Transactional Memory: Programming Multi-Core Systems

Ruini Xue

School of Computer Science and Engineering

University of Electronic Science and Technology of China

2016

A Trend in Hardware

“The Movement to Multi-core Processors”

- Originates from inability to increase processor clock rate
- Profound impact on architecture, OS and applications

How to program multi-core systems effectively?

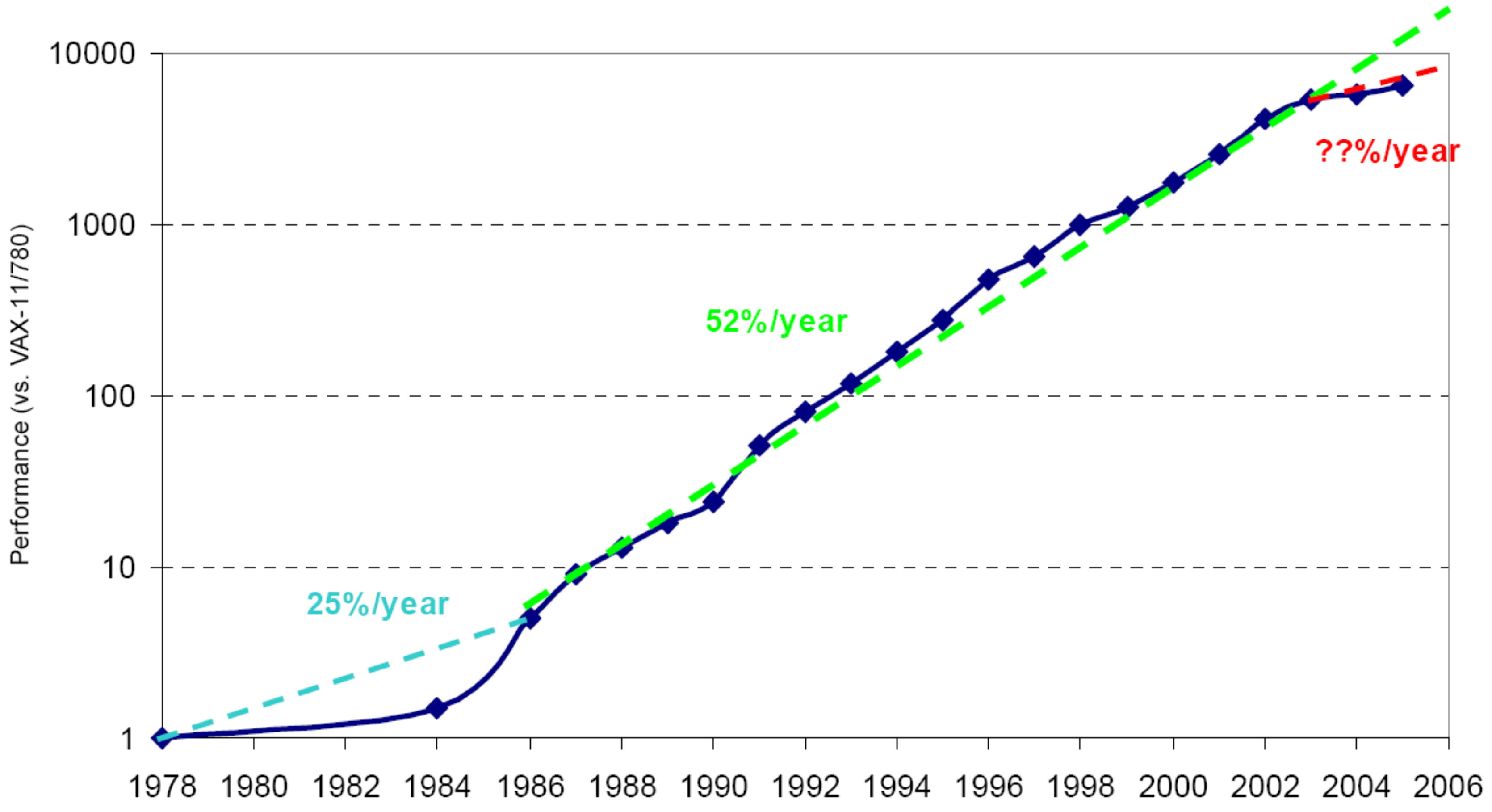
Outline

- Overview of Multi-core
- Paradigms: Lock vs. Transaction
- Transactional Memory

Why Multi-Core?

- Areas of improving CPU performance in last 30 years
    1. Clock speed
    2. Execution optimization/ILP (Cycles-Per-Instruction)
    3. Cache
- All 3 are concurrency-agnostic
- Now
    1. Disappears: no Intel 4GHz CPU
    2. Slows down
    3. Still good

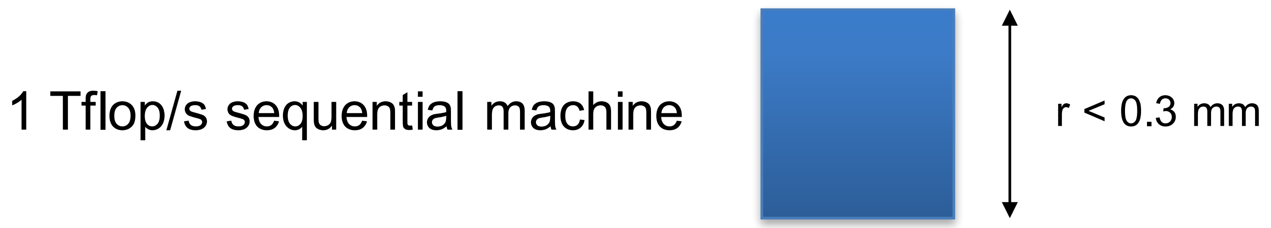
CPU Speed History



From "Computer Architecture: A Quantitative Approach", 4th edition, 2007

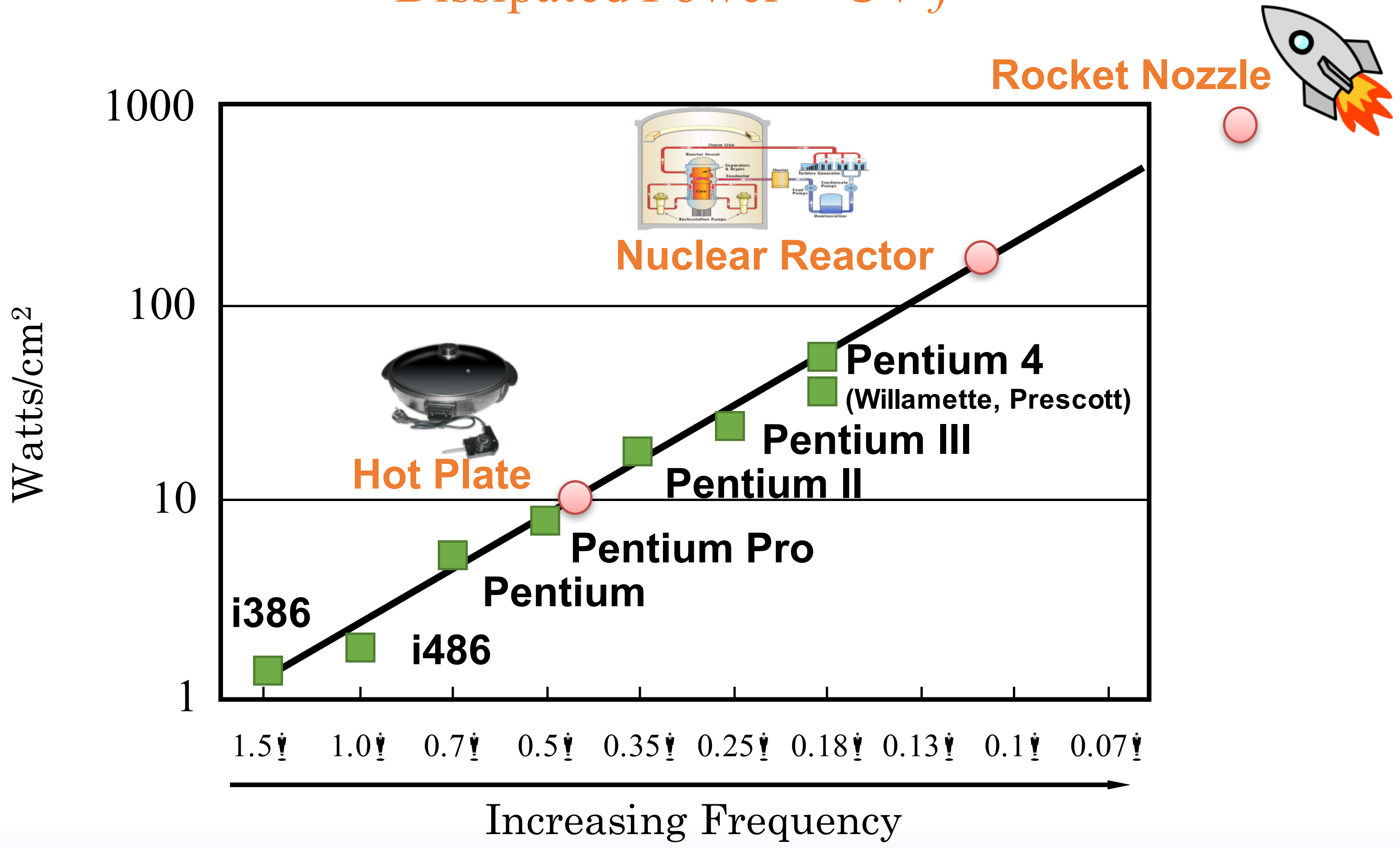
Physical Distance

- Consider the 1 TFLOPS sequential machine:
  - Data must travel some distance, r, to get from memory to CPU.
  - To get 1 data element per cycle, this means 10^{12} times per second at the speed of light, $c = 3\text{x}10^8$ m/s. Thus $r < c/10^{12} = 0.3$ mm.



Power Issue

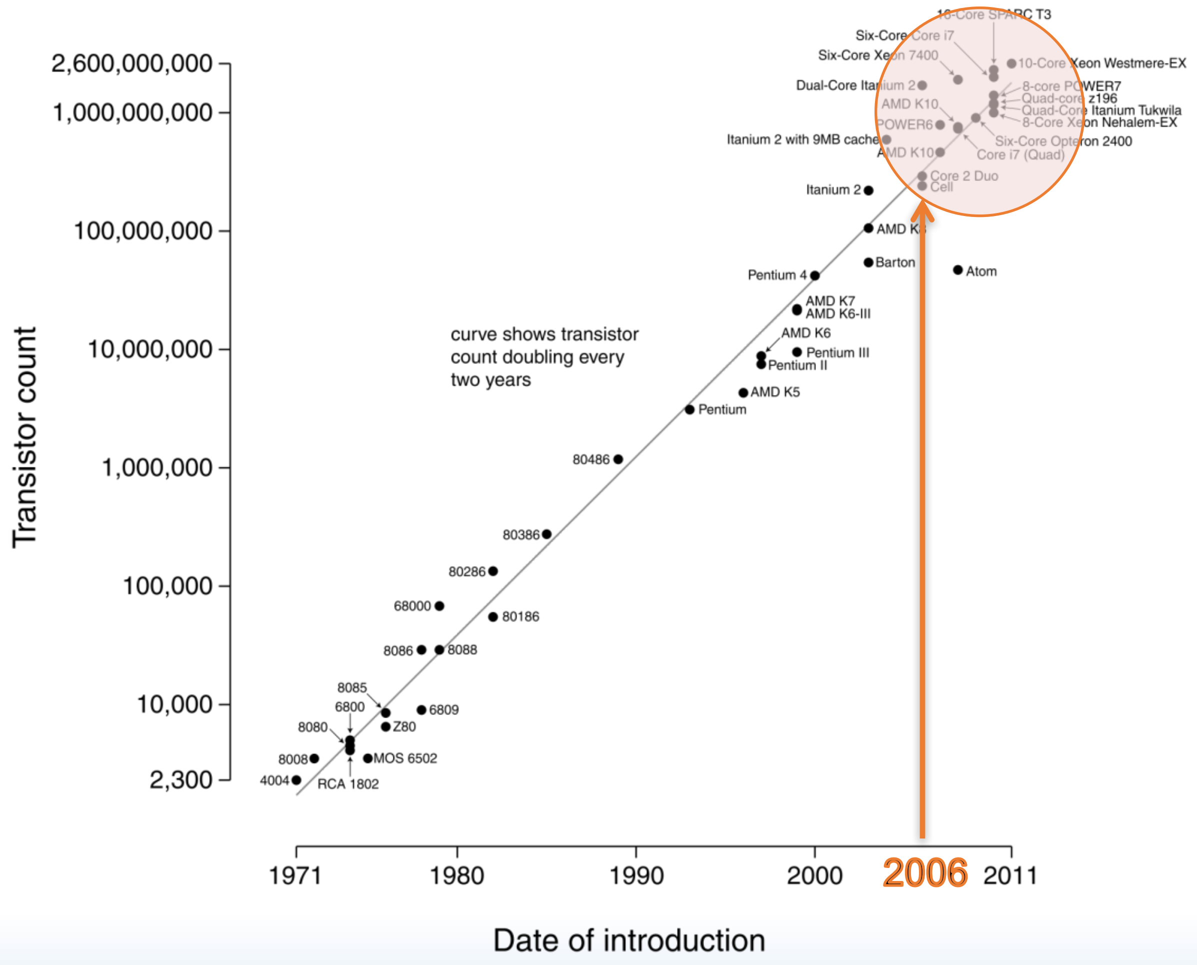
Dissipated Power ~ $CV^2 f$



Source: http://docencia.ac.upc.edu/master/DTM/docs/01-Implications.pdf

Transistor Count

Microprocessor Transistor Counts 1971-2011 & Moore’s Law



Source: Wikipedia: http://en.wikipedia.org/wiki/Moore's_law

Reality Today

- Multicore is the mainstream
- Exploit the performance
  - Let the cores run concurrently.
  - Design concurrent running units
    - multi-threads, multi-processes
- The software must be **concurrent**

"Concurrency is the next revolution in how we write software"
—— Hurb Sutter (2005): The Free Lunch is Over

Costs/Problems of Concurrency

- pthread, MPI, PVM, openMP
- Overhead of locks, message passing
- Not all programs are parallelizable
- **Programming concurrently is HARD**
  - Complex concepts: mutex, read-write lock, queue…
  - Correct synchronization: race, deadlocks…
  - Getting speed-up

Status in Synchronization

- Current mechanism: **manual locking**
    - Organization: lock for each shared structure
    - Usage: (block) ➔ acquire ➔ access ➔ release
- Correctness issues
    - Under-locking ➔ data races
    - Acquires in different orders ➔ deadlock
- Performance issues
    - Difficult to find right granularity
    - Overhead of acquiring vs. allowed concurrency

Transactions / Atomic Sections

- Databases has provided automatic concurrency control for 30 years: ACID transactions
- A transaction is a finite sequence of machine instructions executed by a single process, that satisfies the following properties
    - Atomicity
    - Isolation
    - Serialization only on conflicts
    - (optional) Rollback/abort support

```
UPDATE user set name=“bar” where name=“foo”;
```

Question: Is it possible to provide database transaction semantics to general programming?

How transactions work

- Prepare
  - Make private copy of shared data
- Work
  - Make updates on private copy
- Commit
  - If shared data is unchanged
    - Update shared data with private copy
  - Else conflict has occurred
    - Discard private copy and repeat transaction

Transactions vs. Manual Locks

- Manual locking issues:
  - Under-locking
  - Acquires in different orders
  - Blocking
  - Conservative serialization (pessimistic)
- How transactions help:
  - No explicit locks
  - No ordering
  - Non-blocking, can cancel transactions
  - Serialization only on conflicts (optimistic)

Transactions: simpler and more efficient

Transactional Memory

- Attempts to simplify parallel programming by allowing a group of load and store instructions to execute in an atomic way.
- A concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing.
- Simplifies concurrent programming significantly
  - Hardware TM
  - Software TM

STM

- Transactions run in software
- A thread executes a transaction
- The transaction either commits or aborts
- Non-blocking - The system makes progress
- Features
    - More flexible
    - Easier to modify and evolve
    - Integrate better with existing systems and languages

Language Support

```
// Insert into a doubly-linked list atomically
atomic { // acquire lock
   newNode->prev = node;
   newNode->next = node->next;
   node->next->prev = newNode;
   node->next = newNode;
} // release lock

// Guard condition
atomic (queueSize > 0) { // conditional variable
   // remove item from queue and use it
}
```

Direct Update STM

- Augment objects with (i) a lock, (ii) a version number
- Transactional write:
  - Lock objects before they are written to (abort if another thread has that lock)
  - Log the overwritten data – we need it to restore the heap in case of retry, transaction abort, or a conflict with a concurrent thread
  - Make the update in place to the object
  - Increase the version numbers of object we've written, unlocking them

Direct Update STM

- Transactional read:
    - Log the object's version number
    - Read from the object itself
    - Check the version numbers of objects we've read
        - The same: commit
        - Different: abort

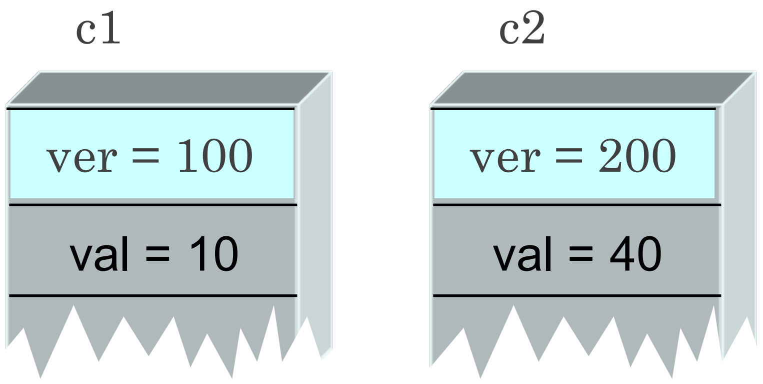
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

T2's log:

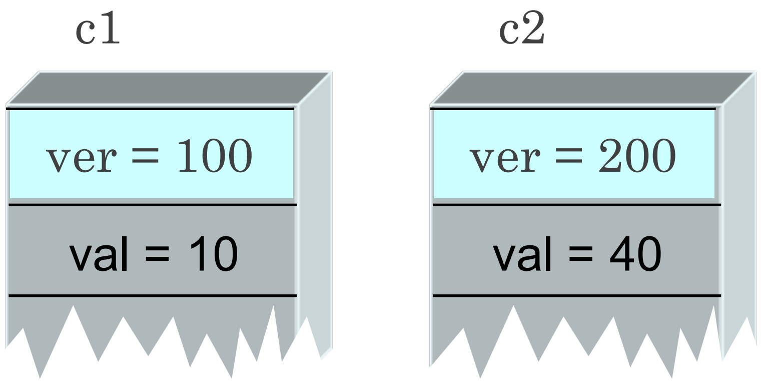
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100

T2's log:

T1 reads from c1: logs that it saw version 100

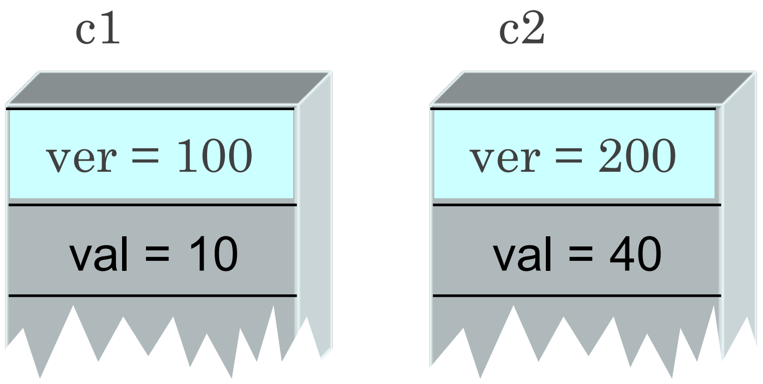
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100

T2's log:

c1.ver=100

T2 also reads from c1: logs that it saw version 100

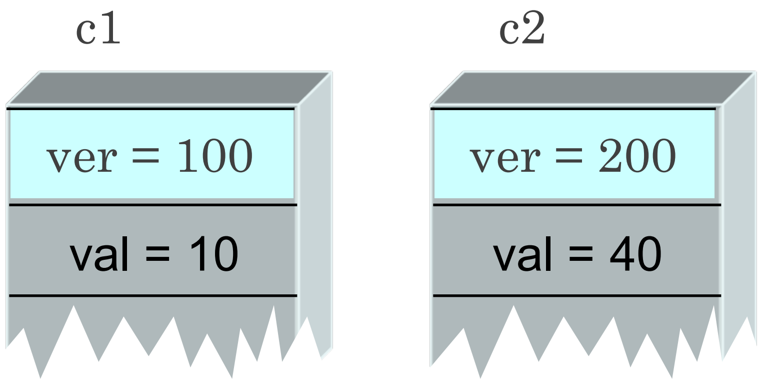
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100

Suppose T1 now reads from c2, sees it at version 200

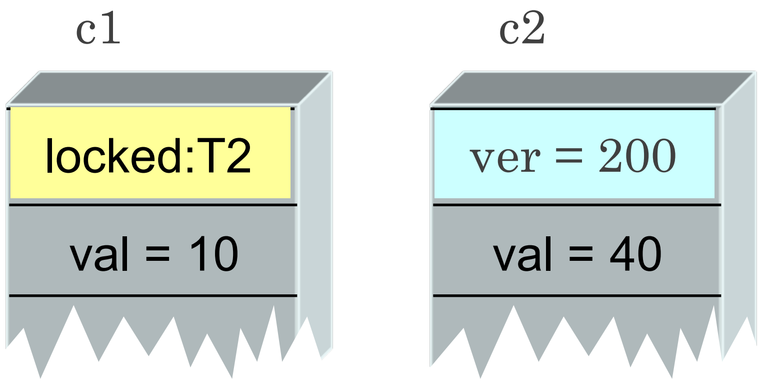
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100
lock: c1, 100

Before updating c1, thread T2 must lock it: record old version number

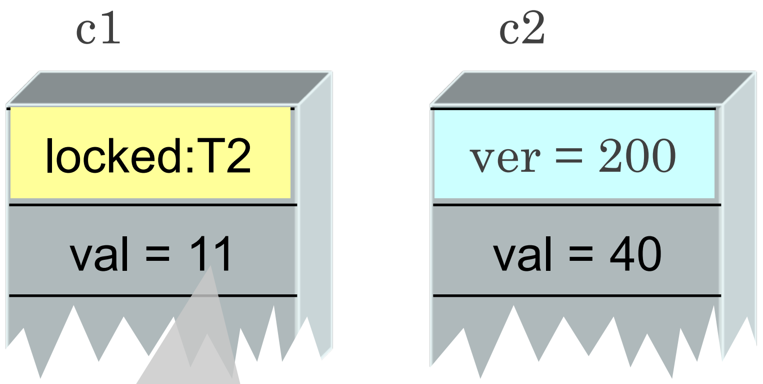
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



(2) After logging the old value, T2 makes its update in place to c1

T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100
lock: c1, 100
c1.val=10

(1) Before updating c1.val, thread T2 must log the data it's going to overwrite

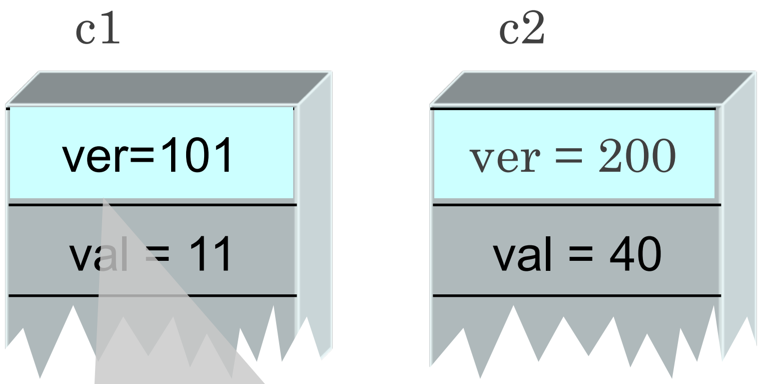
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

```
c1.ver=100
c2.ver=200
```

T2's log:

```
c1.ver=100
lock: c1, 100
c1.val=10
```

(2) T2's transaction commits successfully. Unlock the object, installing the new version number

(1) Check the version we locked matches the version we previously read

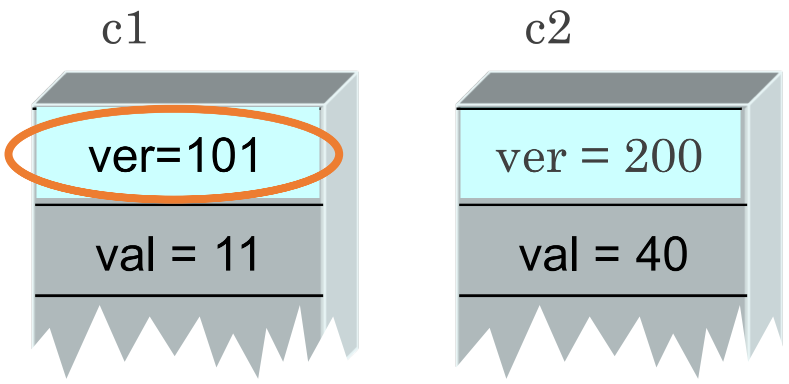
Example: contention between transactions

Thread T1

```
int t = 0;
atomic {
  t += c1.val;
  t += c2.val;
}
```

Thread T2

```
atomic {
  t = c1.val;
  t ++;
  c1.val = t;
}
```



T1's log:

c1.ver=100
c2.ver=200

T2's log:

c1.ver=100
lock: c1, 100
c1.val=10

(1) T1 attempts to commit. Check the versions it read are still up-to-date.

(2) Object c1 was updated from version 100 to 101, so T1's transaction is aborted and re-run.

State of the art

- Hardware:
  - Rock processor (canceled by Oracle)
  - Blue Gene/Q processor from IBM (Sequoia supercomputer)[9]
  - IBM zEnterprise EC12, the first commercial server to include transactional memory processor instructions
  - Intel's Transactional Synchronization Extensions (TSX), available in select Haswell-based processors and newer
  - IBM POWER8[10][11]

- Software:
  - Vega 2 from Azul Systems[12]
  - STM Monad in the Glasgow Haskell Compiler
  - STMX in Common Lisp[13]
  - Refs in Clojure
  - gcc 4.7+ for C/C++[14][15][16][17]
  - PyPy[18]

Summary

- Why is multi-core architecture chosen?
- How does transaction simplify concurrency control?
- What is TM and how does STM work?

Discussion

- Why TM is not popular in single-core era?

- STM challenges
  - Communications, or side-effects
    - File I/O, network……
  - Interrupts
  - Long transactions
  - Nested transactions

THANKS

Backup slides

Compiler integration

- We expose decomposed log-writing operations in the compiler's internal intermediate code (no change to MSIL)
  - OpenForRead – before the first time we read from an object (e.g. c1 or c2 in the examples)
  - OpenForUpdate – before the first time we update an object
  - LogOldValue – before the first time we write to a given field

Source code

```
atomic {
  …
  t += n.value;
  n = n.next;
  …
}
```

Basic intermediate code

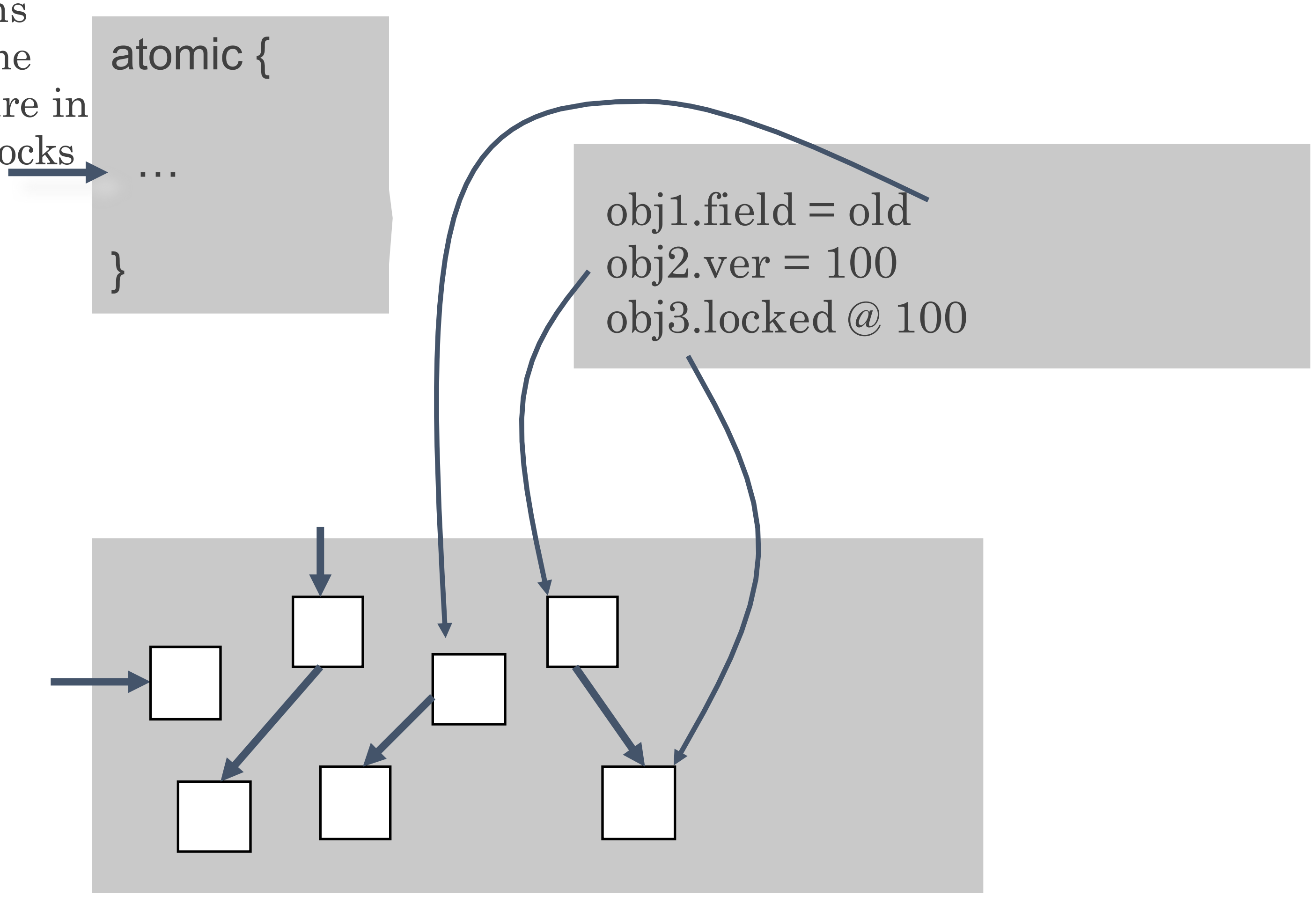
```
OpenForRead(n);
t = n.value;
OpenForRead(n);
n = n.next;
```

Optimized intermediate code

```
OpenForRead(n);
t = n.value;
n = n.next;
```

Runtime integration – garbage collection

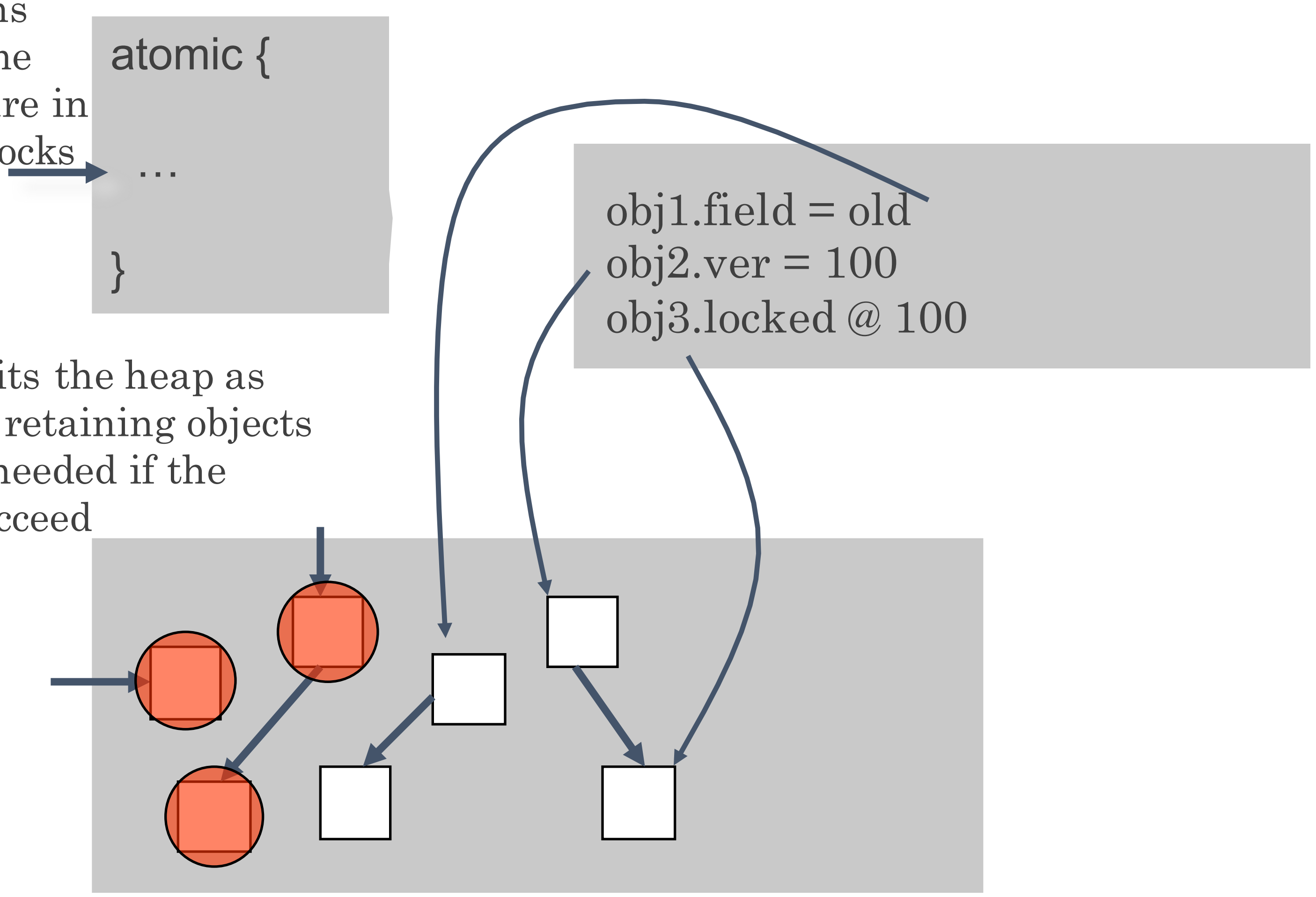
1. GC runs while some threads are in atomic blocks



Runtime integration – garbage collection

1. GC runs while some threads are in atomic blocks

2. GC visits the heap as normal – retaining objects that are needed if the blocks succeed

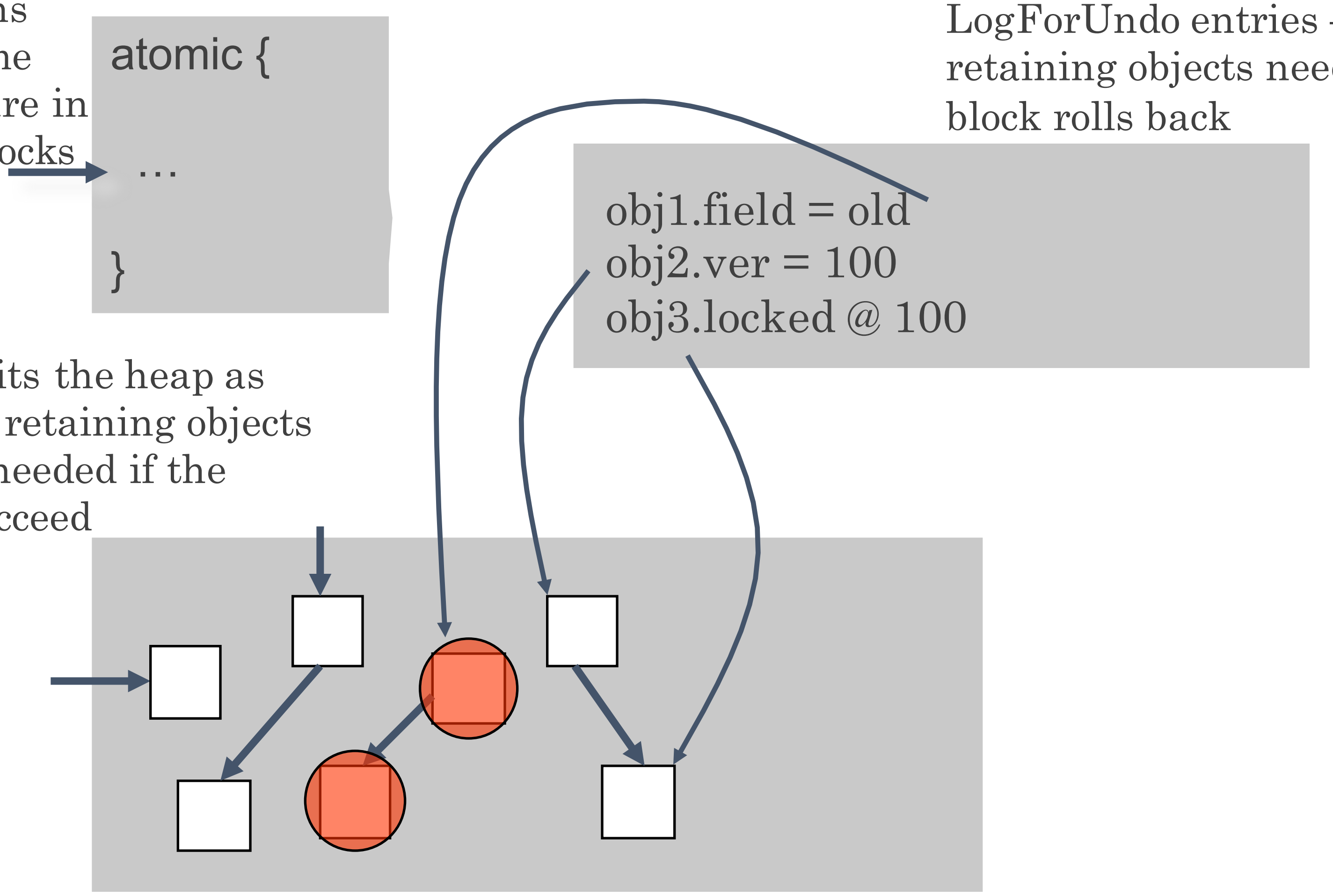


Runtime integration – garbage collection

1. GC runs while some threads are in atomic blocks

2. GC visits the heap as normal – retaining objects that are needed if the blocks succeed

3. GC visits objects reachable from refs overwritten in LogForUndo entries – retaining objects needed if any block rolls back



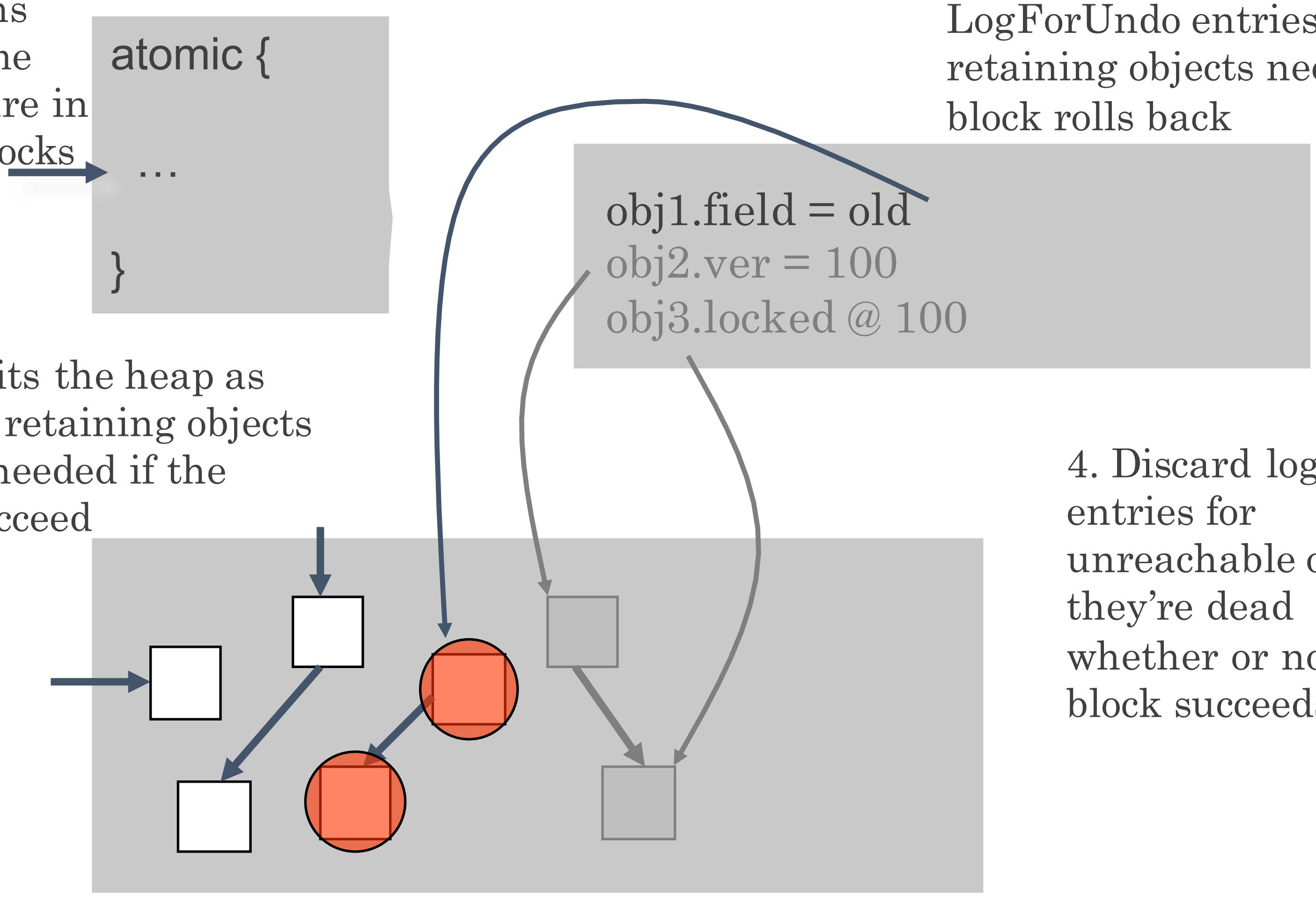
Runtime integration – garbage collection

1. GC runs while some threads are in atomic blocks

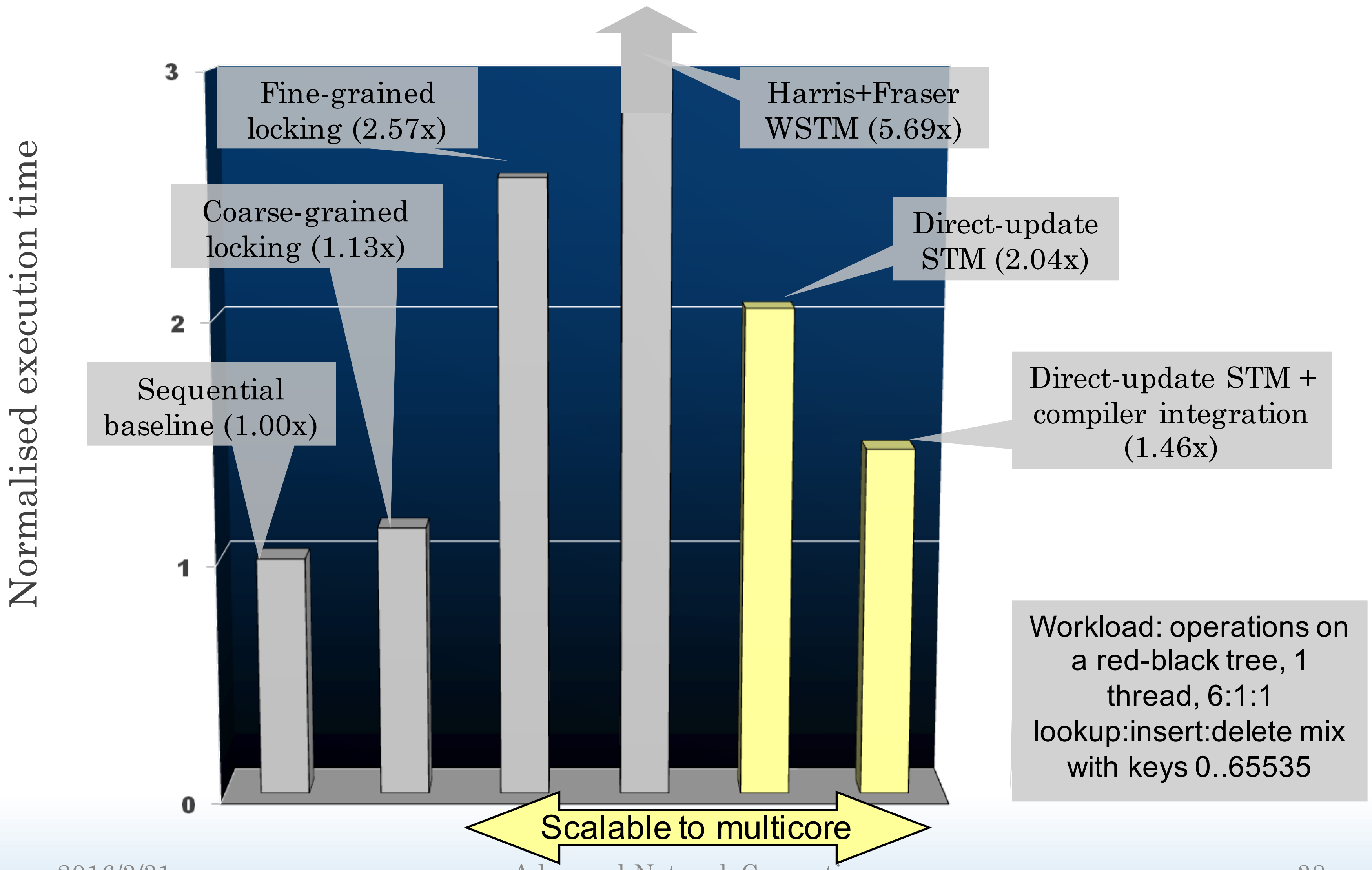
2. GC visits the heap as normal – retaining objects that are needed if the blocks succeed

3. GC visits objects reachable from refs overwritten in LogForUndo entries – retaining objects needed if any block rolls back

4. Discard log entries for unreachable objects: they're dead whether or not the block succeeds

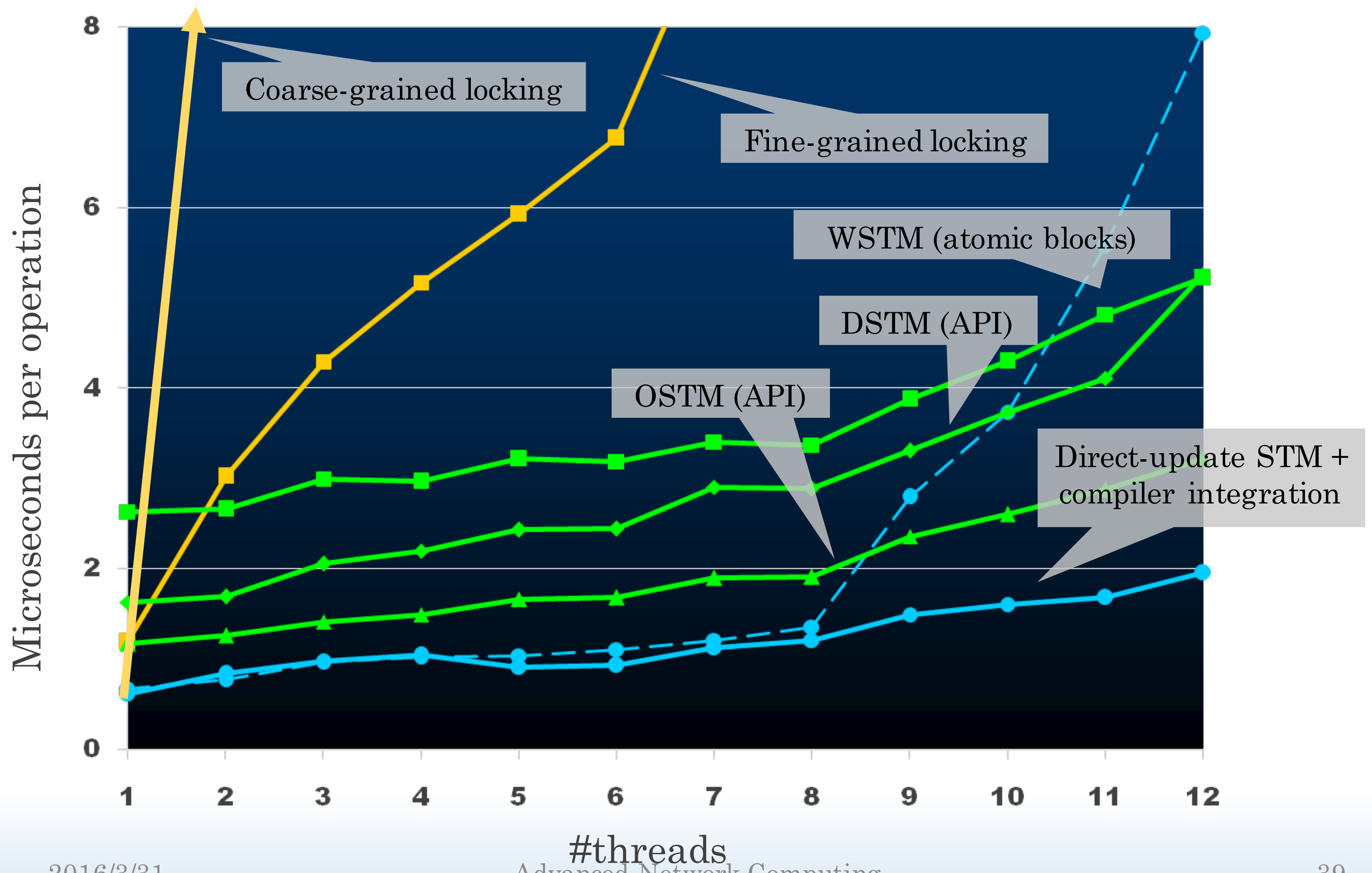


Results: Against Previous Work

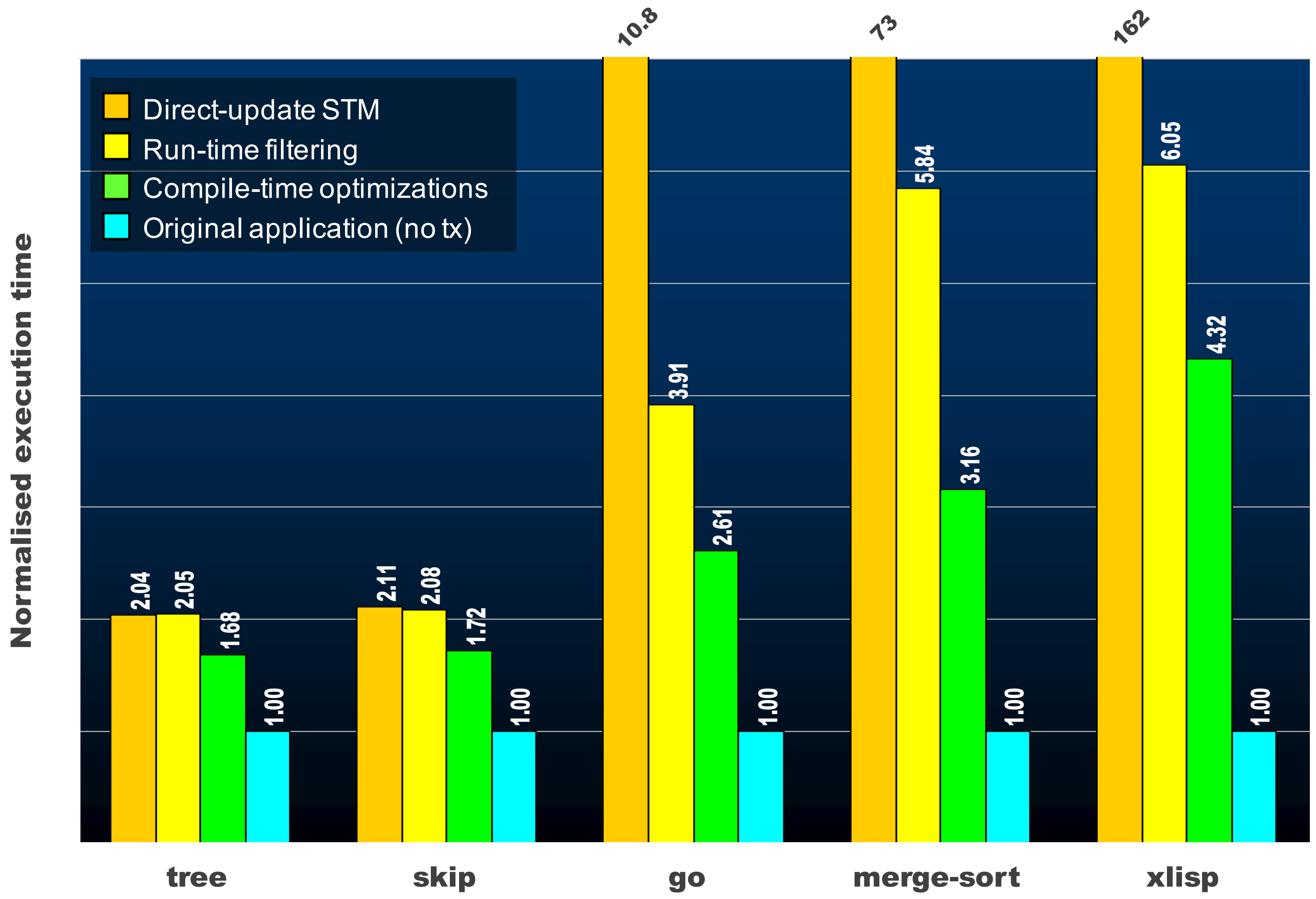


Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

Scalability (μ-benchmark)



Results: long running tests



Summary

- A pure software implementation can perform well and scale to vast transactions
  - Direct update
  - Pessimistic & optimistic CC
  - Compiler support & optimizations
- Still need a better understanding of realistic workload distributions

Design Space

- Hardware Transactional Memory vs. software TM
- Granularity: object, word, block
- Update method
  - Deferred: discard private copy on aborts
  - Direct: control access to data, erase update on aborts
- Concurrency control
  - Pessimistic: prevent conflicts by locking
  - Optimistic: assumes no conflict and retry if there is
- …

Potential Multi-Core Apps

Application Category	Examples
Server apps w/o shared state	Apache web server
Server apps with shared state	MMORPG game server
Stream-Sort data processing	MapReduce, Yahoo Pig
Scientific computing (many different models)	BLAS, Monte Carlo, N-Body …
Machine learning	HMM, EM algorithm
Graphics and games	NVIDIA Cg, GPU computing
……	……