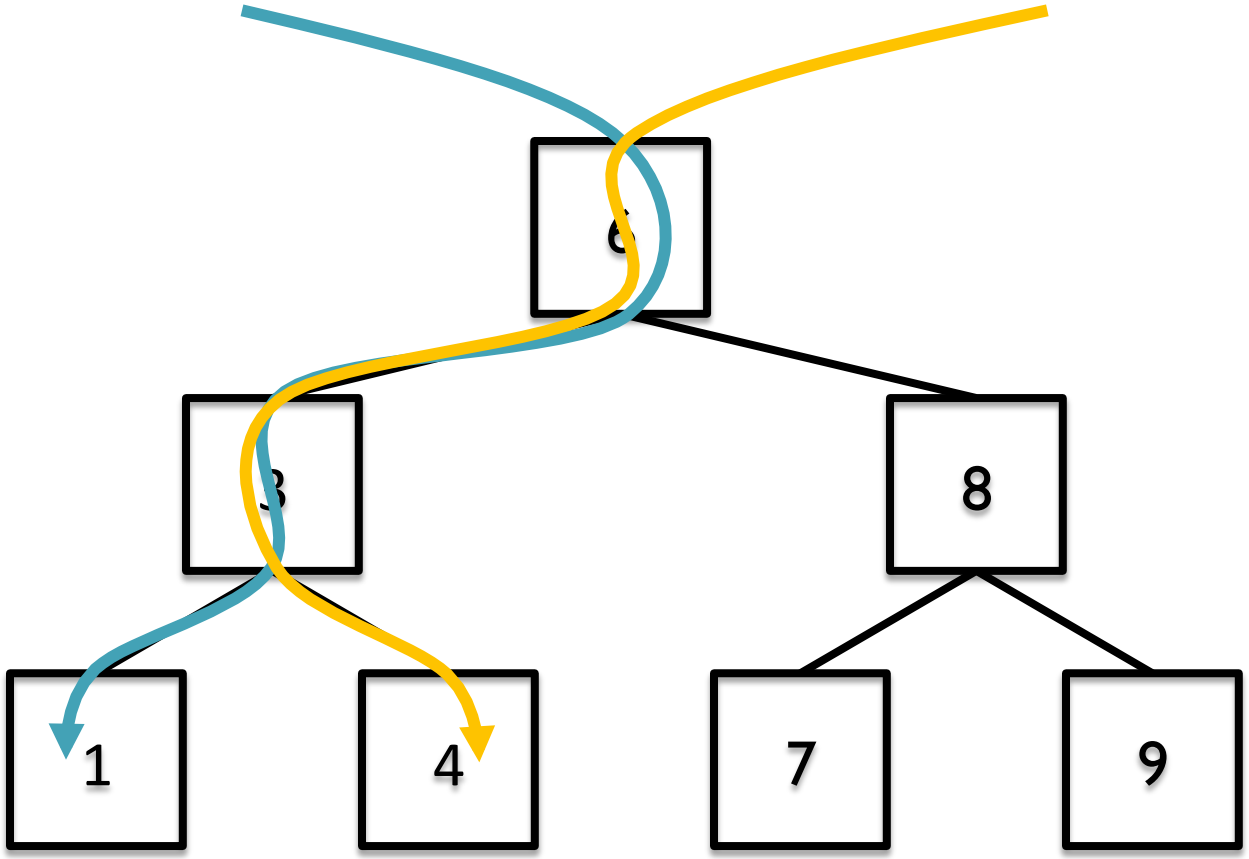
Read-set: 6, 3, 1

Write-set: 1

Thread 2: insert 5

Read-set: 6, 3, 4

Write-set: 4



Parallel Programming With TM

Thread 1: insert 2

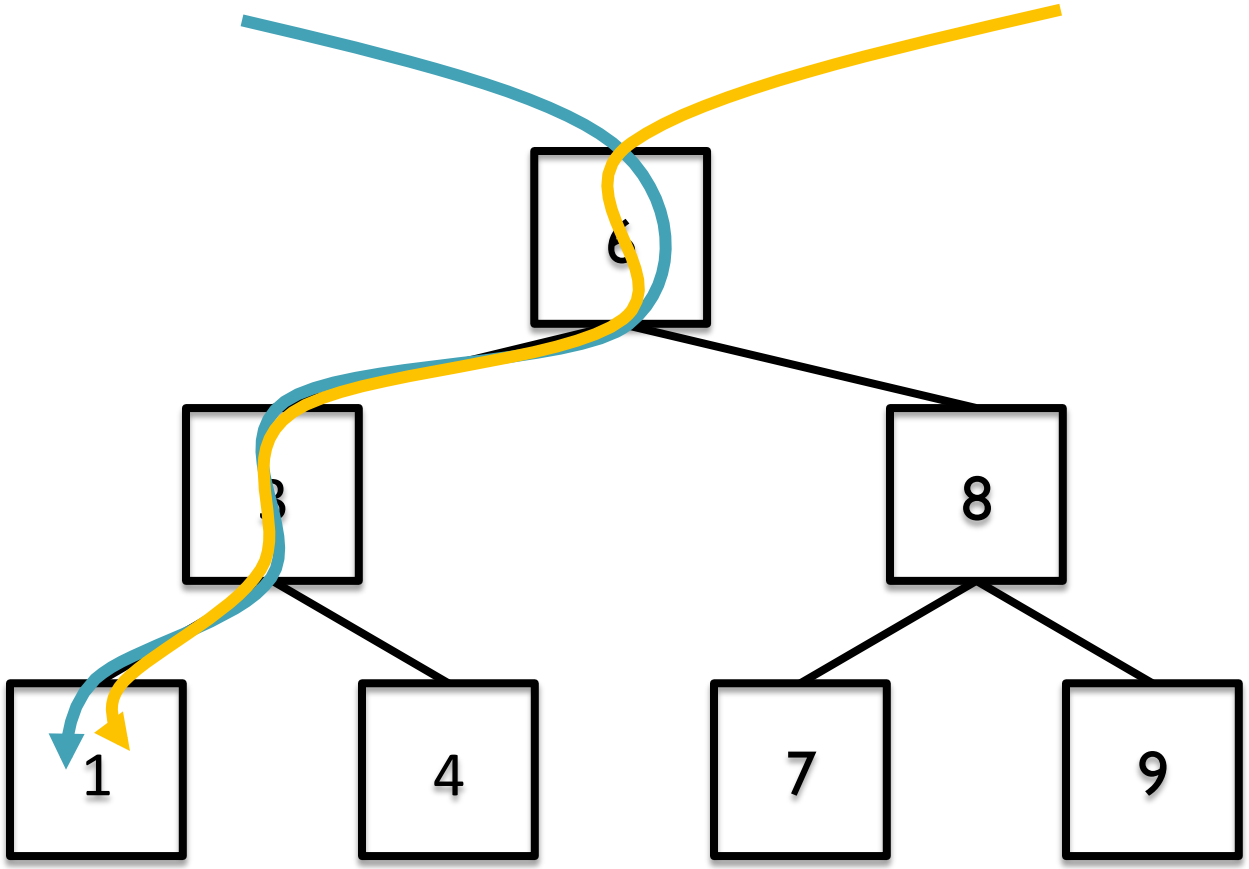
Read-set: 6, 3, 1

Write-set: 1

Thread 2: insert 0

Read-set: 6, 3, 1

Write-set: 1



TM Implementations

- TM can be implemented in hardware or software

- Hardware-based (HTM)
  - [Herlihy 93], [Rajwar 02], [Hammond 04], [Moore 06]
  - Strengths: high performance & predictable semantics
  - Weaknesses: costly & inflexible

- Software-based (STM)
  - [Shavit 95], [Herlihy 03], [Harris 03], [Saha 06], [Dice 06]
  - Strengths: low-cost & flexible
  - Weaknesses: low performance & unpredictable semantics

TM Community Wishlist

- Standard method to compare TM systems
  - Each TM system evaluated with different apps
  - How to pick the better of two HTMs?

- TM system that combines strengths of HTM and STM
  - High-performance
  - Flexibility
  - Low-cost
  - Predictable semantics

My Contributions

- STAMP: Benchmark suite for TM
  - 8 applications specifically for evaluating TM
  - Comprehensive *breadth* and *depth* analysis
  - *Portable* to many kinds of TMs
  - Public release: `http://stamp.stanford.edu`
  - *IEEE Intl. Symposium on Workload Characterization* (IISWC) 2008

- Signature-Accelerated TM (SigTM): Hybrid TM
  - Hardware acceleration of software transactions
  - Fast, flexible, cost-effective, & predictable semantics
  - *Intl. Symposium on Computer Architecture* (ISCA) 2007

Outline

- Background & Motivation
- STAMP: Benchmark suite for TM
- SigTM: Effective hybrid TM
  - Fast, flexible, low-cost
  - Predictable semantics
- Conclusions

Computer Benchmarks

- What is a benchmark?
  - Program used to evaluate computer performance
  - Help accelerate innovation in computer design
- Benchmarks for multiprocessors
  - SPLASH-2 (1995), SPEComp (2001), PARSEC (2008)
  - Not good for evaluating TM
    - Regular algorithms without synchronization problems
- Benchmarks for TM systems
  - Microbenchmarks from RSTMv3 (2006)
  - STMBench7 (2007)
  - Haskell applications by Perfumo et. al (2007)

TM Benchmark Requirements

- *Breadth:* variety of algorithms & app domains
- *Depth:* wide range of transactional behaviors
- *Portability:* runs on many classes of TM systems

Benchmark	Breadth	Depth	Portability	Comments
RSTMv3	no	yes	yes	Microbenchmarks
STMbench7	no	yes	yes	Single program
Perfumo et al.	no	yes	no	Microbenchmarks; Written in Haskell

STAMP Meets 3 Requirements

- Breadth
  - 8 applications covering different domains & algorithms
  - Applications not trivially parallelizable
- Depth
  - Wide range of transactional behaviors
    - Transaction length
    - Read and write set size
    - Contention amount
  - Most spend significant execution time in transactions
- Portability
  - Written in C with macro-based transaction annotations
  - Works with HTM, STM, and hybrid TM

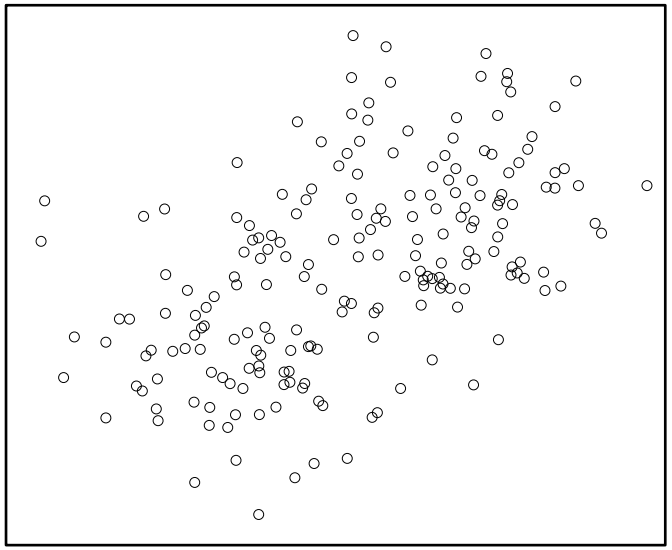
STAMP Applications

Application	Domain	Description
bayes	Machine learning	Learns structure of a Bayesian network
genome	Bioinformatics	Performs gene sequencing
intruder	Security	Detects network intrusions
kmeans	Data mining	Implements K-means clustering
labyrinth	Engineering	Routes paths in maze
ssca2	Scientific	Creates efficient graph representation
vacation	Online transaction processing	Emulates travel reservation system
yada	Scientific	Refines a Delaunay mesh

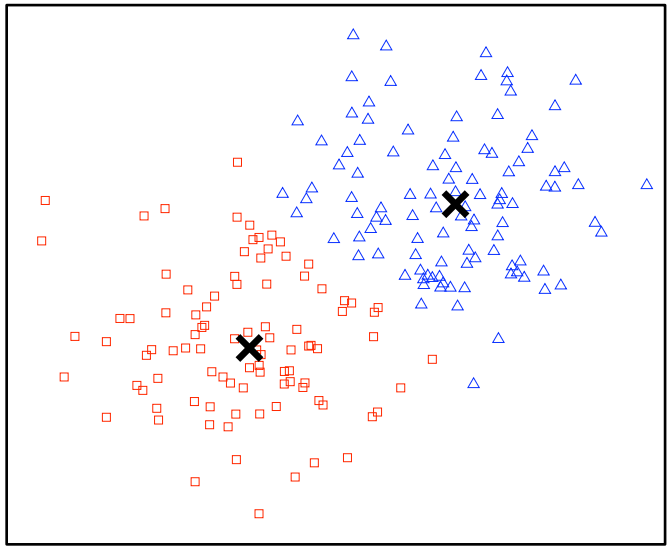
Kmeans Description

- Groups data into K clusters

Initial data

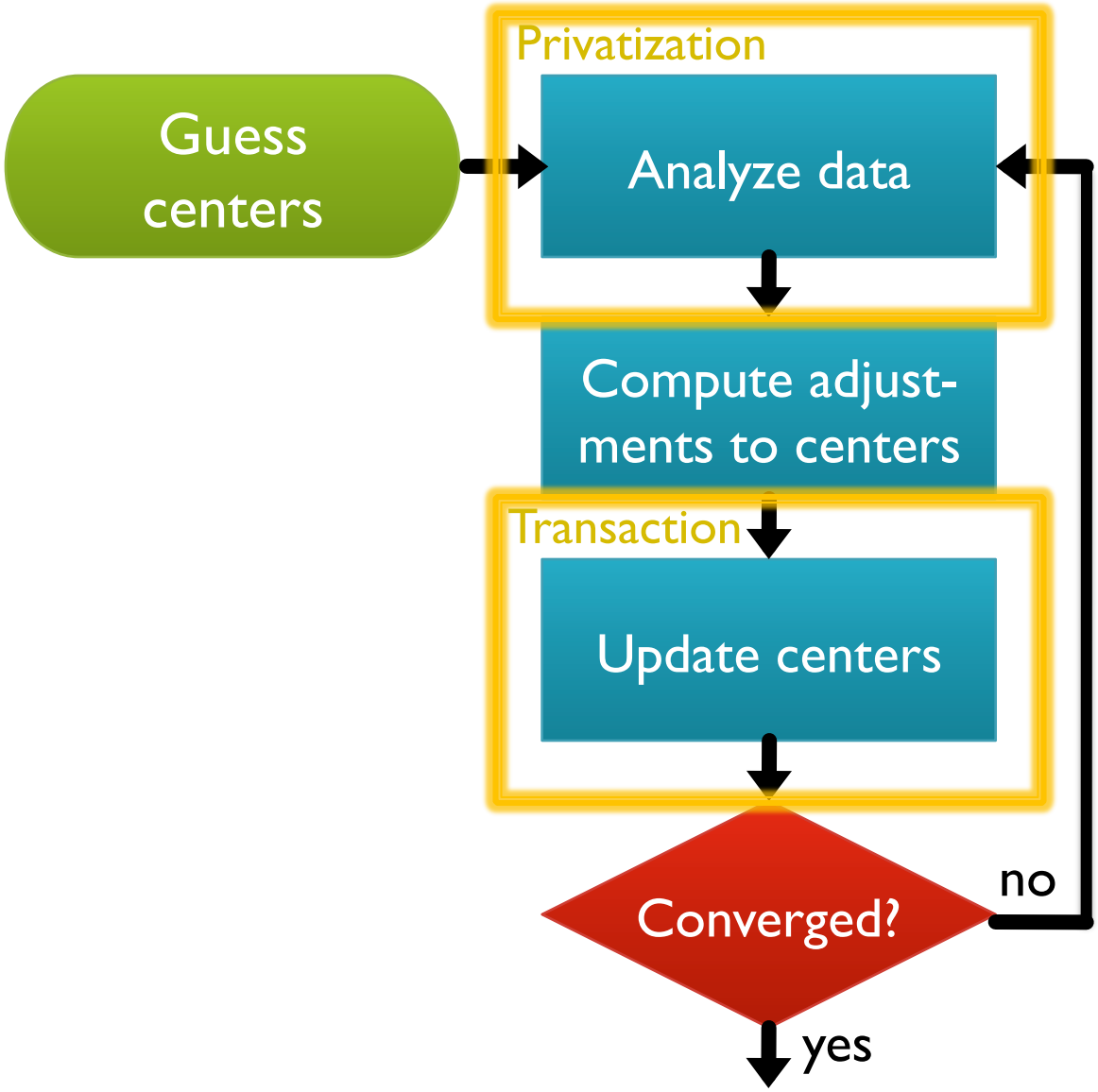


Grouped data (K = 2)



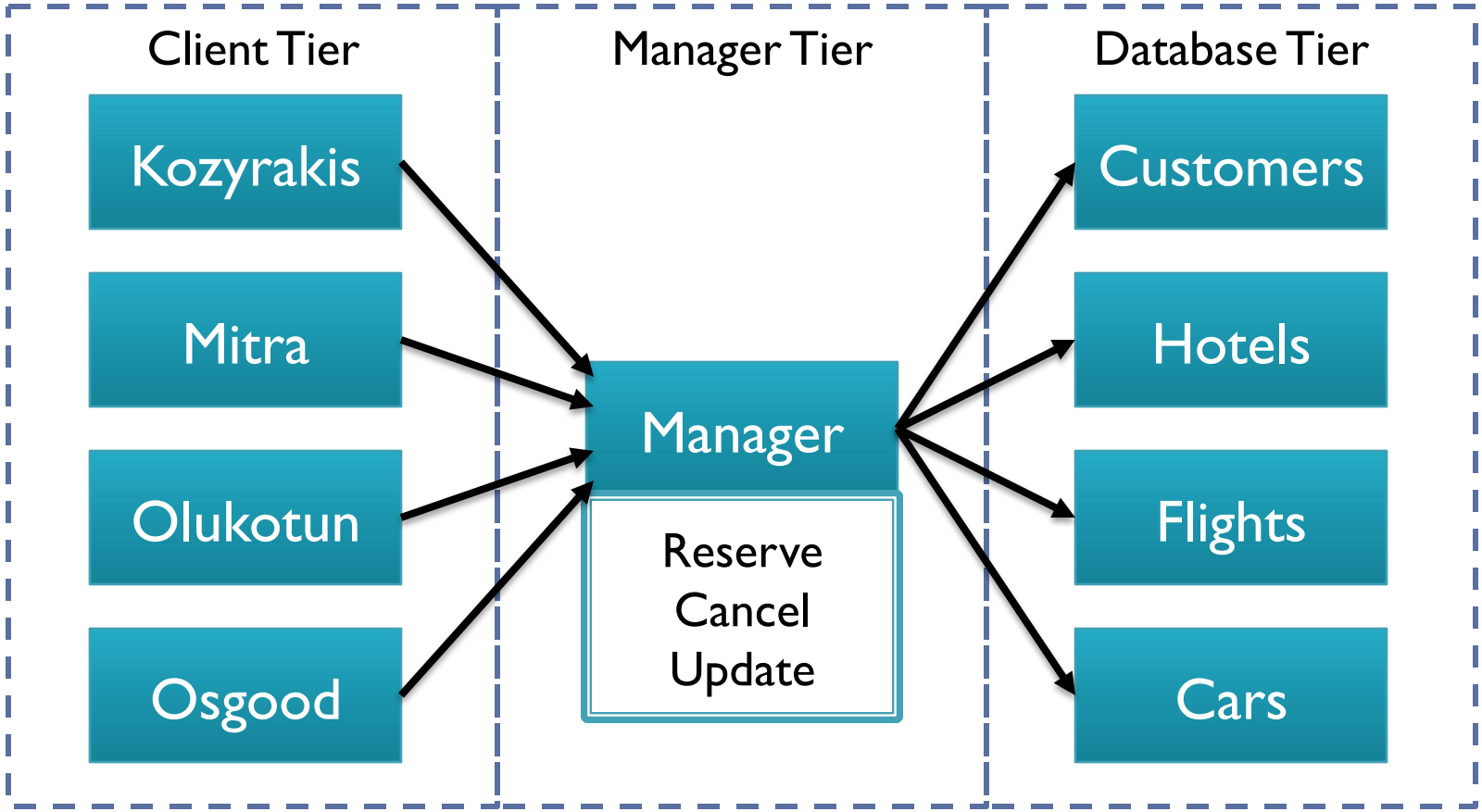
- Possible applications:
  - *Biology:* plant and animal classification
  - *WWW:* analyze web traffic for patterns

Kmeans Algorithm

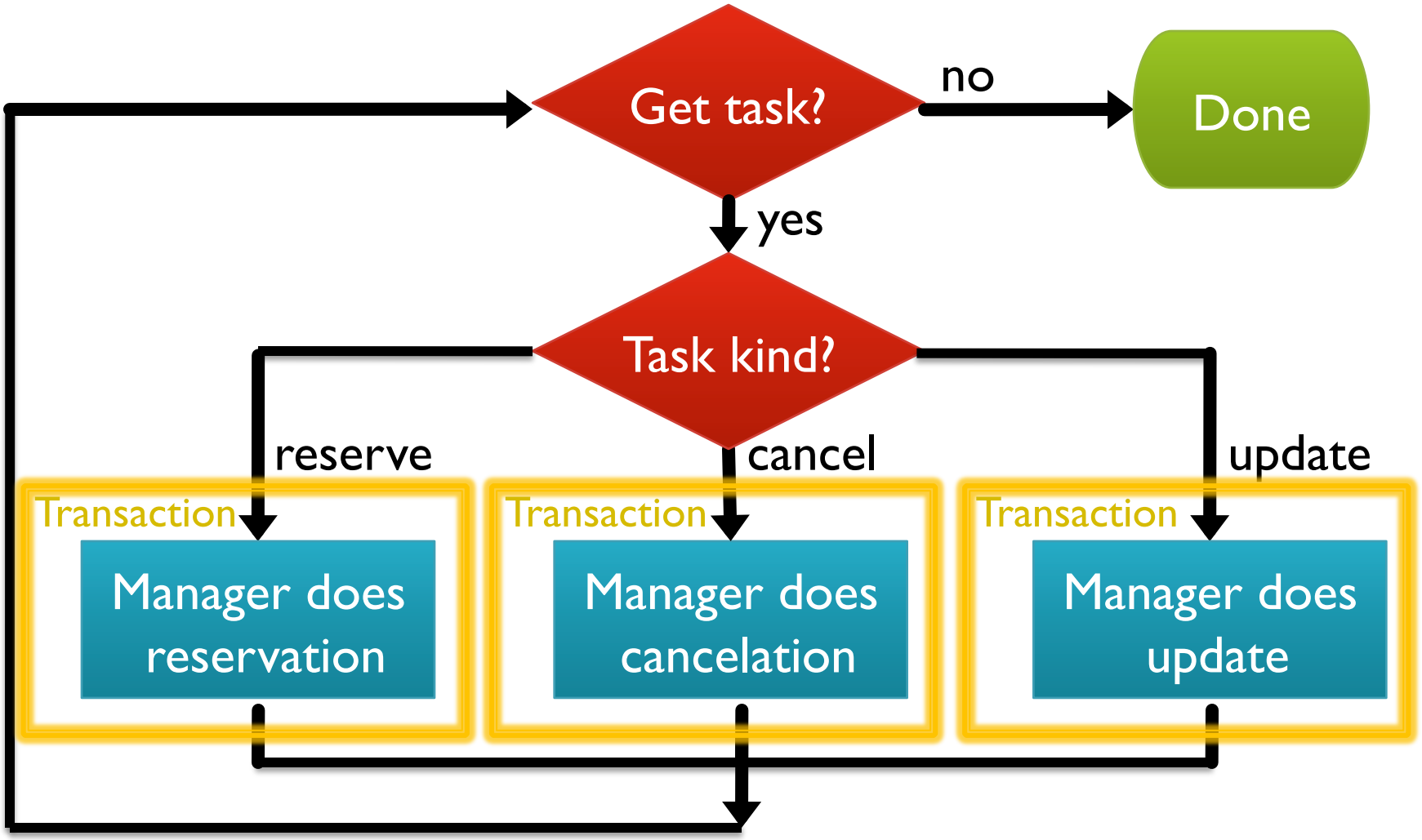


Vacation Description

- Emulates travel reservation system
  - Similar to 3-tier design in SPECjbb2000



Vacation Algorithm



STAMP Characterization

Application	Per Transaction				Time in Transactions
	Instructions	Reads	Writes	Retries	
bayes	60584	24	9	0.59	83%
genome	1717	32	2	0.14	97%
intruder	330	71	16	3.54	33%
kmeans	153	25	25	0.81	3%
labyrinth	219571	35	36	0.94	100%
ssca2	50	1	2	0.00	17%
vacation	3161	401	8	0.02	92%
yada	9795	256	108	2.51	100%

STAMP Summary

- First comprehensive benchmark suite for TM
  - Meets *breadth*, *depth*, and *portability* requirements
  - Useful tool for analyzing TM systems (including SigTM)

- Public release: `http://stamp.stanford.edu`
  - Early adopters:
    - Industry: Microsoft, Intel, Sun, & more
    - Academia: U. Wisconsin, U. Illinois, & more
  - TL2-x86 STM

Outline

- Background & Motivation
- STAMP: Benchmark suite for TM
- SigTM: Effective hybrid TM
  - Fast, flexible, low-cost
  - Predictable semantics
- Conclusions

Hardware vs. Software TM

- HTM: HW does write versioning & conflict detection
  - Advantages:
    - High performance
    - Predictable semantics
  - Disadvantages:
    - Expensive (e.g., requires cache modifications)
    - Inflexible (e.g., fixed capacity for write versioning)
- STM: SW does write versioning & conflict detection
  - Advantages:
    - Low-cost
    - Easy to change and evolve
  - Disadvantages:
    - High overhead
    - Unpredictable semantics

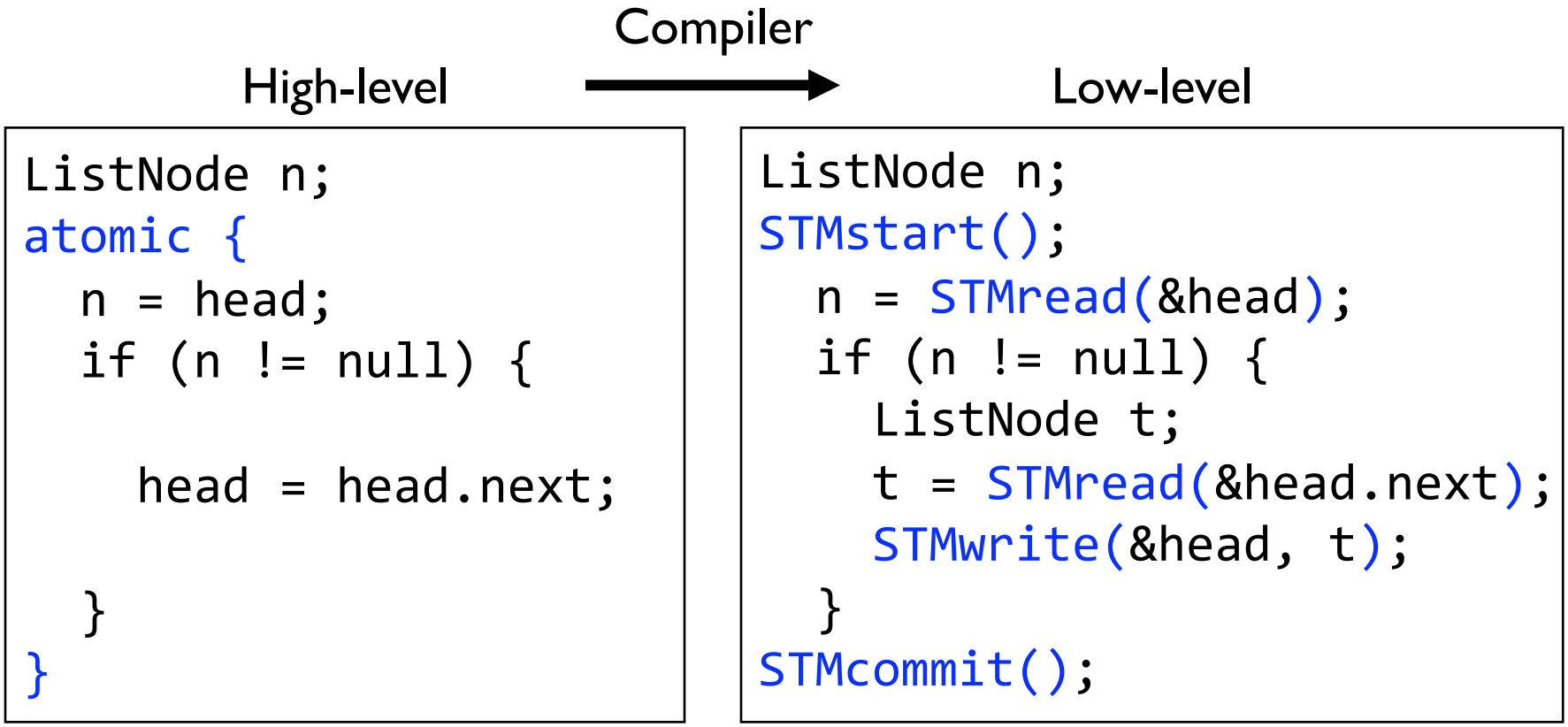
Signature-Accelerated TM (SigTM)

- Hybrid hardware and software TM design
  - Fast, flexible, cost-effective
  - Predictable semantics
- Design approach:
  - Start with software transactions → flexible & cost-effective
  - Add hardware ("signatures") to accelerate → fast
    - Also provides predictable semantics

	HTM	STM	SigTM
Write versioning	HW	SW	SW
Conflict detection	HW	SW	HW

Software Transactions

- Program: atomically remove head of linked-list



STMstart

- Called at transaction start → init transaction meta data

```
STMstart() {
  checkpoint(); // used to rollback
  other_initialization();
}
```

- Constant total cost per transaction
- Expensive only for short transactions

STMread

- Called to read shared data → add to read-set

```
STMread(addr) {
  if (addr in WriteSet) // get latest value
    return WriteSet.getValue(addr);

  ReadSet.insert(addr);

  return *addr;
}
```

- Building read-set is expensive
- Total cost per transaction varies
  - Locality of read accesses, size of read-set, transaction length

STMwrite

- Called to write shared data → add to write-set

```
STMwrite(addr, val) {
  WriteSet.insert(addr, val);
}
```

- Total cost per transaction varies
  - Locality of write accesses, size of write-set, transaction length
- Less cost than STMread (# reads ≥ # writes)

STMcommit

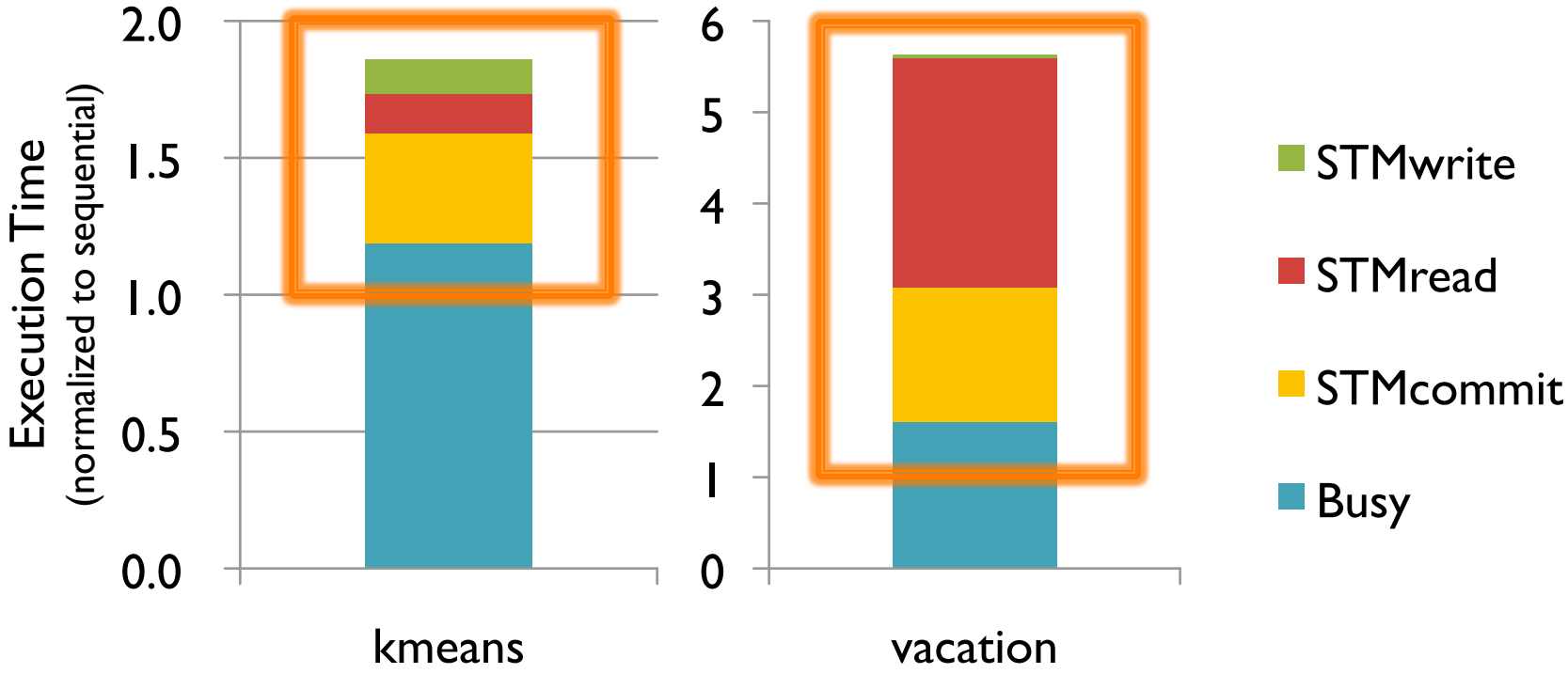
- Called at transaction end → atomically commit changes

```
STMcommit() {
  foreach (addr in WriteSet)    // write set scan 1
    lock(addr);
  foreach (addr in ReadSet)     // read set scan
    validate(addr);             // someone wrote?
  foreach (addr in WriteSet)    // write set scan 2
    *addr = WriteSet.getValue(addr);
  foreach (addr in WriteSet)    // write set scan 3
    unlock(addr);
}
```

- Expensive: scan read-set (1x); scan write-set (3x)

How Slow Are SW Transactions?

- Measured single-thread STM performance

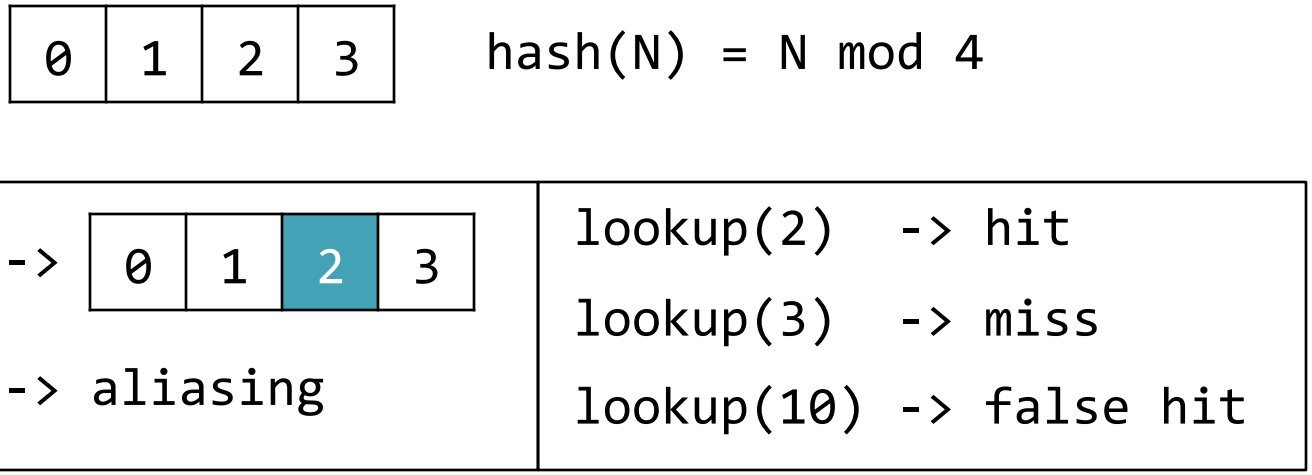


- 1.8x – 5.6x slowdown over sequential
- Hybrid TM should focus on `STMread` and `STMcommit`

SigTM Hardware

- Each HW thread has 2 HW signatures (read & write)
  - Each signature implemented by a Bloom filter
    - Fixed-size bit array with set of hash functions
  - No other HW modifications (e.g., no extra cache bits)

- Operations on signature (Bloom filter): insert & lookup



SigTM Hardware (continued)

- How SigTM uses its signatures:
  - Tx read/write → insert address into read/write signature
  - Coherence messages → look up address in signature
    - Enabled/disabled by software
- If lookup hits in signature, either:
  - Trigger SW abort handler (conflict detection)
  - NACK remote request (atomicity & isolation enforcement)
- Signatures may generate false conflicts
  - Performance but not correctness issue
  - Reduce with longer signatures & better hash functions
- With this HW, how does the SW change?

SigTMread

```
SigTMread(addr) {
  if (addr in WriteSet) // get latest value
    return WriteSet.getValue(addr);
  read_sig_insert(addr); // 1 instruction
  return *addr;
}
```

- No need to build SW read-set
  - Replaced by read signature
- Read signature provides continuous validation
  - Snoops coherence messages & any hits cause abort

SigTMcommit

```
SigTMcommit() {
  enable_write_sig_lookup();
  foreach (addr in WriteSet) // write set scan 1
    fetch_exclusive(addr);

  enable_write_sig_nack();
  foreach (addr in WriteSet) // write set scan 2
    *addr = WriteSet.getValue(addr);
  disable_write_sig_lookup();
}
```

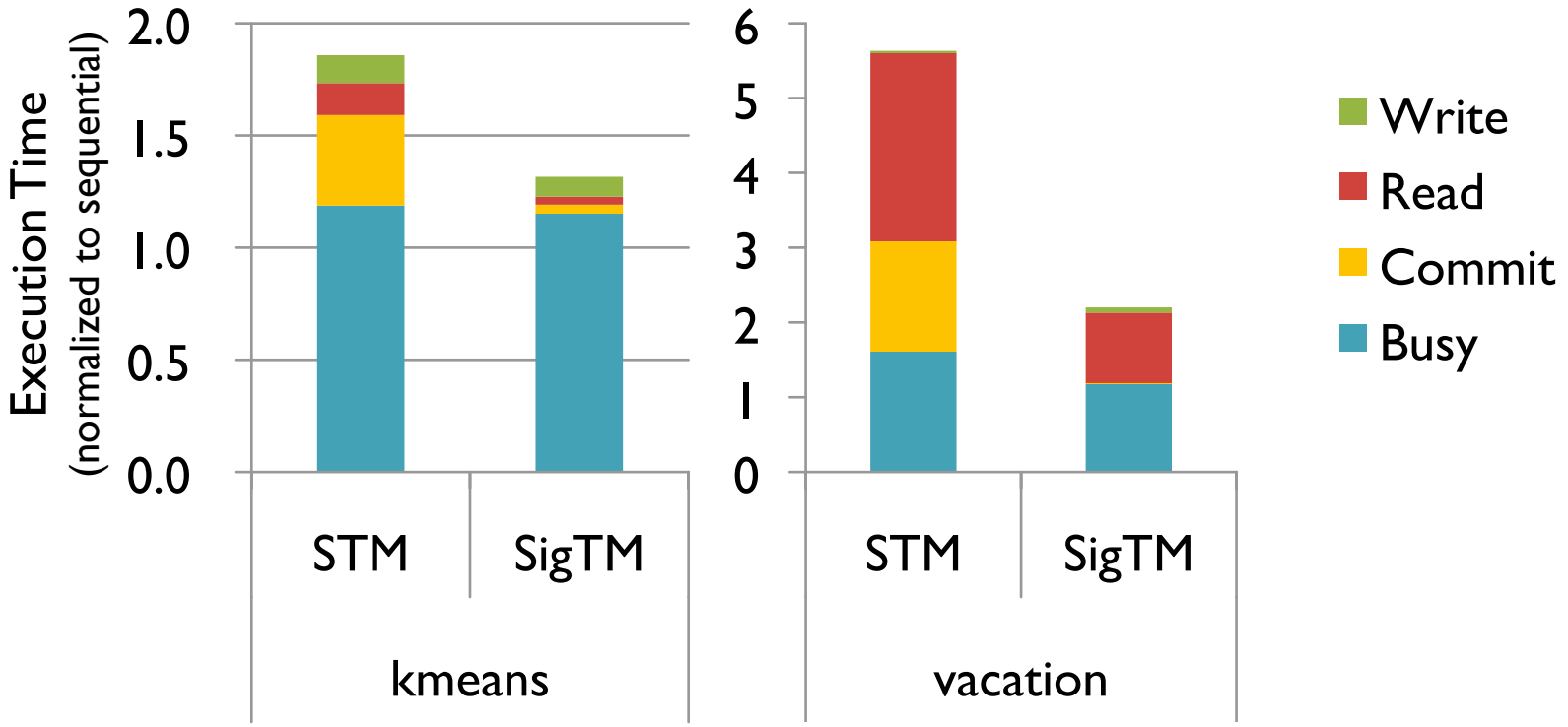
- Read signature eliminates scan of read-set to validate
- Write signature eliminates locks
  - Snoops coherence messages & NACKs any hits
- Two write-set scans instead of three

Experimental Setup

- Execution-driven simulation
  - 1–16 core x86 chip-multiprocessor with MESI coherence
  - Supports HTM, STM, and SigTM
- Used STAMP benchmark suite for evaluation
- Three experiments:
  - Does SigTM reduce the overhead of SW transactions?
  - How fast is SigTM?
  - How much hardware does SigTM cost?

How Much Smaller is the Overhead?

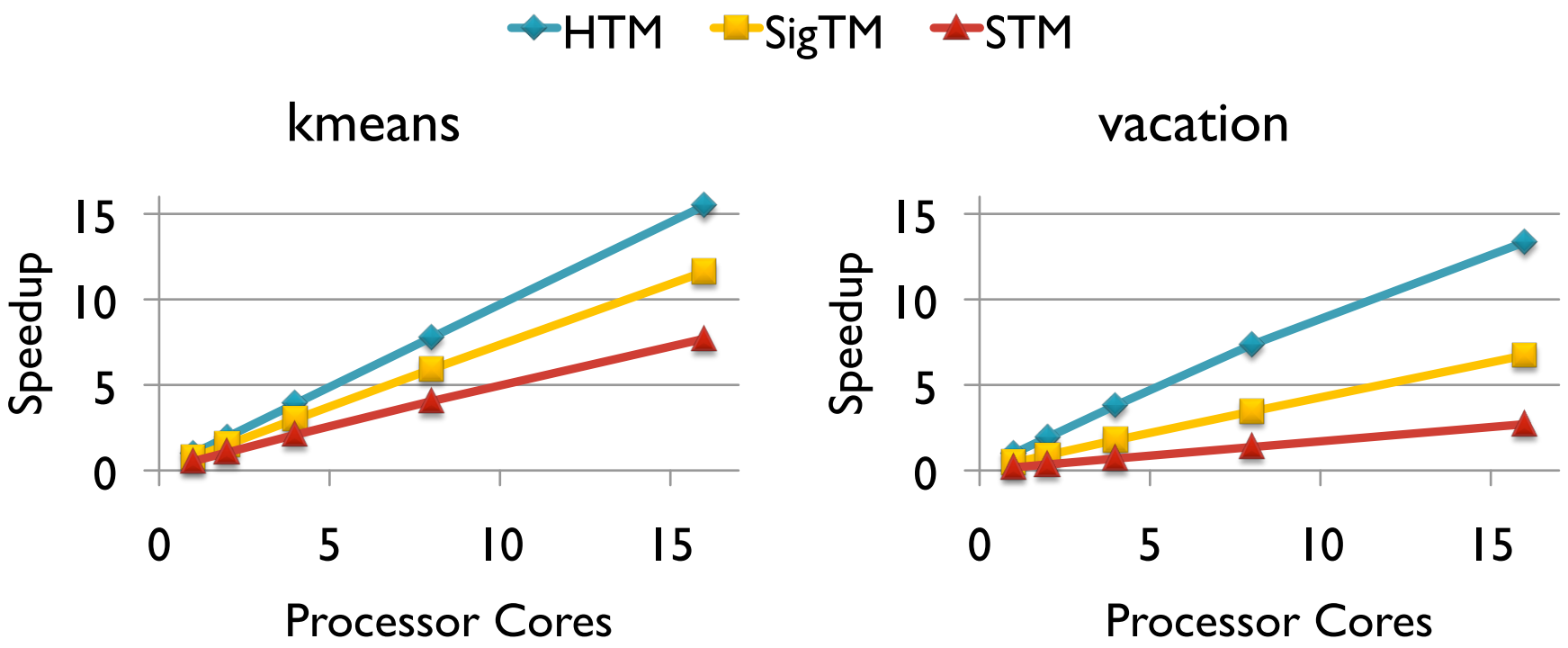
- Measured single-thread performance on STM and SigTM



- SigTM effectively accelerates read & commit

How Fast is SigTM?

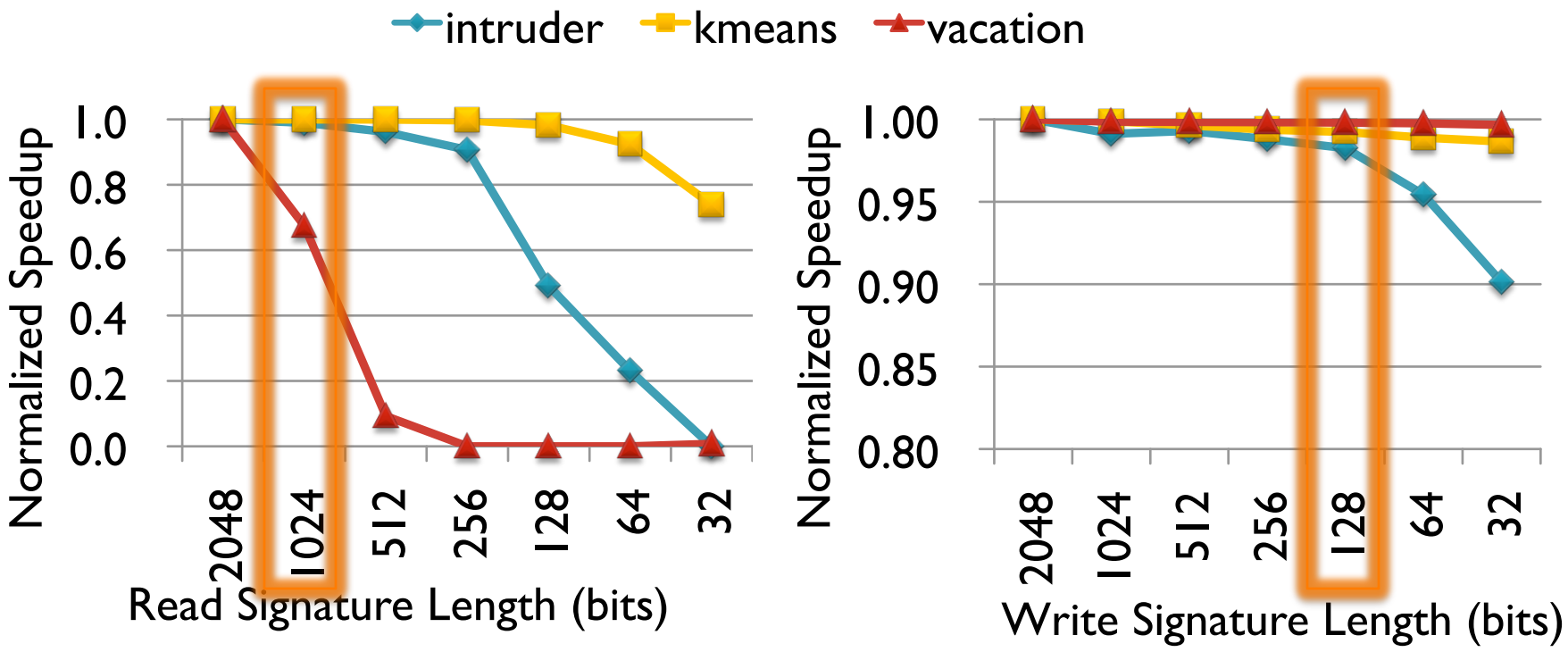
- Measured speedup on 1–16 cores



- In general, SigTM faster than STM but slower than HTM

How Much Hardware Does it Cost?

- Measured performance drop as signatures get shorter



- Recommend 1024 bits for read sig, 128 bits for write sig

Outline

- Background & Motivation
- STAMP: Benchmark suite for TM
- SigTM: Effective hybrid TM
  - Fast, flexible, low-cost
  - Predictable semantics
- Conclusions

Example Program: Privatization

Thread 1

```
ListNode n;
atomic {
  n = head;
  if (n != null)
    head = head.next;
}
// use n.val many times
```

Thread 2

```
atomic {
  ListNode n = head;
  while (n != null) {
    n.val++;
    n = n.next;
  }
}
```

- Two acceptable outcomes:
  - T1 commits first; T1 uses only non-incremented `n.val`
  - T2 commits first; T1 uses only incremented `n.val`
- Works correctly with lock-based synchronization
  - Race-free program

Unpredictable Results with STM?

Thread 1

```
ListNode n;
atomic {
  n = head;
  if (n != null)
    head = head.next;
}
// use n.val many times
```

Thread 2

```
atomic {
  ListNode n = head;
  while (n != null) {
    n.val++;
    n = n.next;
  }
}
```

- All STMs may give unexpected results
  - T1 may use both old & new value after privatization
- Cause: Non-transactional accesses are not instrumented
  - Non-Tx writes do not cause Tx to abort
  - Tx commit not atomic with respect to non-Tx accesses

Strong Isolation

- Definition: Transactions isolated from non-Tx accesses
- HTM → inherent strong isolation
  - Non-Tx cause coherence messages
  - Conflict detection mechanism enforces strong isolation
- STM → supplemented strong isolation
  - Additional annotations needed for non-Tx accesses
  - Some can be optimized but still a source of overhead
- SigTM → inherent strong isolation
  - Without additional instrumentation or overhead

How SigTM Provides Strong Isolation

- STMs have unpredictable results because:
  - Non-Tx writes do not cause transactions to abort
  - Tx commit not atomic with respect to non-Tx accesses
- Non-Tx writes cause SigTM to abort a transaction
  - Coherence messages looked up in read signature
  - Hits in read signature trigger transaction abort
- SigTM commit is atomic with respect to non-Tx accesses
  - Write signature used to provide atomic writeback
  - Coherence messages looked up in write signature
  - Hits in write signature → NACK non-Tx accesses

Conclusions

- TM is promising for simplifying parallel programming

My contributions to the TM community:

- STAMP
  - Comprehensive benchmark suite for TM
  - Public release: `http://stamp.stanford.edu`
  - Early adopters: MSFT, Intel, U. Wisconsin, U. Illinois, & more
- Signature-Accelerated TM (SigTM)
  - Hardware acceleration of software transactions
  - Fast, flexible, cost-effective, & predictable semantics
  - Attractive design for industry

Thank you

- Committee
  - Advisor: Christos Kozyrakis
  - Co-advisor: Kunle Olukotun
  - Brad Osgood, Subhasish Mitra
- Family
  - Chanh, Ling Ling, Lyly
- TCC research group
  - Austen, Brian, JaeWoong, Jared, Martin, Nathan, Nju, Sewook, Tayo, Woongki
  - Darlene Hadding, Teresa Lynn
- Joseph and Hon Mai Goodman
- Friends
  - AAGSA, SVSA, & many more