ByteSTM: Virtual Machine-Level Java Software Transactional Memory

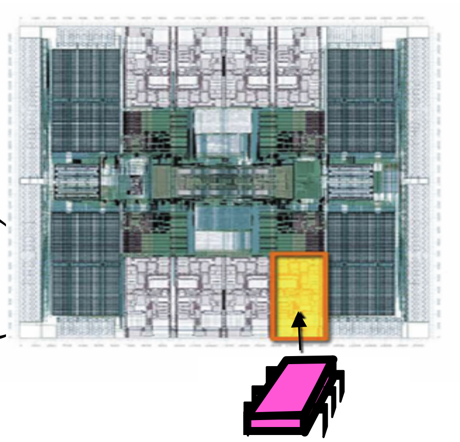
Mohamed Mohamedin, Binoy Ravindran, and Roberto Palmieri
Virginia Tech
USA
{mohamedin,binoy,robertop}@vt.edu

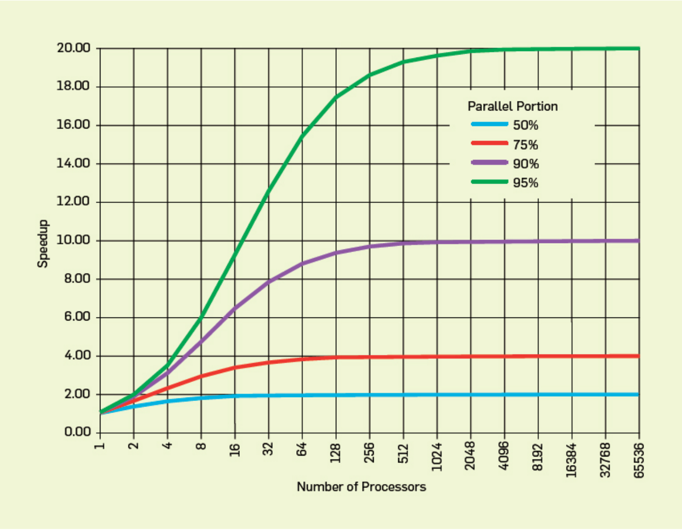
Coordination 2013

Concurrency control on chip multiprocessors significantly affects performance (and programmability)

- Improve performance by exposing greater concurrency
  - *Amdahl's law: relationship between sequential execution time and speedup reduction is not linear*

Sun T2000 Niagara (8-core)





Lock-based concurrency control has serious drawbacks

- Coarse grained locking
  - Simple
  - But no concurrency

```
public boolean add(int item) {
  Node pred, curr;
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
  } finally {
    lock.unlock();
  }
}
```

Fine-grained locking is better, but…

- Excellent performance
- Poor programmability
- Lock problems don’t go away!
  - Deadlocks, livelocks, lock-convoying, priority inversion,….
- Most significant difficulty – composition

```
public boolean add(int item) {
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.val < item) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Lock-free synchronization overcomes some of these difficulties, but…

“lock-free retry loop”

```
public boolean add(int item) {
 while (true) {
   Node pred = null, curr = null, succ = null;
   boolean[] marked = {false};  boolean snip;
   retry: while (true) {
     pred = head; curr = pred.next.getReference();
     while (true) {
       succ = curr.next.get(marked);
       while (marked[0]) {
         snip = pred.next.compareAndSet(curr, succ, false, false);
         if (!snip) continue retry;
          curr = succ; succ = curr.next.get(marked);
       }
       if (curr.val < item)
            pred = curr; curr = succ;
     }
   }
   if (curr.val == item) {  return false;
   } else {
     Node node = new Node(item);
     node.next = new AtomicMarkableReference(curr, false);
     if (pred.next.compareAndSet(curr, node, false, false)) {return true;}
   }
 }
}
```

Transactional memory

- Like database transactions
- ACI properties (no D)
- Easier to program
- Composable

- First HTM, then STM, later HyTM

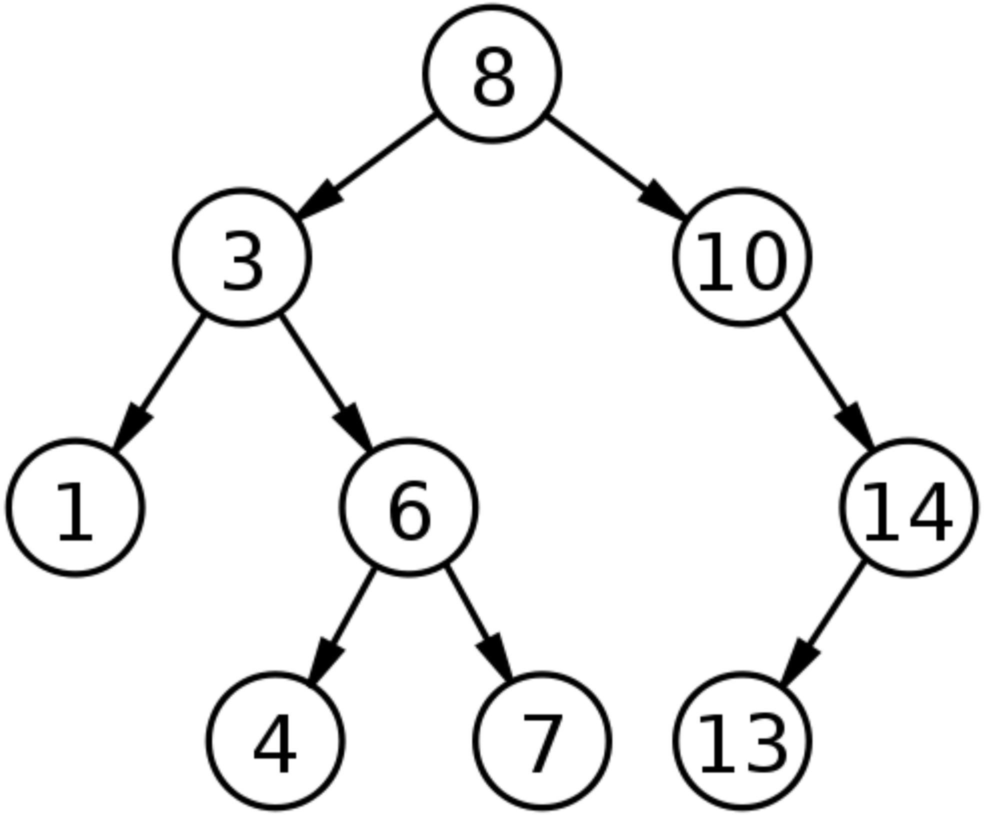
```
public boolean add(int item) {
  Node pred, curr;
  atomic {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
}
```


M. Herlihy and J. B. Moss (1993). Transactional memory: Architectural support for lock-free data structures. *ISCA*. pp. 289–300.

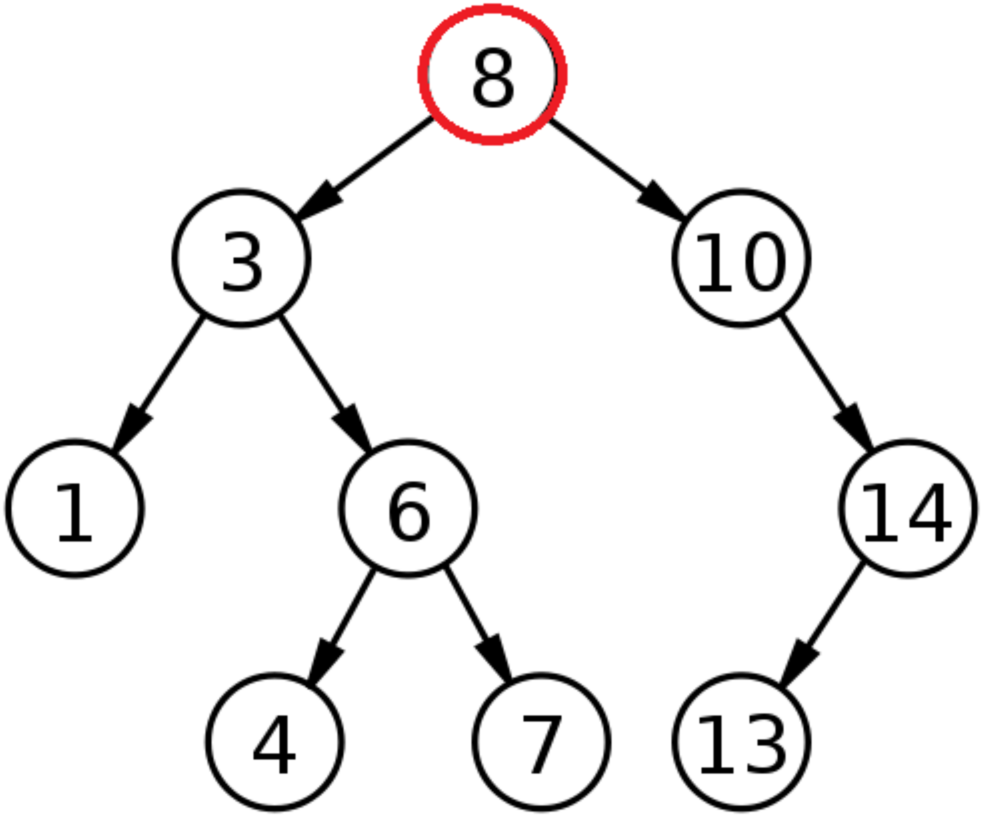
N. Shavit and D. Touitou (1995). Software Transactional Memory. *PODC*. pp. 204—213.

How TM works?

- Optimistic concurrency
- Example: Adding 9 & 15 concurrently



Thread A adds 9 & Thread B adds 15



Thread A

Read-set: 8

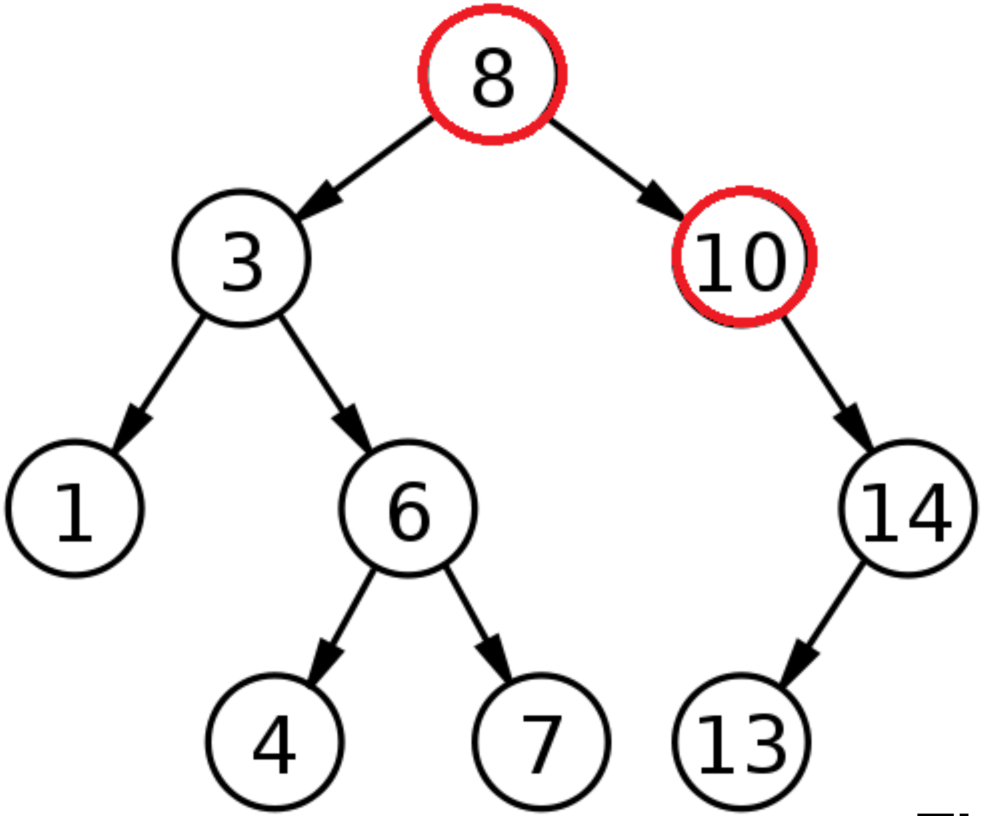
Write-set:

Thread B

Read-set: 8

Write-set:

Thread A adds 9 & Thread B adds 15



Thread A

Read-set: 8, 10

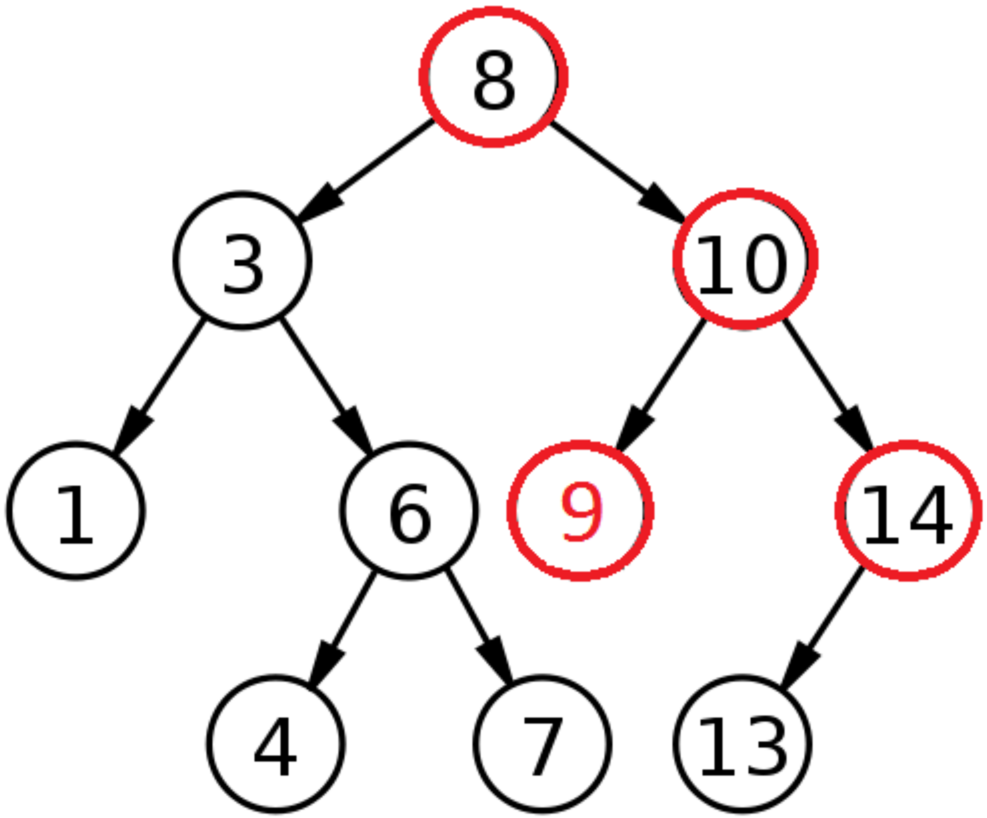
Write-set:

Thread B

Read-set: 8, 10

Write-set:

Thread A adds 9 & Thread B adds 15



Thread A

Read-set: 8, 10

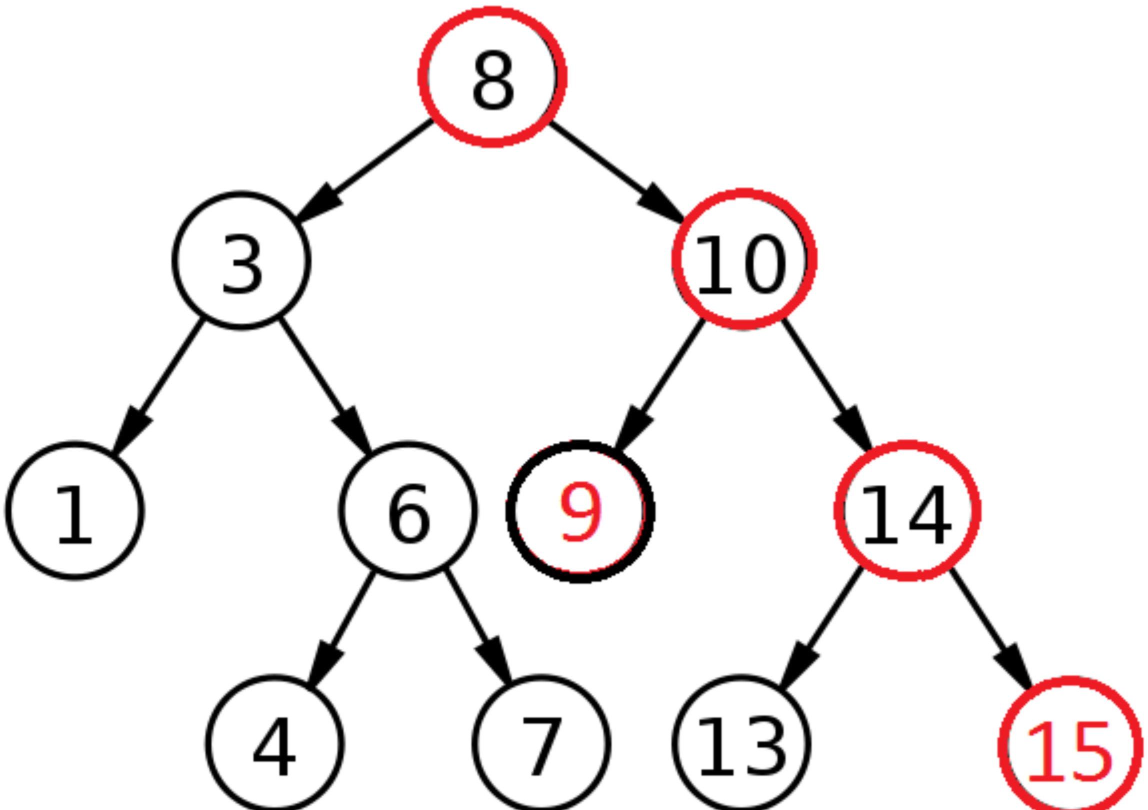
Write-set: 10 (left child pointer)

Thread B

Read-set: 8, 10, 14

Write-set:

Thread A adds 9 & Thread B adds 15



Thread A

Read-set: 8, 10

Write-set: 10 (left child pointer)

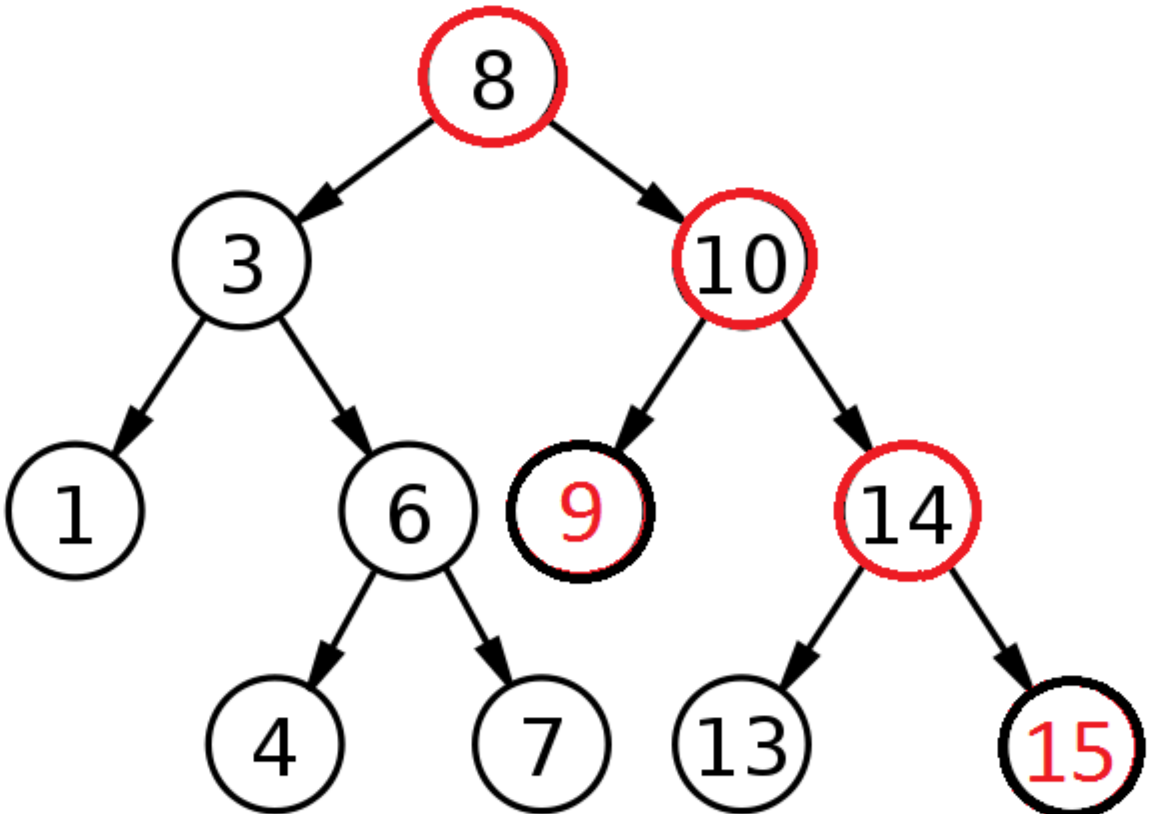
Committed successfully

Thread B

Read-set: 8, 10, 14

Write-set: 14 (right child pointer)

Thread A adds 9 & Thread B adds 15



Thread A

Read-set: 8, 10

Write-set: 10 (left child pointer)

Committed successfully

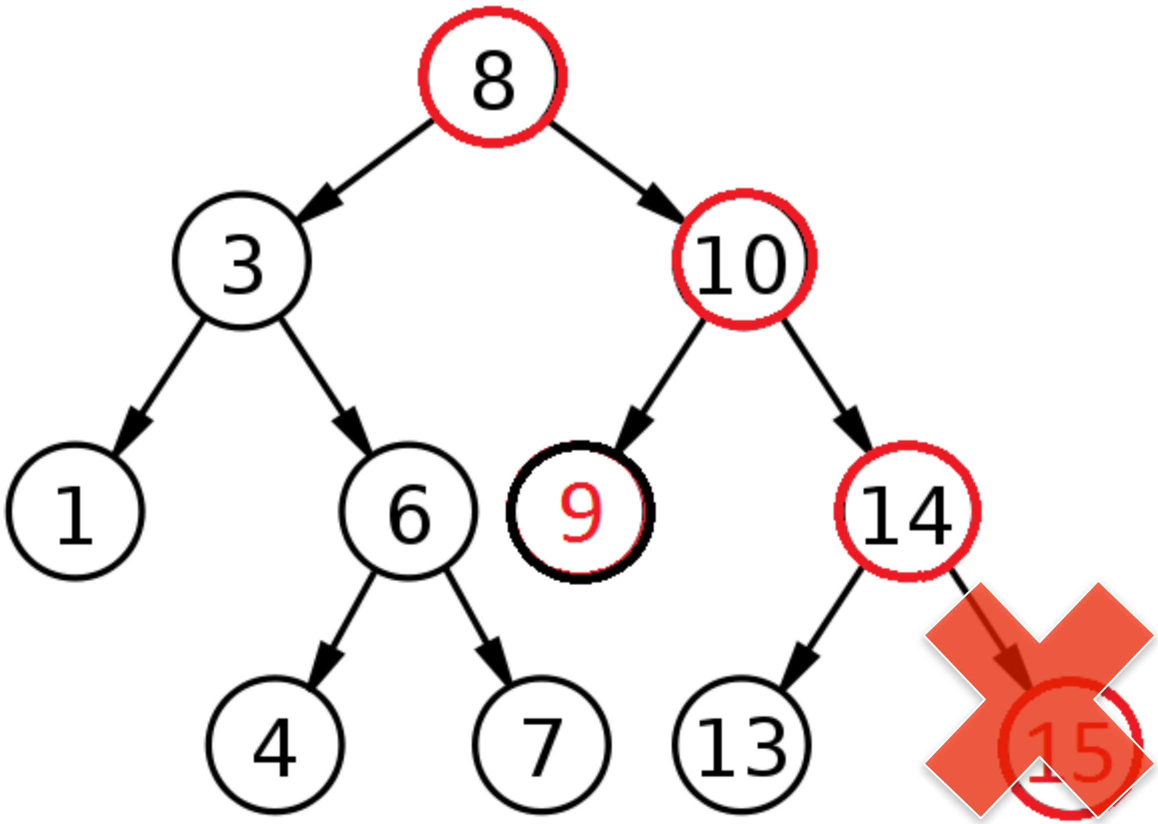
Thread B

Read-set: 8, 10, 14

Write-set: 14 (right child pointer)

Committed successfully

Object-based granularity



Thread A

Read-set: 8, 10

Write-set: **10** ←WAR→

Committed successfully

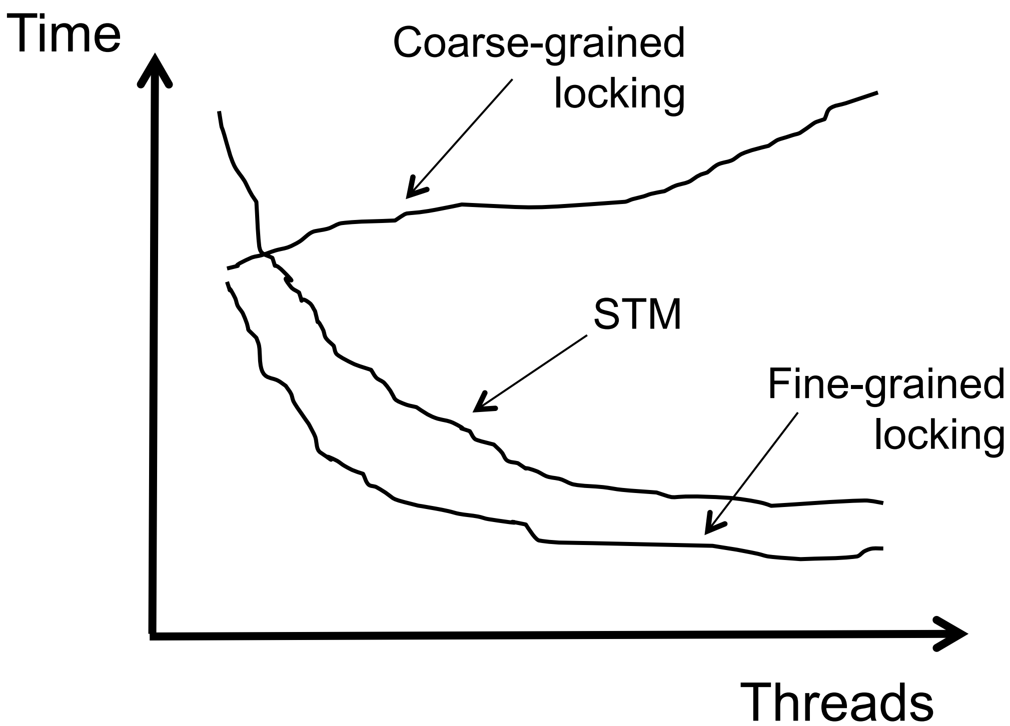
Thread B

Read-set: 8, **10**, 14

Write-set: 14

Conflict → Abort

Optimistic execution yields performance gains at the simplicity of coarse-grain, but no silver bullet



- High data dependencies
- Irrevocable operations
- Interaction between transactions and non-transactions
- Conditional waiting
- ……

E.g., C/C++ Intel Run-Time System STM (B. Saha et. al. (2006). McRT-STM: A High Performance Software Transactional Memory. *ACM PPoPP*)

Three key mechanisms needed to create atomicity illusion

Versioning

```
atomic{
    x = x + y;
}
```

Where to store new `x` until commit?

- *Eager*: store new `x` in memory; old in *undo log*
- *Lazy*: store new `x` in *write buffer*

Conflict detection

T0

```
atomic{
    x = x + y;
}
```

T1

```
atomic{
    x = x / 25;
}
```

How to detect conflicts between `T0` and `T1`?

- Record memory locations read in *read set*
- Record memory locations wrote in *write set*
- Conflict if one's read or write set intersects the other's write set

Third mechanism is contention management

T0

```
x = x + y;
```

T1

```
x = x / 25;
```

```
x = x / 25;
```

Which transaction to abort?

- Greedy: favor those with an earlier start time
- Karma: ….

STM implementations can be broadly classified

- Library-based
  - No changes to the language
  - Both explicit and implicit transactions
  - E.g., Deuce (MultiProg 10)
- Compiler-based
  - Adds new language constructs
  - Implicit transactions
  - E.g., Intel® C++ STM Compiler, GCC 4.7
- Virtual machine-based
  - Implicit transactions supported through bytecode instructions
    - Either with compiler support (like HTM) or by special marker functions
  - Relatively less studied
  - E.g., ByteSTM, Harris & Fraser (OOPSLA 03)

Motivations for VM-based STM

- Direct memory access
- Full control over garbage collector (GC)
- Full control over bytecode instruction behavior
- Can manipulate thread’s header
- HTM-compatible

ByteSTM

- Built by modifying Jikes RVM (v3.1.2) Optimizing Compiler
  - Jikes RVM is a research JVM written in Java
  - Jikes RVM has no interpreter and bytecode must be compiled first to native code
  - Two types of compilers
    - Baseline compiler: fast compilation but with no optimizations
    - Optimizing compiler: better performance (register allocation, inlining, code reordering,…)
- ByteSTM instrumentation exists in bytecode-to-native code compilation

ByteSTM: programming interface

- Implicit transaction

```
atomic{
    A = B;
    B++;
}
```

Or:

```
stm.STM.xBegin();
    A = B;
    B++;
stm.STM.xCommit();
```

Implicit transaction
(e.g., ByteSTM)

```
Transaction T;
T.begin();
do{
    A.txWrite(B.txRead());
    B.txWrite(B.txRead() + 1);
} while( !T.commit());
```

Explicit transaction
(e.g., RSTM's explicit transaction)

ByteSTM: data types

- No special transactional instructions
  - Bytecode instructions have two modes
    - Transactional
    - Non-transactional
  - Two new bytecode instructions only (xBegin and xEnd)
  - One copy of code
  - Behavior added by modifying the bytecode-to-native code compiler
- Works on all data types
  - Memory access is monitored at bytecode instruction level
- Supports external libraries inside transactions

ByteSTM: program state save/restore

- Atomic blocks anywhere in the code
  - Saves program state at transaction start
    - Stack pointer, registers, local variables
    - Leverage Java’s exception mechanism plus saving local variables
  - Restores the saved state when transaction aborted
  - Only non-local variables are monitored
    - Rely on Java-to-bytecode compiler’s special instructions for local and non-local variables

```
@Atomic
void  method(int c){
  c = c / 2;
  a = c
}
//Java annotation
//Deuce, LSA-STM
```

```
int  c=10;
c  = a + 5;
atomic{
   c = c / 2;
   a = c;
}
```

```
saveLocalVariables();
do {
    try{
          xBegin();
          //transaction body
          xEnd();
          break;
    }catch(STMException e) {
          restoreLocalVariales();
    }
} while(true);
```

ByteSTM: memory model

- Direct memory access
  - Faster write back

- Raw memory model
  - One code to handle all cases
  - Moving GC compatible (absolute address is not used)

Instance field: Object address + field offset
Static field: Static memory address + field offset
Array element: Array address + element size x element index

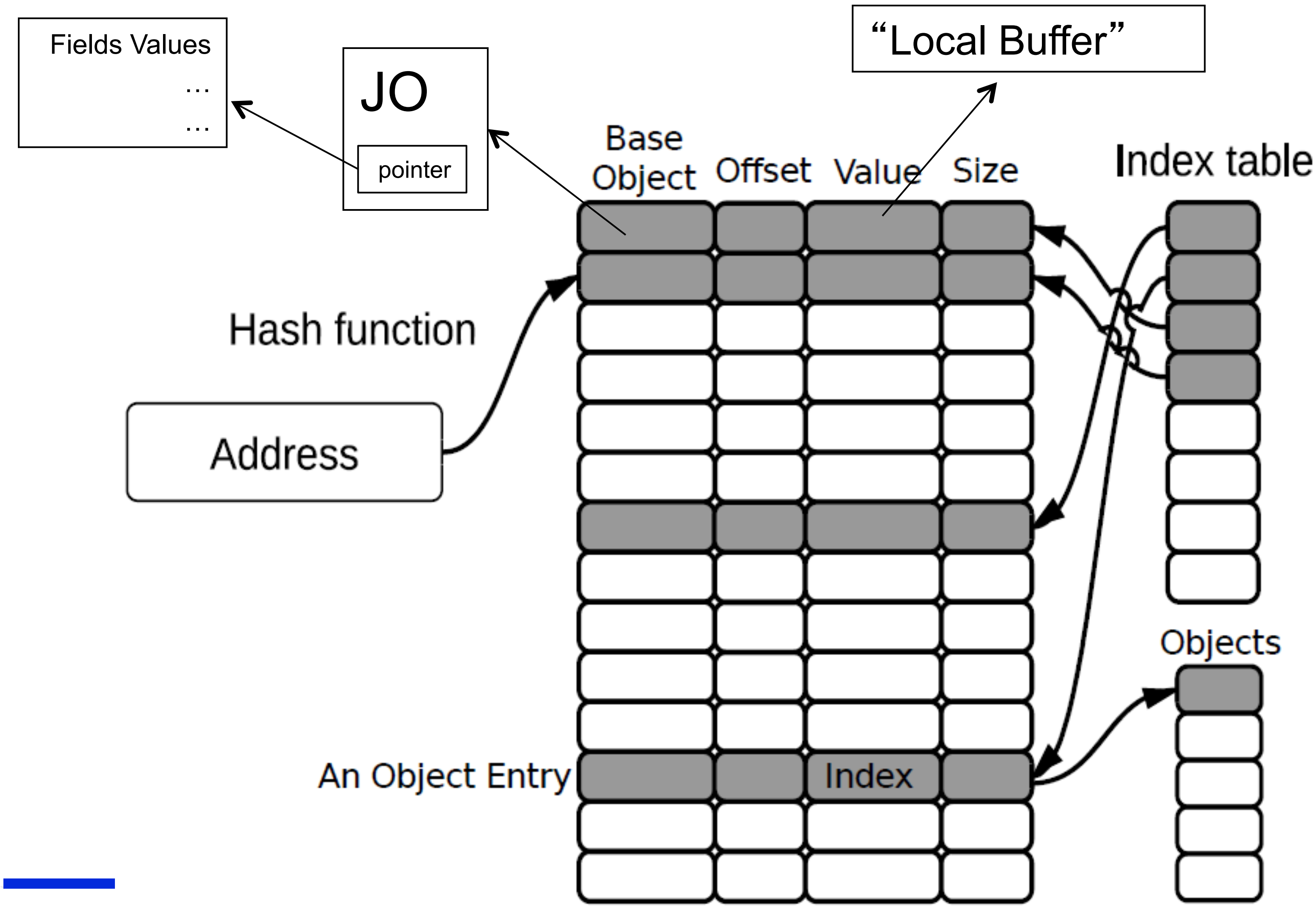
Absolute address

	Data Type	Base Object	offset	Value	Size
Obj1.x	int	Obj1	0	20	4
Obj1.y	double	Obj1	4	46	4
Obj2.obj	Object (reference)	Obj2	0	0 (index)	4

Raw memory model

ByteSTM: write-set representation

- Arrays of primitive + open addressing hashing



ByteSTM: GC issues

- Metadata in the thread header
  - Faster than Java standard ThreadLocal

- GC issues
  - Manually allocates and recycles memory for transactional metadata; reduces GC overhead
    - Jikes RVM immortal memory
  - Since write-set includes object references, they are not GCed
    - At commit-time, we can write-back (otherwise, objects won't exist!)

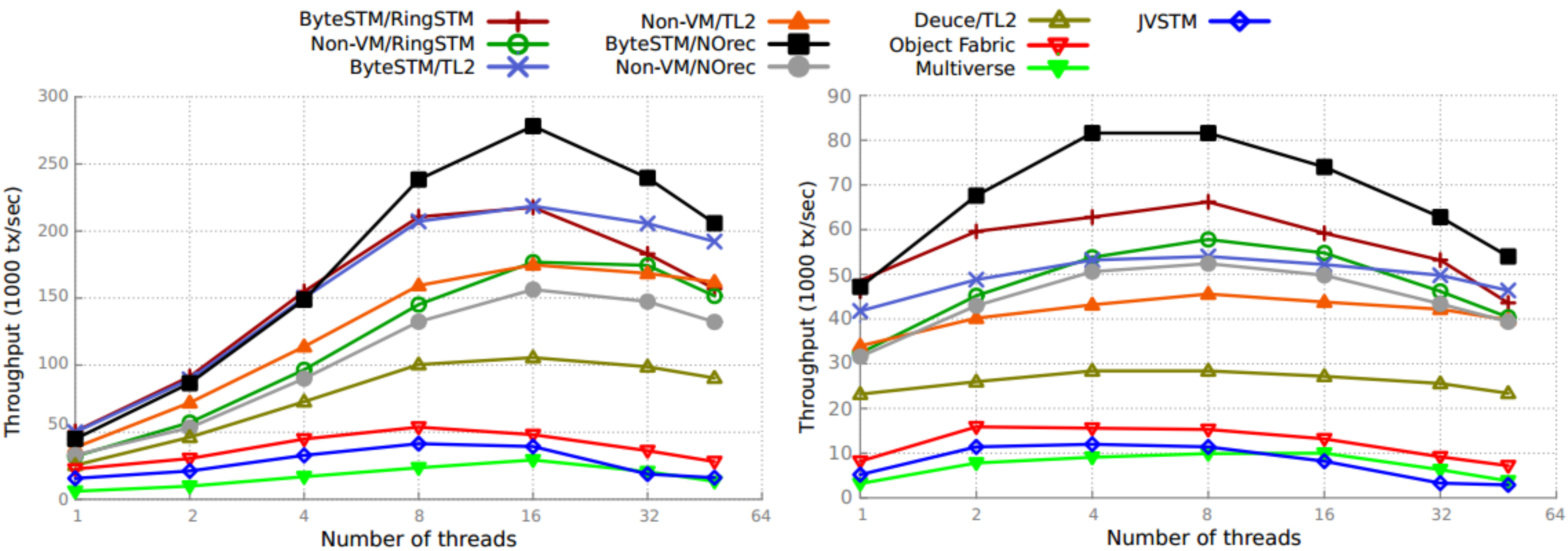
Summary and contrast

	Library-based						Compiler-based	VM-based			
Feature	Deuce [17]	JVSTM [5]	ObjectFabric [21]	DSTM2 [14]	Multiverse [26]	LSA-STM [23]	AtomJava [15]	Harris & Fraser [12]	Atomos[1] [7]	Trans. monitors [27]	ByteSTM
Implicit transactions	√	√	X	X	X	√	√	√	√	√	√
All data types	√	X	X	X	X	X	√	√	√	√	√
External libraries	√	X	X	X	X	X	√ [2]	√	X [3]	√	√
Unrestricted atomic blocks	X	X	√	√	√	X	√	√	√	√	√
Direct memory access	√[4]	X	X	X	X	X	X	√	√	X	√
Field-based granularity	√	X	X	X	X	X	X	X	X	X	√
No GC overhead	√ [5]	X	X	X	X	X	X	√	√	X	√
Compiler support	X	X	X	X	X	X	√	√	√	√	√ & X [6]
Strong atomicity	X	X	√	X	X	X	√	X	√	X	X
Closed/Open nesting	X	√	√	X	X	X	X	X	√	X	X
Conditional variables	X	X	X	X	X	X	X	X	√	X	X

Experimental Testbed

- Platform
  - 48-core machine (4 AMD Opteron with 12 cores; 700 MHz), 16 GB
  - Ubuntu Linux Server 10.04 LTS 64-bit, JikesRVM v3.1.2
- Benchmarks
  - Micro-benchmarks
    - Linked List, Skip List, Red-black Tree, and Hash set
  - Macro-benchmarks
    - STAMP benchmark (Vacation, KMeans, Genome, Labyrinth, Intruder)
- Competitors:
  - Deuce, JVSTM, ObjectFabric, Multiverse
  - Three STM algorithms: NOrec, RingSTM, TL2
- VM vs. Non-VM
  - Non-VM: same implementation but runs as Deuce plugin
  - Reduces comparison factors and gives fair comparison

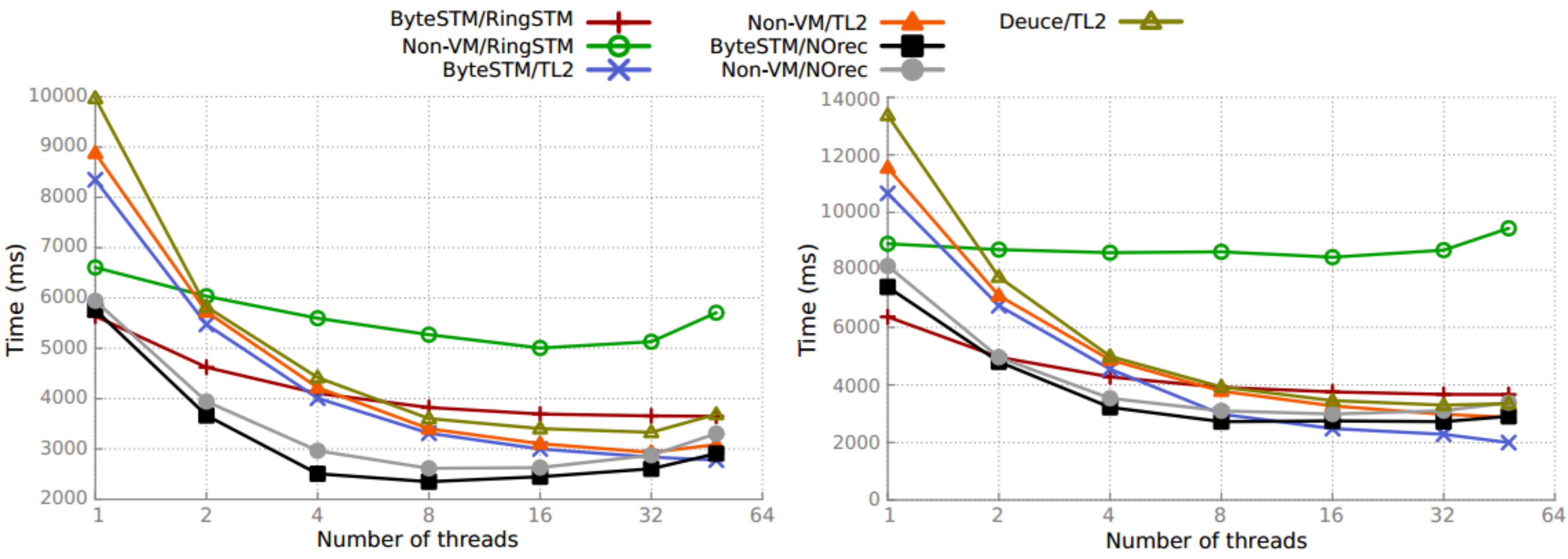
Performance: linked-list



(a) 20% writes.

(b) 80% writes.

Performance: Vacation



(a) Low contention.

(b) High contention.

Conclusions

- Implementing a Java STM at the VM-level yields significant performance benefits
- Micro-benchmarks: 6% to 70%
- Macro-benchmarks: 7% to 60%

- VM-level STM is likely the most performant STM implementation approach for managed languages

- Compile-time optimization specific for STM?
  - STM optimization pass
- STM-aware thread scheduler?