14.–17. 09. 2009
in Nürnberg

Herbstcampus

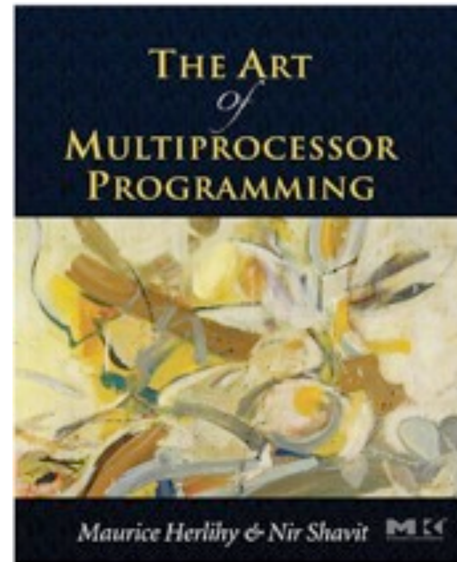
Wissenstransfer
par excellence

Stumbling Blocks and Stepping Stones auf dem Weg zum

Software Transactional Memory

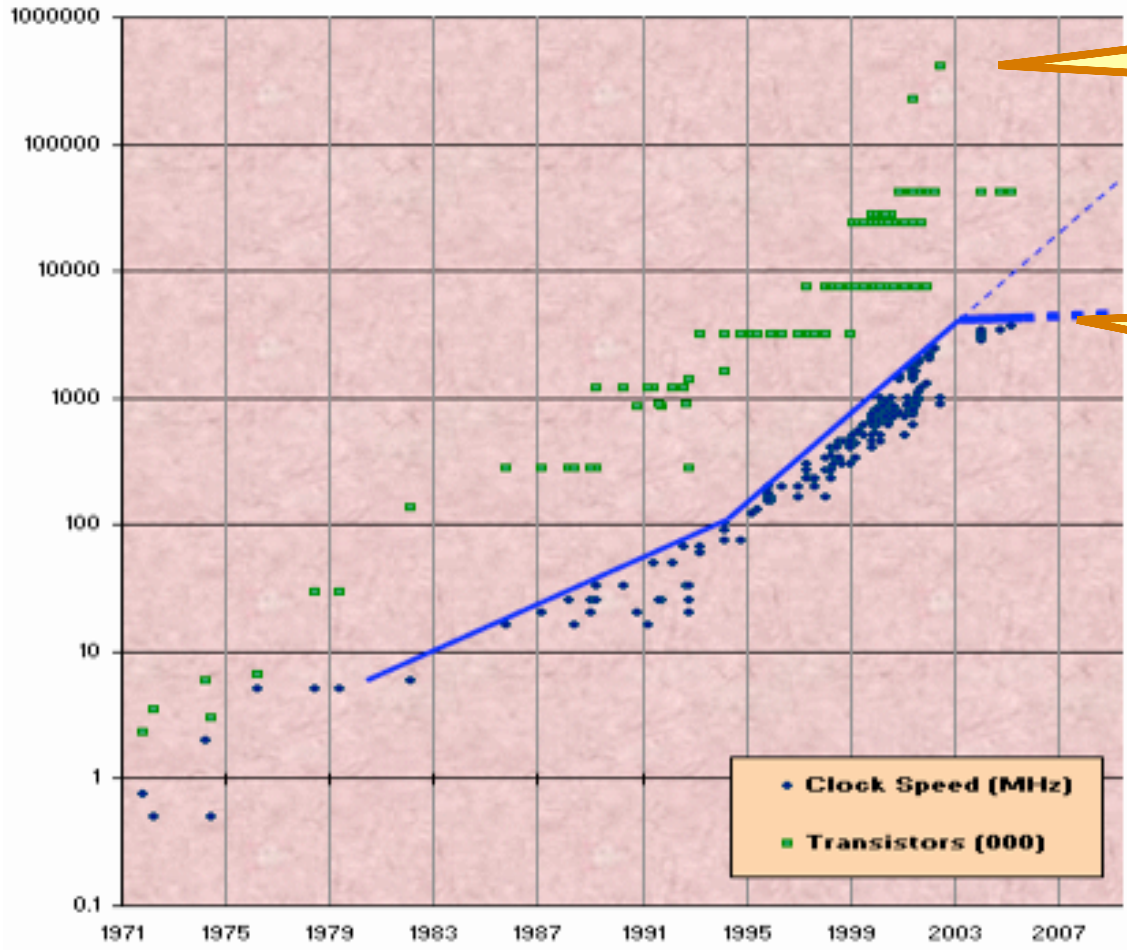
Christoph von Praun
Geoorg-Simon-Ohm Hochschule, Nürnberg

[Herlihy&Shavit'08]



Maurice Herlihy, Nir Shavit:
"The Art of Multiprocessor Programming",
Morgan Kaufmann, 2008.

Moore's Law

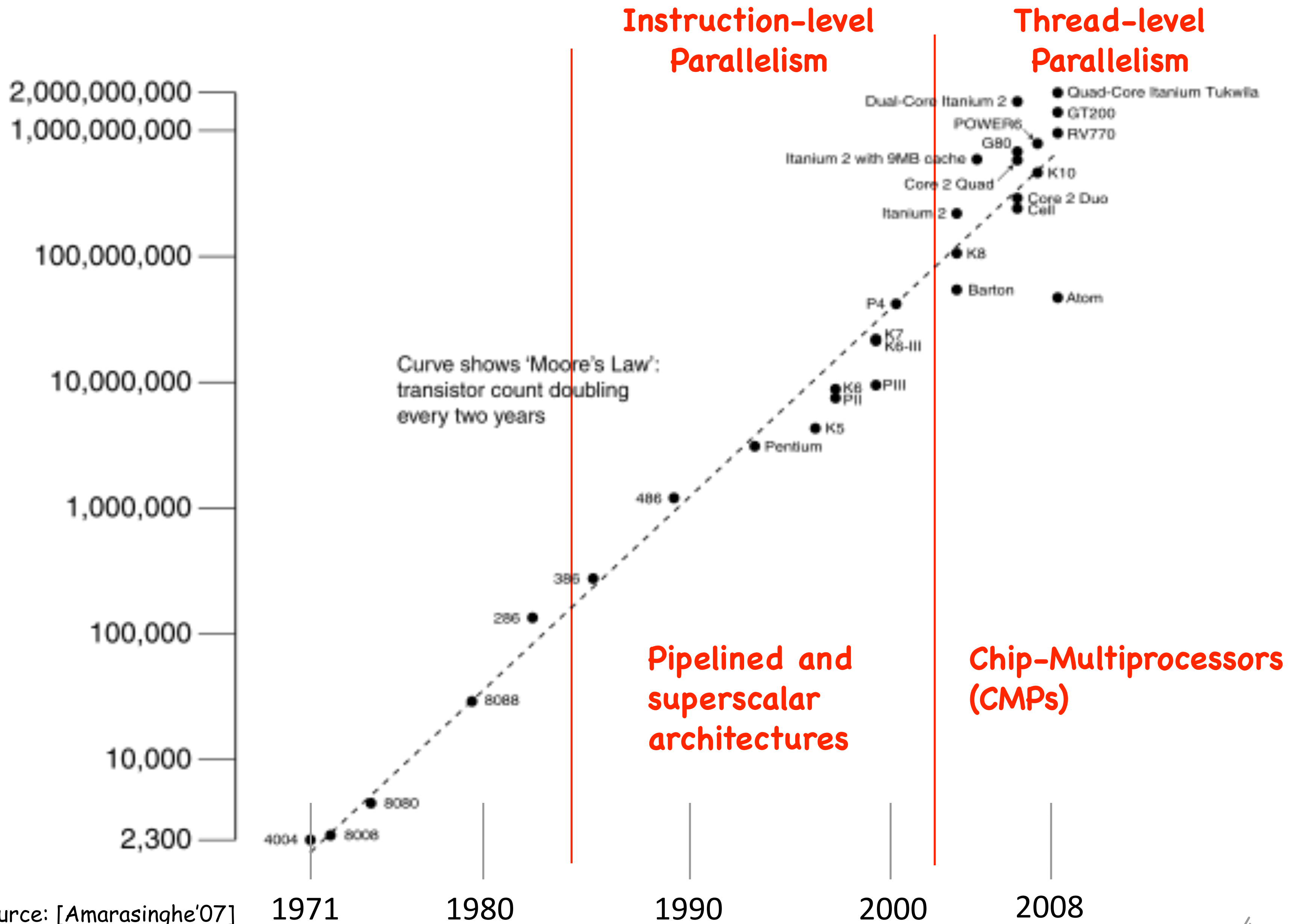


Transistor count still rising

Clock speed flattening sharply

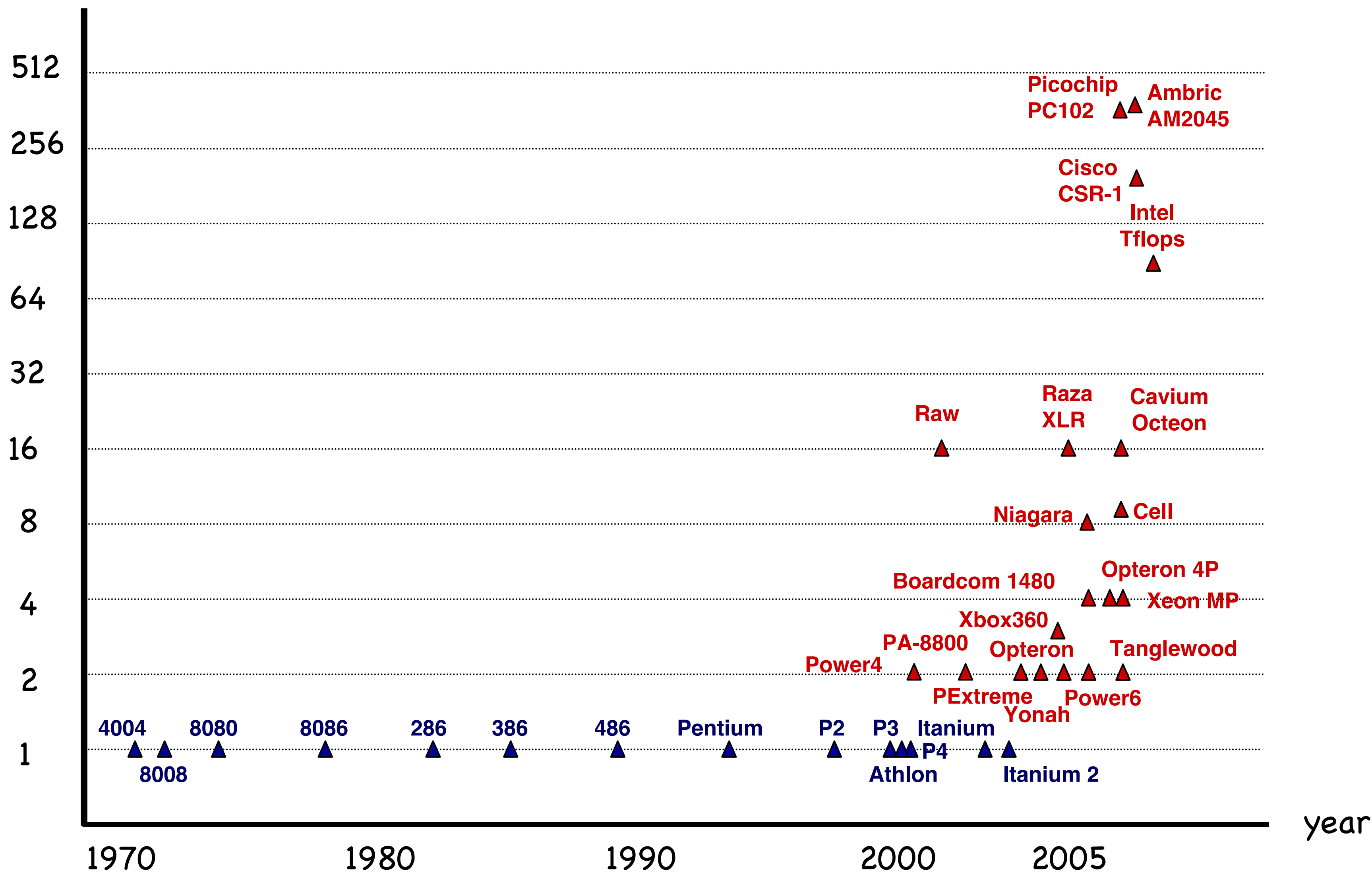
Source: [Peyton-Jones]

VLSI Generations



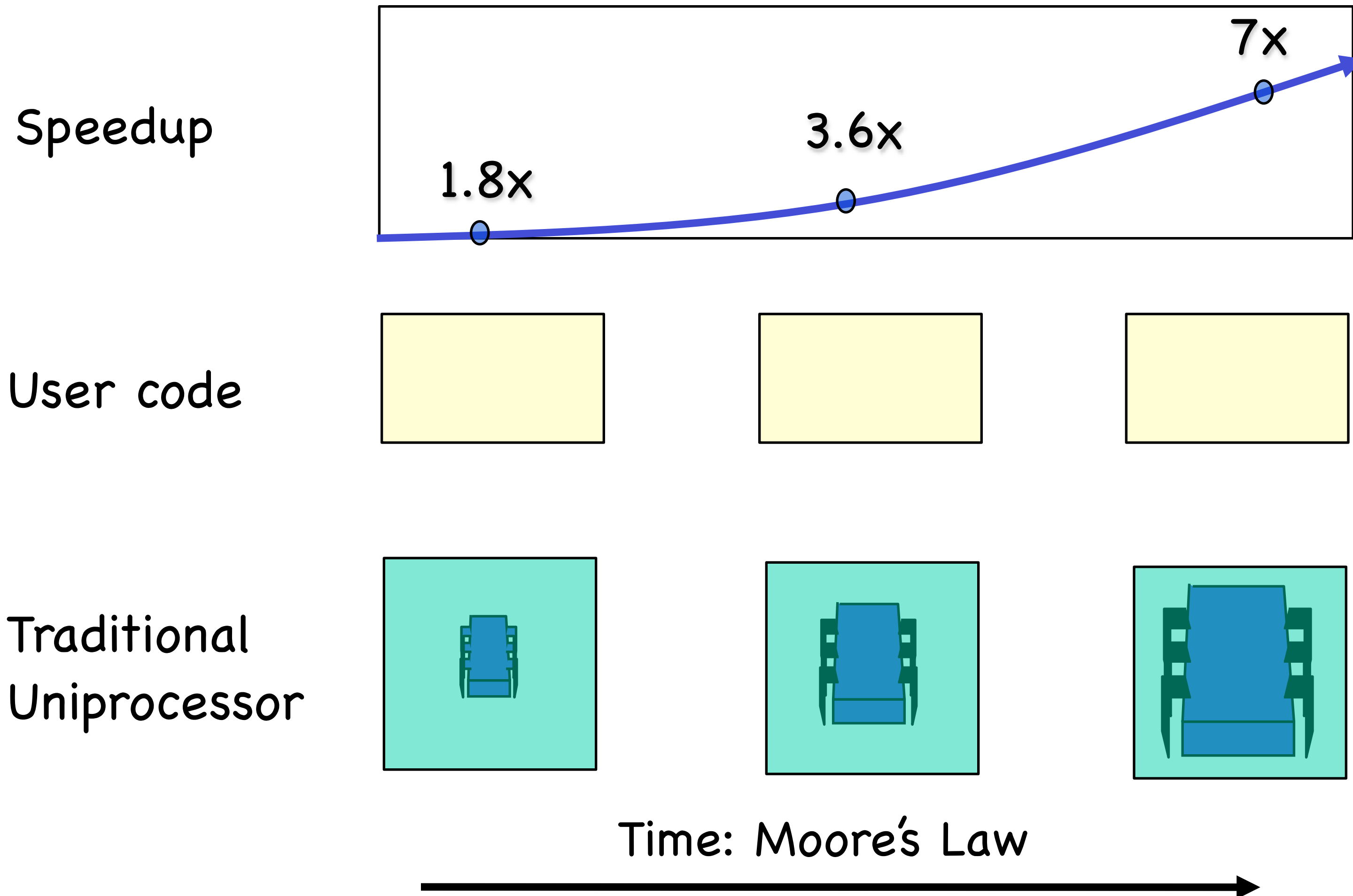
Source: [Amarasinghe'07]

Multicore Architectures



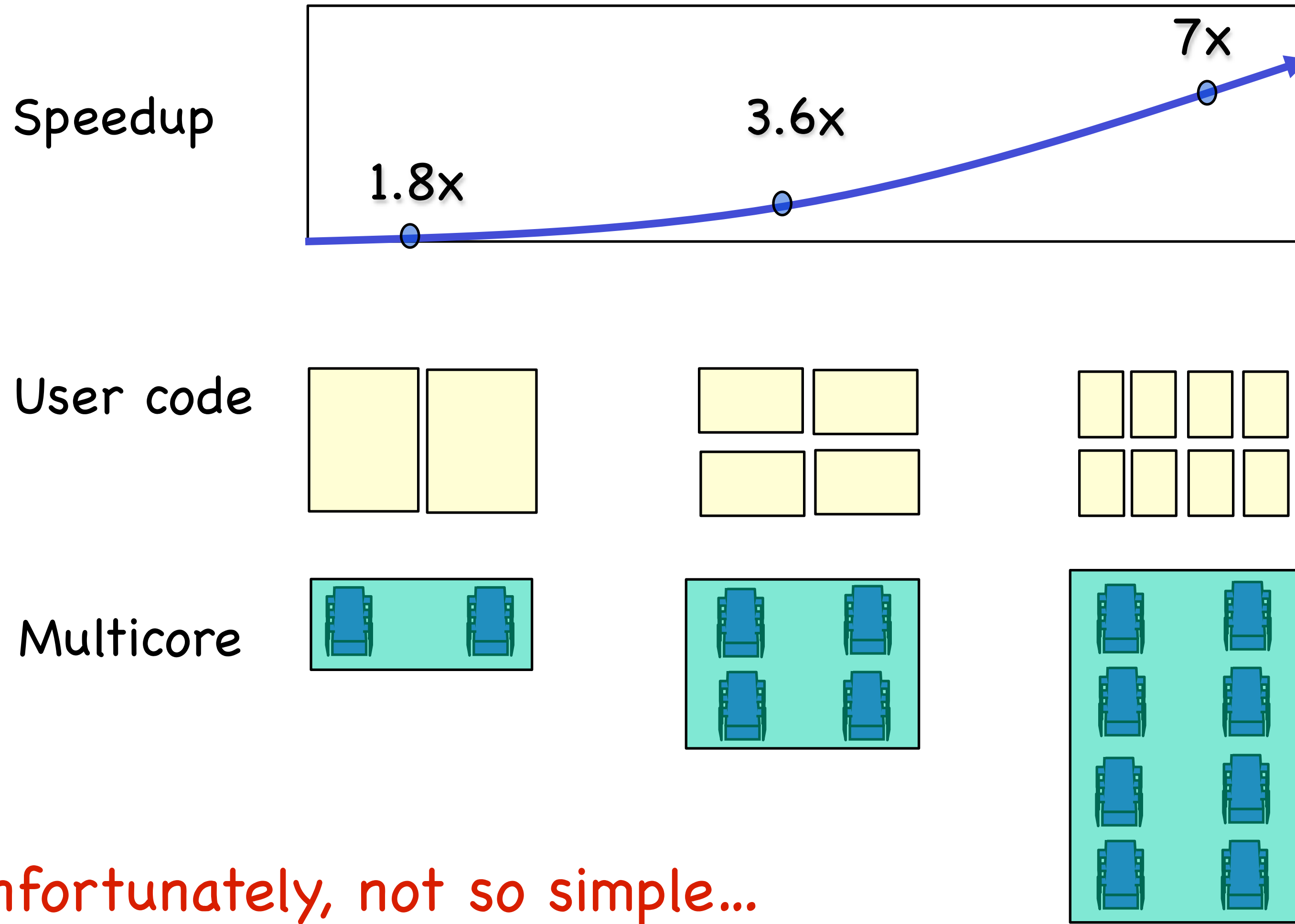
Source: [Amarasinghe'07]

Traditional Scaling



Source: [Herlihy&Shavit'08]

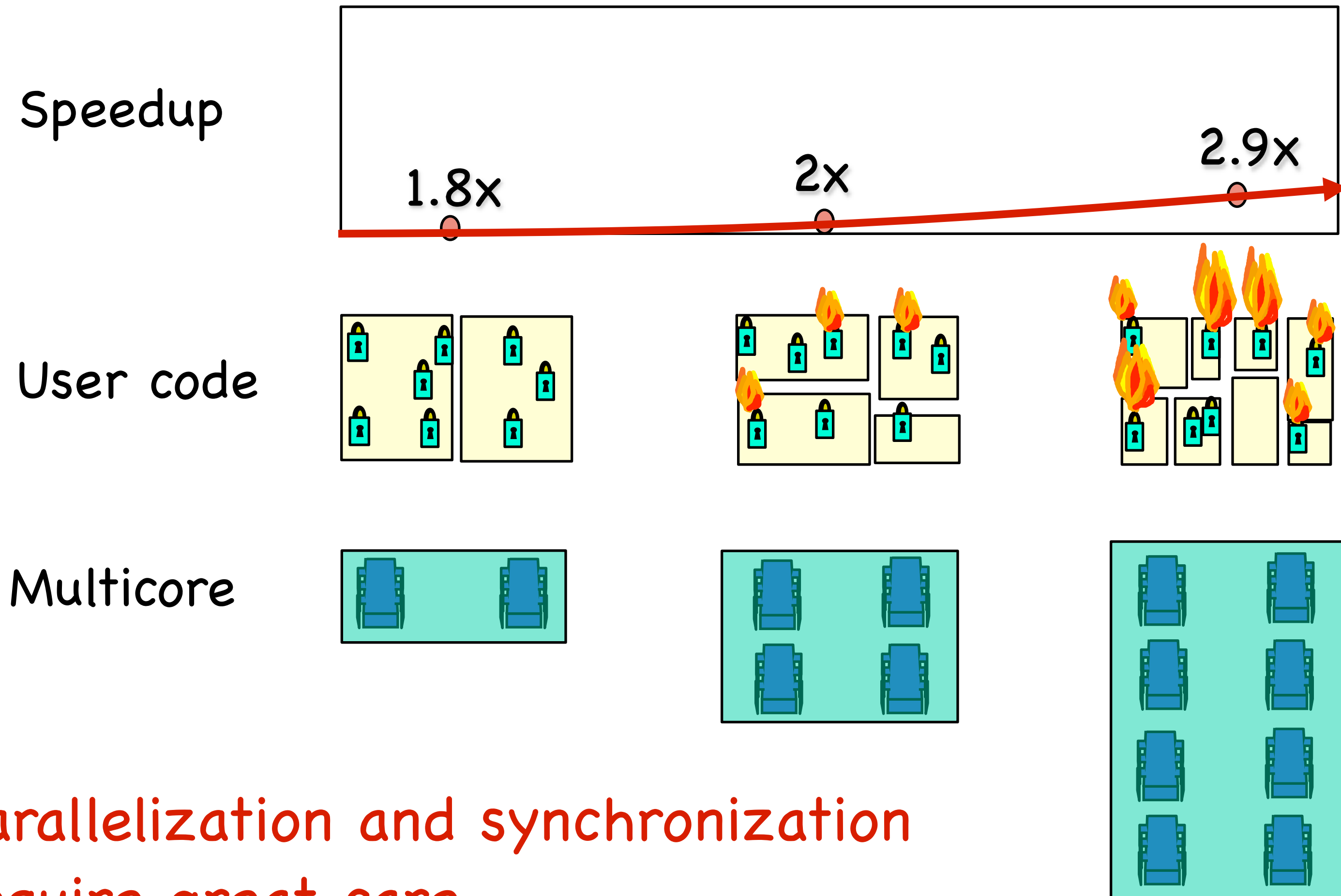
Multicore Scaling Process



Unfortunately, not so simple…

Source: [Herlihy&Shavit'08]

Real-World Scaling Process



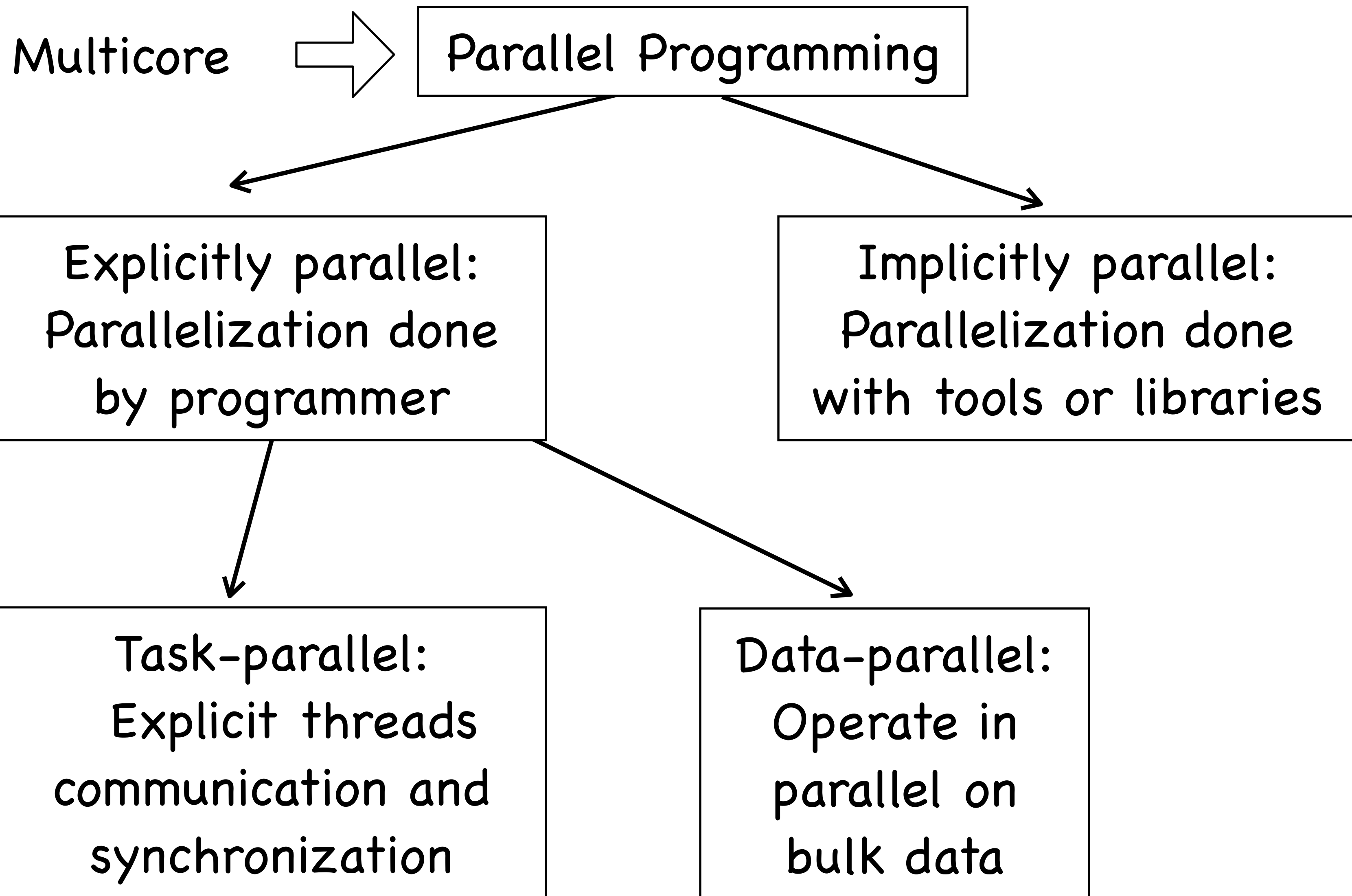
Parallelization and synchronization require great care…

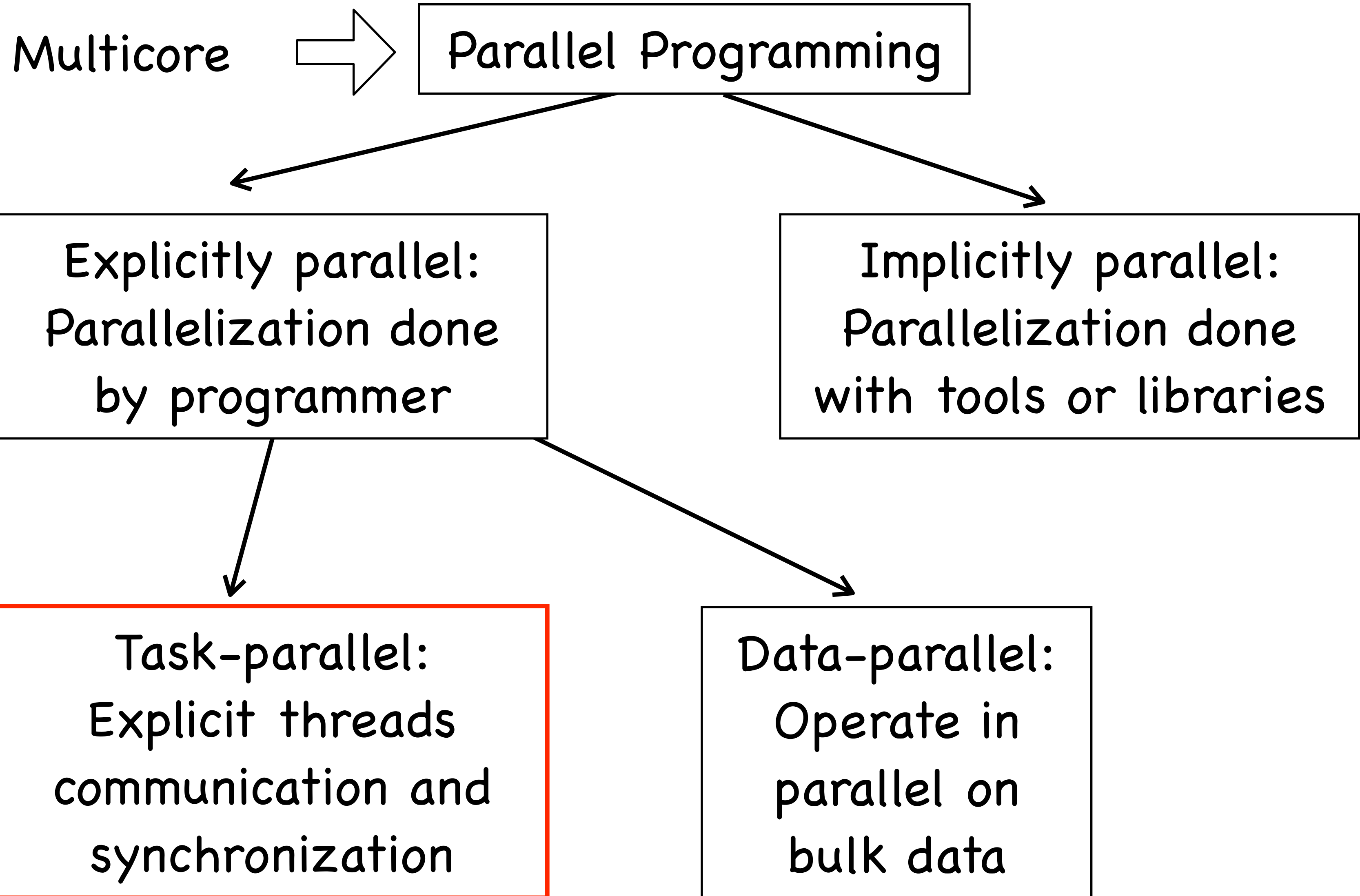
Source: [Herlihy&Shavit'08]

Why do we care?

- Time no longer cures software bloat
- When you double your path length
  - You can't just wait 6 months
  - Your software must somehow exploit twice as much concurrency

Source: [Herlihy&Shavit'08]


Multicore
Parallel Programming
Explicitly parallel: Parallelization done by programmer
Implicitly parallel: Parallelization done with tools or libraries
Task-parallel: Explicit threads communication and synchronization
Data-parallel: Operate in parallel on bulk data


Multicore
Parallel Programming
Explicitly parallel:
Parallelization done
by programmer
Implicitly parallel:
Parallelization done
with tools or libraries
Task-parallel:
Explicit threads
communication and
synchronization
Data-parallel:
Operate in
parallel on
bulk data
This talk

Outline

- Problems with Locking
- TM Intro
- Language Integration
- Empirical Studies about TM
- Design Space for TM Semantics
- Design Space for TM Implementations
- STM Performance

State of the Art in Task Parallel Programming

- Today's software
  - Non-scalable methodologies (Locks)
  - State of the art has not much changed in 30 years
- Today's hardware
  - Poor support for scalable synchronization
- Cannot exploit cheap (hardware) threads

Source: [Herlihy&Shavit'08]

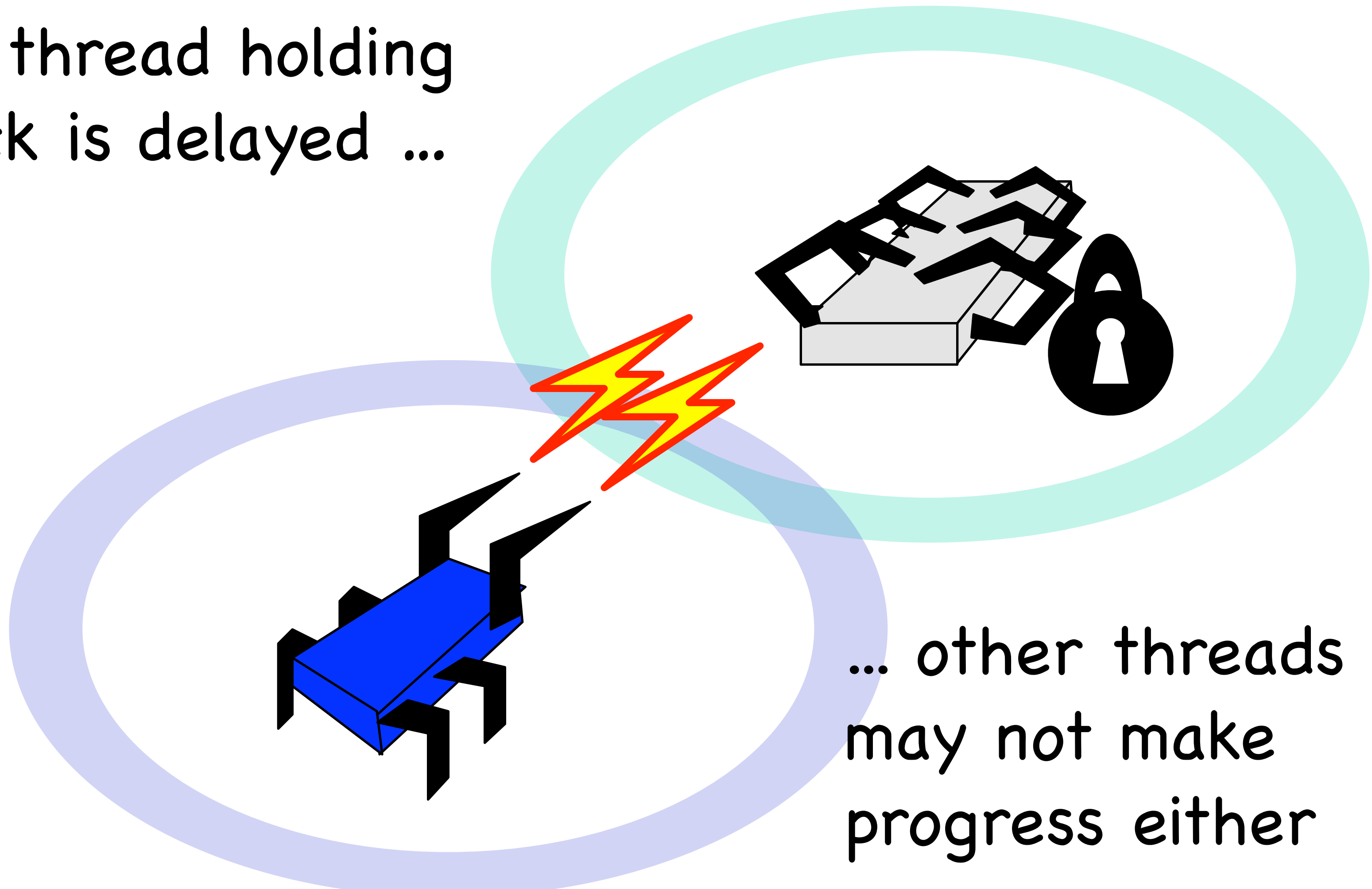
Why Locking Doesn't Scale

- Not Robust
- Relies on Conventions
- Hard to Use
  - Conservative
  - Deadlocks
  - Lost wake-ups
- Not Composable

Source: [Herlihy&Shavit'08]

Locks are not Robust

If a thread holding
a lock is delayed …



… other threads
may not make
progress either

Source: [Herlihy&Shavit'08]

Why Locking Doesn't Scale

- Not Robust
- Relies on conventions
- Hard to Use
  - Conservative
  - Deadlocks
  - Lost wake-ups
- Not Composable

Source: [Herlihy&Shavit'08]

Locking Relies on Conventions

- Relation between
  - Lock bit and object bits
  - Exists only in programmer's mind
- Order in which locks are taken

Actual comment from Linux Kernel

```
/*
 * When a locked buffer is visible to the I/O layer
 * BH_Launder is set. This means before unlocking
 * we must clear BH_Launder,mb() on alpha and then
 * clear BH_Lock, so no reader can see BH_Launder set
 * on an unlocked buffer and then risk to deadlock.
 */
```

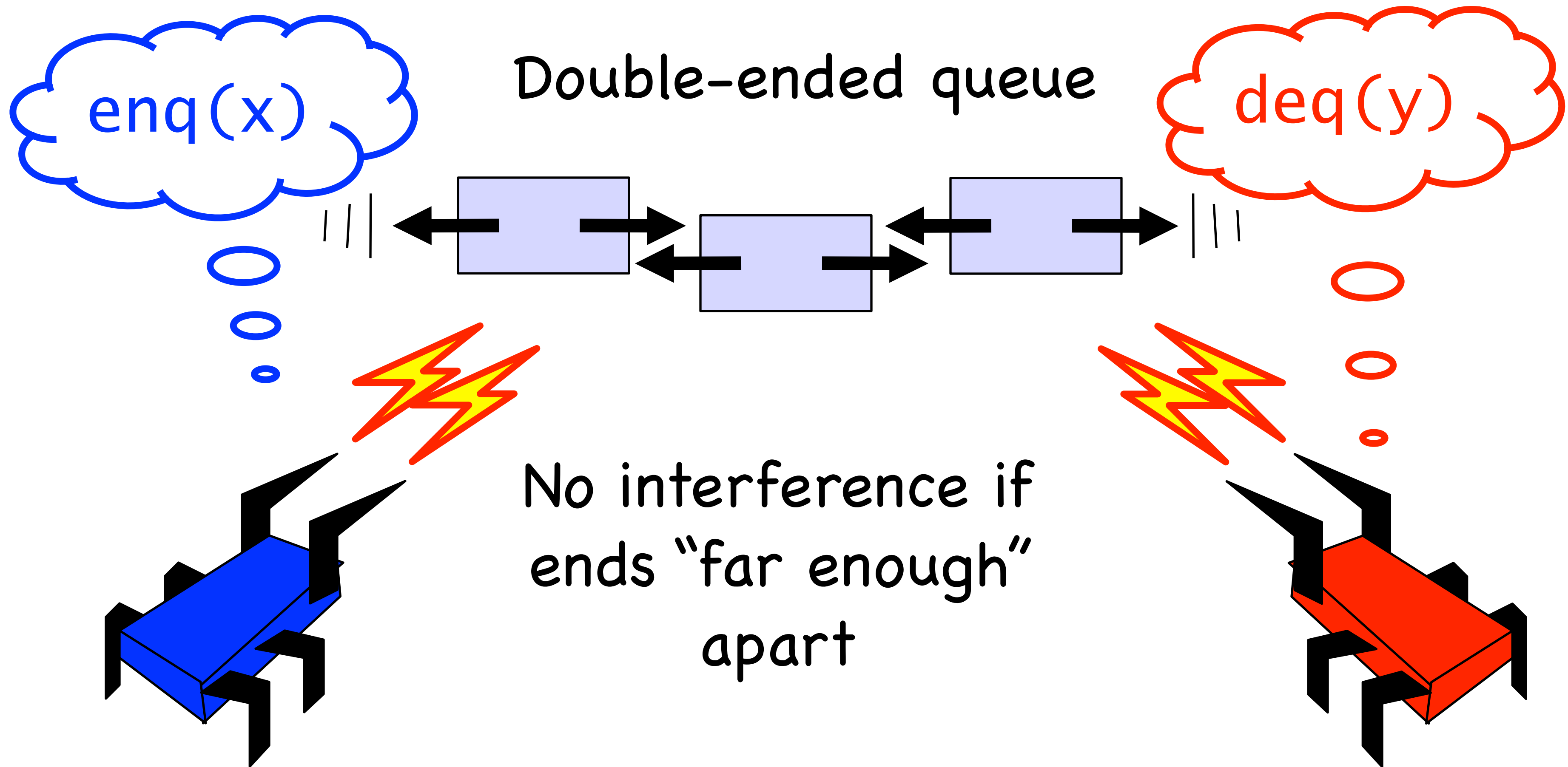
Source: [Bradley Kuszmaul]

Why Locking Doesn't Scale

- Not Robust
- Relies on conventions
- Hard to Use
  - Conservative
  - Deadlocks
  - Lost wake-ups
- Not Composable

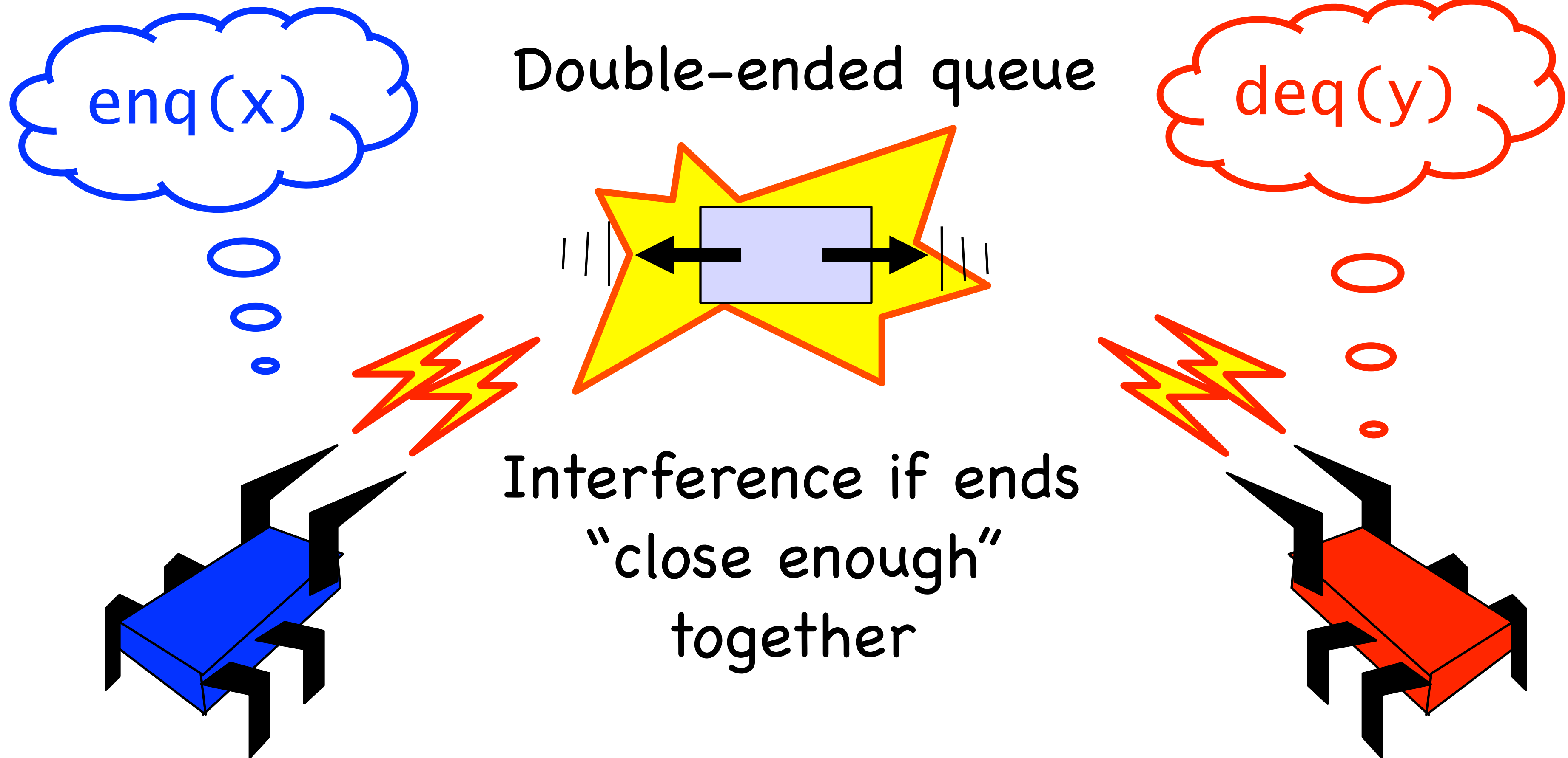
Source: [Herlihy&Shavit'08]

Programming Challenge



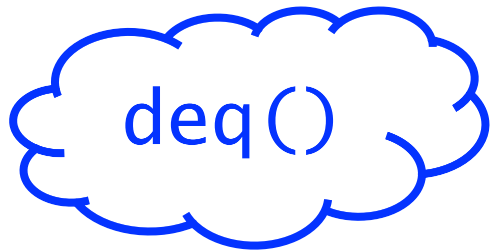
Source: [Herlihy&Shavit'08]

Programming Challenge

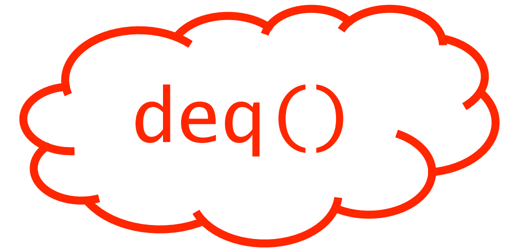


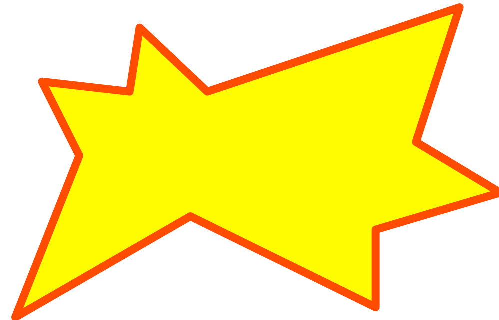
Source: [Herlihy&Shavit'08]

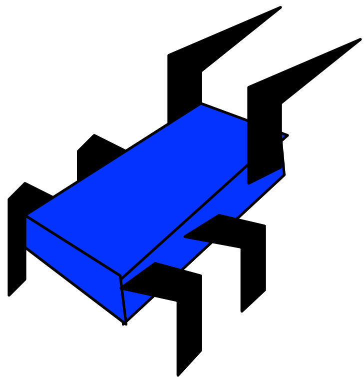
Programming Challenge

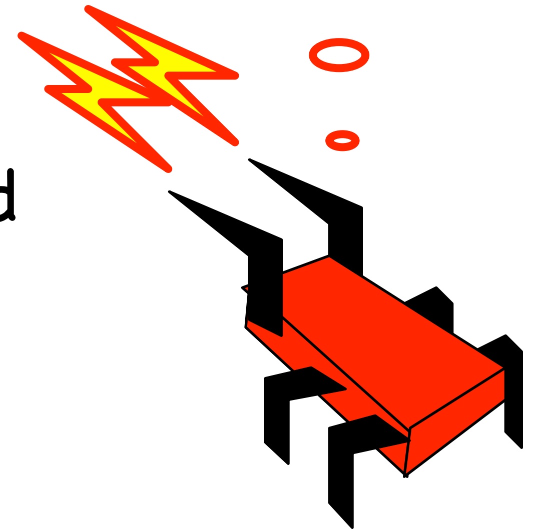


Double-ended queue









Make sure suspended dequeuers awake as needed

Source: [Herlihy&Shavit'08]

You Try It …

- One lock?
  - Too conservative
- Locks at each end?
  - Deadlock
- Without locks, solely using atomic operations?

Source: [Herlihy&Shavit'08]

Actual Solution

- [Michael&Scott'96]
- What good is a methodology (locks, fine-grain atomic operations) where solutions to such elementary problems are hard enough to be publishable?

Source: [Herlihy&Shavit'08]

Why Locking Doesn't Scale

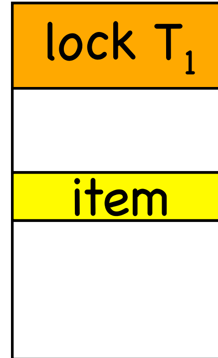
- Not Robust
- Relies on conventions
- Hard to Use
  - Conservative
  - Deadlocks
  - Lost wake-ups
- Not Composable

Source: [Herlihy&Shavit'08]

Locks do not compose

Hashtable

add(T_1, item)

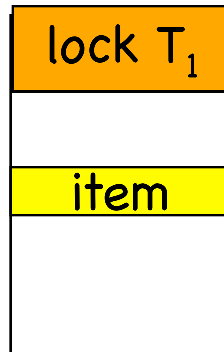


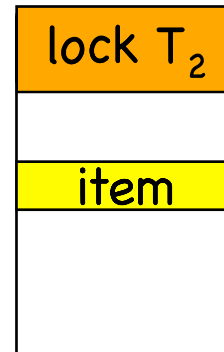
Must lock T_1 before adding item

Move from T_1 to T_2

delete(T_1, item)

add(T_2, item)





Must lock T_2 before deleting from T_1

Exposing lock internals breaks abstraction.

Source: [Herlihy&Shavit'08]

Outline

- Problems with Locking
- TM Intro
- Language Integration
- Empirical Studies about TM
- Design Space for TM Semantics
- Design Space for TM Implementations
- STM Performance

Double ended queue revisited

```
public void enq(Item x) {

    QNode q = new QNode(x);
    q.left = this.left;
    this.left.right = q;
    this.left = q;

}
```

sequential program

Double ended queue revisited

```
public void enq(Item x) {
  atomic {
    QNode q = new QNode(x);
    q.left = this.left;
    this.left.right = q;
    this.left = q;
  }
}
```

transaction

sequential program

ACID principle from database systems:

- Atomicity: All-or-nothing semantics
- Isolation: Effects of concurrent computations do not leak into transaction.

Possible implementation of `atomic`

Optimistic concurrency:

```
atomic { <sequential code> }
```

- … read and write operations in <sequential code> are recorded in thread-local log
  - Writes go to log, not to memory
  - Reads obtain value from log or memory
- Commit at the end:
  - in one atomic step, check validity of prior reads and update memory
  - If commit fails, rerun transaction

atomic is Compositional

Transfer item from one queue to another:

```
public void transfer(Queue q1, Queue q2) {
  atomic {
    Item tmp = q1.deq();
    q2.enq(tmp);
  }
}
```

Conditional Blocking

```
public Item deq() {
  atomic {
    if (this.left == null)
      retry;
    ...
  }
}
```


- Re-execute when the value of a previously read variable changes
- No condition variables, no lost wakeups!

Blocking is Compositional

```
public void transfer(Queue q1, Queue q2) {
  atomic {
    Item tmp = q1.deq();
    q2.enq(tmp);
  }
}
```

- Transaction succeeds only if
  - q1 is not empty
  - q2 is not full
- No need to rewrite deq() and enq()
- Note: wait() and signal() do not compose

- Problems with Locking
- TM Intro
- Language Integration
  - Library and Compiler Support
  - Exception Handling
  - I/O
  - Semantics of Nested Transactions
- Empirical Studies about TM
- Design Space for TM Semantics
- Design Space for TM Implementations
- STM Performance

We Don't have Language Support (Yet)

STMs typically implemented as a library

- sometimes with compiler support

```
...
 a = 5;
 atomic {
   b = a+5;
 }
 c = b;
...
```

library calls

```
...
 a = 5;
 stmStart();
   temp = stmRead(&a);
   stmWrite(&b, temp +5);
 stmCommit();
 c = b;
...
```

Source: [Herlihy&Shavit'08]

We Don't have Language Support (Yet)

- Compiler provides
  - syntactic convenience for the programmer
  - correctness
    - programmer may instrument too few accesses
  - optimizations
    - programmer may instrument too many accesses
- Still, design and development of an STM solely based on a library is hard …

Source: [Herlihy&Shavit'08]

Why It's Hard

- TM is not just a collection of useful objects and methods
- Effect of transactional synchronization is pervasive
  - How functions are defined
  - Control flow: commit & abort
  - Exception handling
  - Irrevocable actions, I/O

Source: [Herlihy&Shavit'08]

Exceptions

```
atomic {
  try {
    ...
  } catch (SomeException e) { ... }
}
```

Should uncaught exceptions commit or abort a transaction?

- Commit: May leave the data structure in inconsistent state.
- Abort: What about exception object itself, and transactional state that may be reachable from exception object?

I/O

- atomic blocks require possibility to revoke operations (rollback)
- Not obvious for I/O:

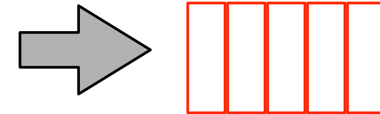
```
atomic {
  if (x == y)
    launchMissiles();
}
```

- Transaction may see x==y due to interleaving with other transactions
- Such transaction is doomed to roll back and must not call launchMissiles()

Transactional Output is OK

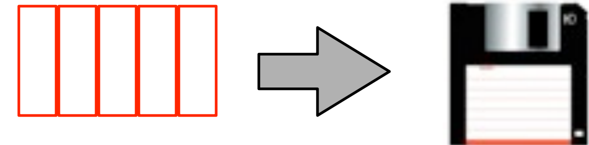
1. Output is buffered in transactional shared memory

```
atomic {
  if (x == y)
    print(txbuffer, “Hello world”);
}
```



2. Separate I/O thread performs “real” output

```
while (true)
  atomic {
    char* tmp = txbuffer.get();
    if (tmp)  print(tmp)
    else retry;
  }
}
```



Outline

- Problems with Locking
- TM Intro
- Language Integration
- Empirical Studies about TM
  - User Study
  - Application Study of Real World Concurrency Bugs
- Design Space for TM Semantics
- Design Space for TM Implementations
- STM Performance

User Study [RossbachEtAl'09]

147 undergraduate students are given simple parallel programming assignment.

Different techniques should be used for concurrency control (one big lock, fine-grained locking, TM)

Results:

- TM is much less error-prone than fine-grain locking
- Newbie programmers have trouble understanding transactions, though TM is still easier than fine-grain locks.

Source: [RossbachEtAl'08]

Study of Real World Concurrency Bugs [LuEtAl'08]

Study of 105 bugs in 4 randomly chosen very large open-source programs:

- "TM can help avoid about one third (39%) of the examined concurrency bugs."
- "Some (19%) of the examined concurrency bugs cannot benefit from basic TM designs because of their bug pattern."

Source: [LuEtAl'08]

Outline

- Problems with Locking
- TM Intro
- Language Integration
- Empirical Studies about TM
- Design Space for TM Semantics
  - Atomicity and Isolation
  - Ordering
- Design Space for TM Implementations
- STM Performance

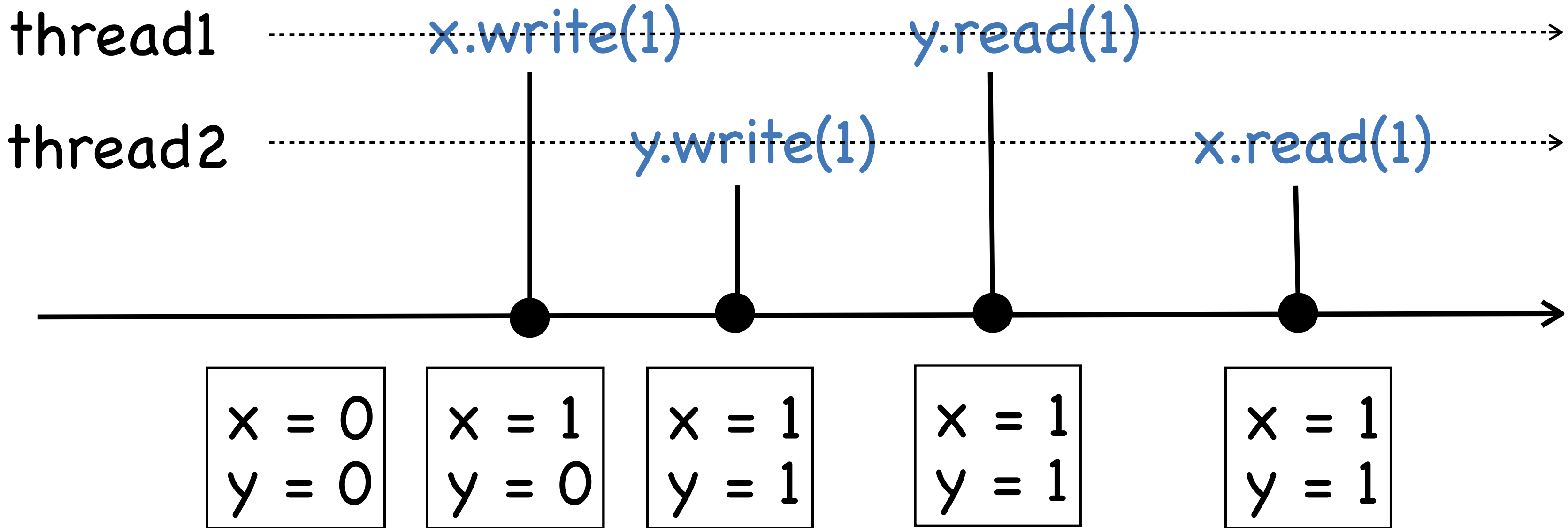
A “Simple” Model of Concurrency

“The behavior of a concurrent program is the interleaving of operations executed by individual threads”.

Premises:

1. Operations are atomic
2. Threads execute operations in program order
3. Operations are observed in a total global order compatible with the program order.

A "Simple" Model of Concurrency



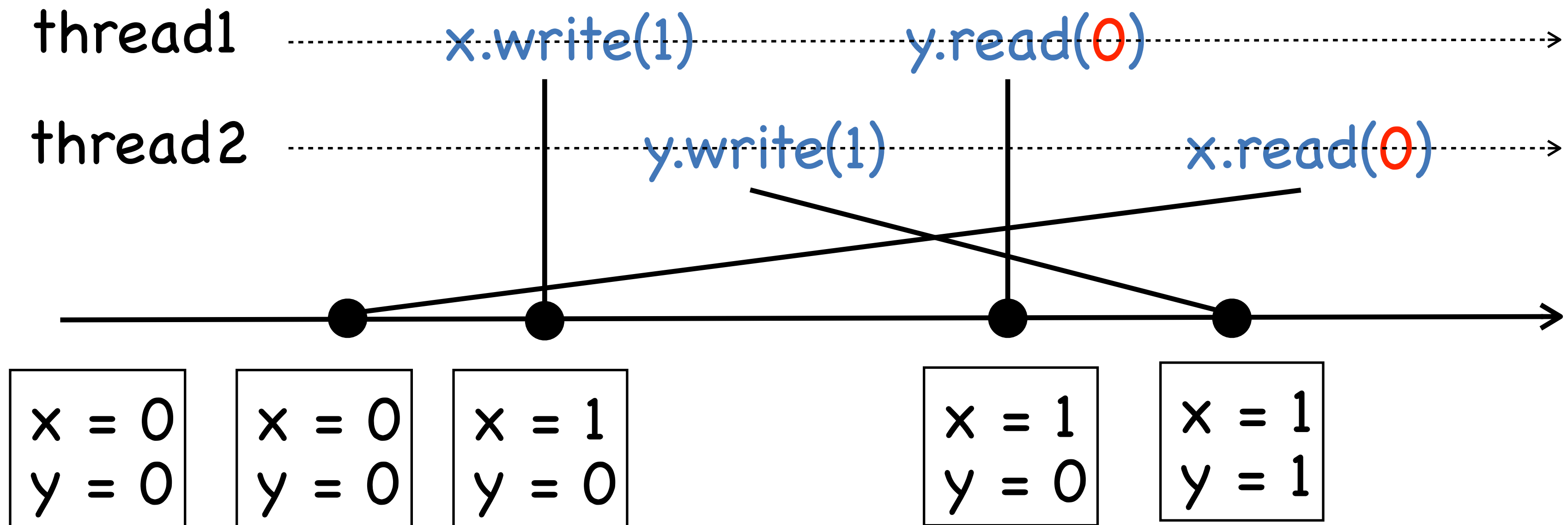
---> program order
⟶ real time execution order

Reality is different

1. Operations are atomic
   Modern processors support atomic read, write and rmw operation at word-granularity
2. Threads execute operations in program order
3. Operations are observed in a total global order compatible with the program order.
   Compiler and processors re-order individual operations if data- and control-dependences in the sequential program permit.

Example

Thread 2 violates principle #2 (program order)



TM to the Rescue!

1. Operations are atomic
   Atomic blocks group composite operations.
2. Threads execute operations in program order
   Atomic blocks order operations within a thread.
3. Operations are observed in a total global order compatible with the program order.
   Atomic blocks induce a global synchronization order.

Semantics of Atomic Blocks

Unfortunately reality is not as bright when looking at the details. Two topics:

Atomicity and Isolation (ACID)

1. Operations are atomic

Ordering

2. Threads execute operations in program order
3. Operations are observed in a total global order compatible with the program order.

ACID Revisited

Atomicity: Partial effects of a transaction are not visible to concurrent computations.

Isolation: Effects of concurrent computations do not leak into a txn.

Any concurrent computation
- Semantics called strong atomicity
- Ideal, but probably inefficient to implement in software

Only other transactions
- Semantics called weak atomicity
- Reasonable model for STM

Initially x, y == 0

Thread 1	Thread 2
```atomic { x = 0; if (x == 1) y = 1; }```	```x = 1;```

```
Thread 1                 | Thread 2
atomic {                 | x = 1;
  x = 0;                 |
  if (x == 1)            |
    y = 1;               |
}                        |
```

Can y == 1?

Strong atomicity says: "No!"

- Sequential reasoning inside atomic block

Weak atomicity says: "Yes!"

- Non-local reasoning necessary

Languages need High-level Memory Models

- Strong vs. weak atomicity is decided by programming language designers
  - high-level memory model
  - Details: [GrossmanEtAl'06]
- Caveat: Weak atomicity has many flavors!

Strong Atomicity

… gives the following guarantees:

1) Inside a transaction, multiple accesses to the same variable return the same value provided that no write intervenes.
2) If a variable is written inside an transaction, subsequent reads in the transaction obtain the value that was written.
3) An intermediate value, which is overwritten in the same transaction or a retry is not visible to other computations.

Flavors of Weak Atomicity

- Weak atomicity gives up one or several guarantees made by strong atomicity

Flavors of Weak Atomicity (1/4)

1) ~~Inside a transaction, multiple accesses to the same variable return the same value provided that no write intervenes.~~

Initially x, y == 0

Thread 1	Thread 2	Thread 3

`atomic {`
`  r1 = x;`
`  r2 = x;`
`}`
	`x = 1;`	`x = 2;`

Can r1 == 1, r2 == 2? Yes!

Flavors of Weak Atomicity (2/4)

~~2) If a variable is written inside an transaction, subsequent reads in the transaction obtain the value that was written.~~

Initially x, y == 0

Thread 1	Thread 2

`atomic {`
`  x = 0;`
`  if (x == 1)`
`    y = 1;`
`}`
	`x = 1;`

Can y == 1? Yes!

Flavors of Weak Atomicity (3/4)

~~3) An intermediate value, which is overwritten in the same transaction or a retry is not visible to other computations.~~

Initially x == 0

Thread 1	Thread 2

atomic {
  x = 1;
  x = 2;
}
	r1 = x;

Can r1 == 1? Yes: "Dirty Read"!

Flavors of Weak Atomicity (4/4)

3) ~~An intermediate value, which is overwritten in the same transaction or a retry is not visible to other computations.~~

Initially x, y == 0

Thread 1	Thread 2

`atomic {`
`  y = 1;`
`  if (x == 0)`
`    retry;`
`}`
	`r1 = y;`
	`atomic {`
	`  x = 1;`
	`}`

Can r1 == 1? Yes: "Speculative dirty read"!

Atomic Blocks vs. Java Synchronized

Initially x == 0

Thread 1	Thread 2

`atomic {`
`  r = x;`
`  x = r+1;`
`}`
	`x = 2;`

Can x == 1?

Strong Atomicity: No!

Any flavor of
weak atomicity: No!

Thread 1	Thread 2

`synchronized(lock) {`
`  r = x;`
`  x = r+1;`
`}`
	`x = 2;`

Can x == 1?

Yes: "Lost Update"!

Ordering Revisited

Accessing the same variable inside and outside a transaction is used in common programming idioms:

- Thread-safe lazy initialization
- Data handoff

Thread-Safe Initialization

Initially flag = false, data = 0;

Initialization (1x) | Unsynchronized read (nx)

Thread 1	Thread 2

`data = 1;`
`mfence;`
`flag = true;`
	`r1 = flag;`
	`if (r1 == true)`
	`  r2 = data;`

Can r1 != 0 && r2 = 0? No!

Idiom works on architectures with processor consistency (X86), resp. TSO (Sparc).

Thread-Safe Initialization with Atomic Block?

Initially flag = false, data = 0;

Thread 1	Thread 2

atomic {
  data = 1;
  flag = true;
}
	r1 = flag;
	if (flag == true)
	  r2 = data;

Can r1 != 0 && r2 = 0?

Some programming languages say yes!, i.e. permit this result (e.g Fortress). Idiom not correct in these languages!

Data Handoff

Initially data = 0 ready = false

Producer	Consumer
Thread 1	Thread 2

```
Thread 1                    | Thread 2
----------------------------|---------------------
data = 42;                  | r1 = false;
atomic {                    | atomic {
  ready = true;             |   r1 = ready;
}                           | }
                            | if (r1) {
                            |   r2 = data
                            | }
```

Can r1 = true && r2 = 0?

Sole purpose of atomic block is to establish synchronization order. It is reasonable to forbid this result (Answer: No!)

Data Handoff

Initially data = 0 ready = false;

Producer	Consumer
Thread 1	Thread 2

data = 42;
atomic {}
ready = true;
	r1 = false;
	atomic {}
	r1 = ready;
	if (r1) {
	  r2 = data
	}

Can r1 = true && r2 = 0?

Answer is not so clear here. If behavior should be forbidden, then empty atomic blocks cannot be eliminated.

Outline

- Problems with Locking
- TM Intro
- Language Integration
- Empirical Studies about TM
- Design Space for TM Semantics
- Design Space for TM Implementations
  - Hardware vs. Software
  - Version Management
  - Conflict Detection
- STM Performance

Hardware vs. Software

Hardware:

+ Efficient
+ Implementation can be based on existing mechanisms for speculative execution
+ Strong atomicity

- Limited capacity for speculative state
- Limited flexibility for different policies, e.g., contention management
- ISA extensions not obvious

Software:

- slow, efficient implementations compromise on semantics (weak atomicity)

Hardware vs. Software

Hardware:

+ Efficient
+ Implementation can be based on existing mechanisms for speculative execution
+ Strong atomicity
- Limited capacity for speculative state
- Limited flexibility for different policies, e.g., contention management
- ISA extensions not obvious [McDonaldEtAl'06]

Software:

This talk: Software Transactional Memory (STM)

- slow, efficient implementations compromise on semantics (weak atomicity)

Hardware vs. Software

Hybrid TM:

- Baseline operation in hardware
- Fallback to software on critical cases (buffer overflow, obstinate contention)

Hardware-accelerated STM [SahaEtAl'06]:

- Starting point is STM
- Selected, frequent STM operations are accelerated with hardware primitives.

STM Design Space

- Version management
- Conflict detection
  - Consistent versus inconsistent views
  - Visible versus invisible reads
  - Contention management
- Blocking versus non-blocking progress
- Engine-room issues …

Version Management

Lazy (redo logs)

- Writes go to log, not to memory
- Reads require look-aside
- Apply changes on commit
- Rolling back wedged transaction easy

Eager (undo logs)

- Update in place (leads to weak atomicity)
- Reads are fast
- Rolling back wedged transaction complex

Conflict Detection

Eager

- conflict with other transaction detected as soon as read would return inconsistent value.
- expensive

Lazy

- Validation of read-set at commit time
- Orphan transactions: another txn wrote into current txn's read set
  - Can orphans observe inconsistent views?

Do Orphan (Zombie) Transactions Always See Consistent States?

Yes!

- Invariants observed (no surprises)
- Expensive (maybe)

No!

- Who cares about surprises?
  - Divide by zero, infinite loops, et cetera ...
  - Use exception/interrupt handlers?
- More efficient (maybe)

Source: [Herlihy&Shavit'08]

Read Synchronization

Visible (mark objects)

- Consistent views
- Additional info for contention management
- Quick validation
- Slower overall (maybe)

Invisible (no footprint)

- Inconsistent views
- Slow validation
- Faster overall (maybe)

Contention Management

Choice of policy can have significant impact on application performance [Scherer&Scott'04].

- "Aggressive": txn aborts other conflicting txn at commit time.
- "Polite": txn aborts itself on conflict and backs off.
- "Timestamp": on conflict, younger txn is aborted.
- … <many more>

Blocking vs. Non-Blocking Progress

Blocking

- Delay of one thread can delay other threads
- Internals based on fine-granular locks
- Design choice of many recent STMs

Non-blocking

- Validation and commit based on lock-free algorithms
- Different progress guarantees (obstruction-free, ..., wait-free): delay only due to contention.
- Slower overall (maybe)

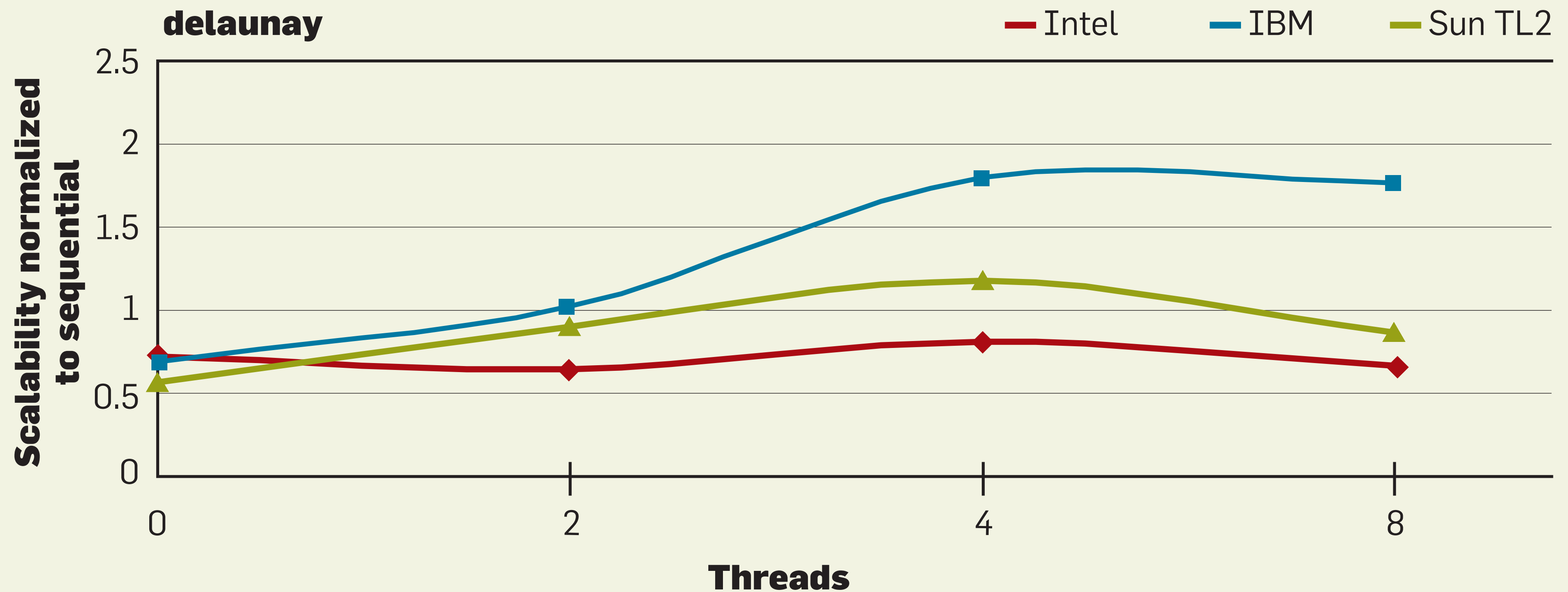
Engine Room Issues …

- Levels of indirection
- Compatibility with HTM
- There's lots more …

Source: [Herlihy&Shavit'08]

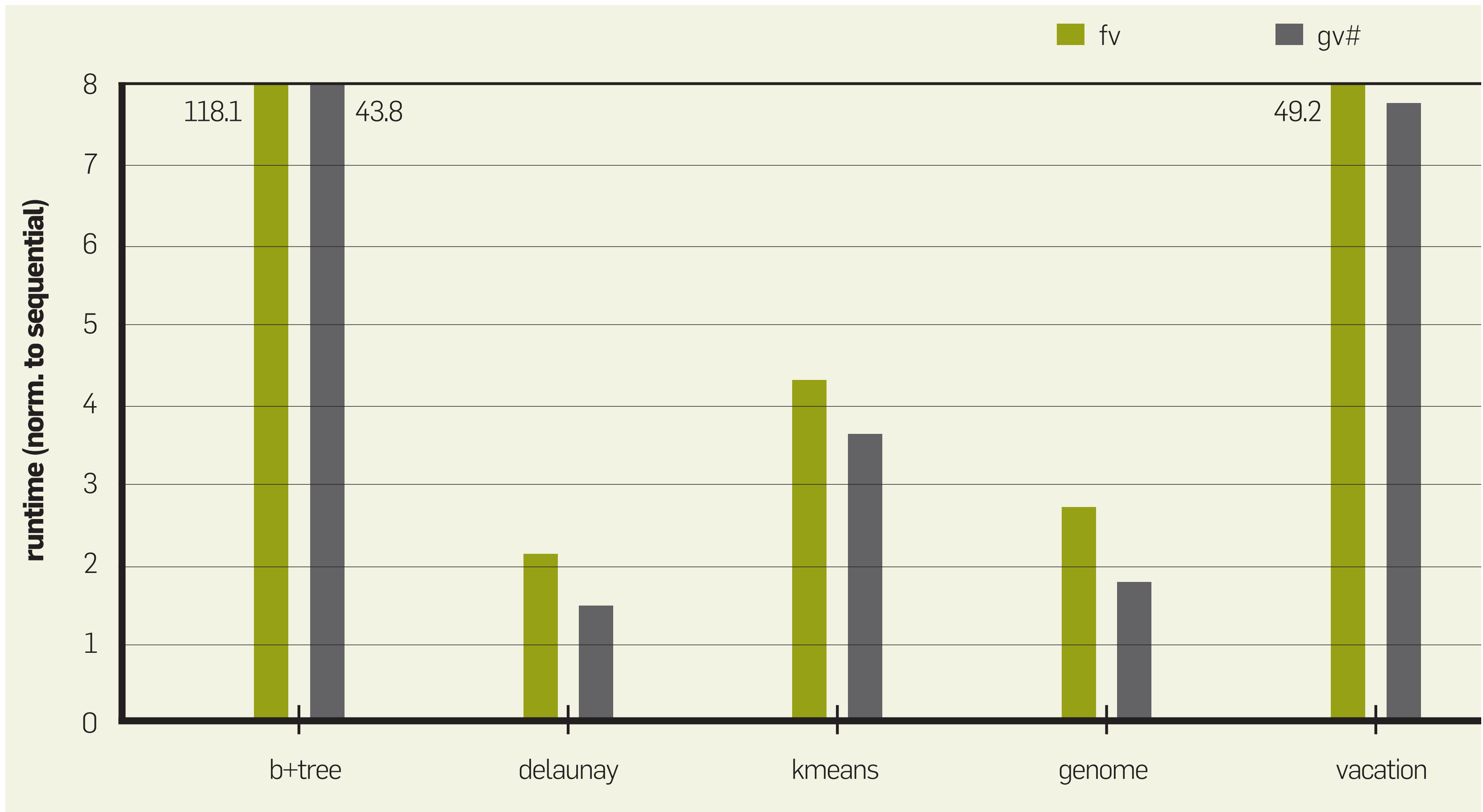
- Problems with Locking
- TM Intro
- Language Integration
- Empirical Studies about TM
- Design Space for TM Semantics
- Design Space for TM Implementations
  - Hardware vs. Software
  - Version Management
  - Conflict Detection
- STM Performance

- Sample data from IBM's STM [CascavalEtAl'08]
- Two different algorithms:
  - fv (full validation),
  - gv# (global version number)
- Metrics:
  - Scaling
  - Single-thread overhead
  - Components of overhead



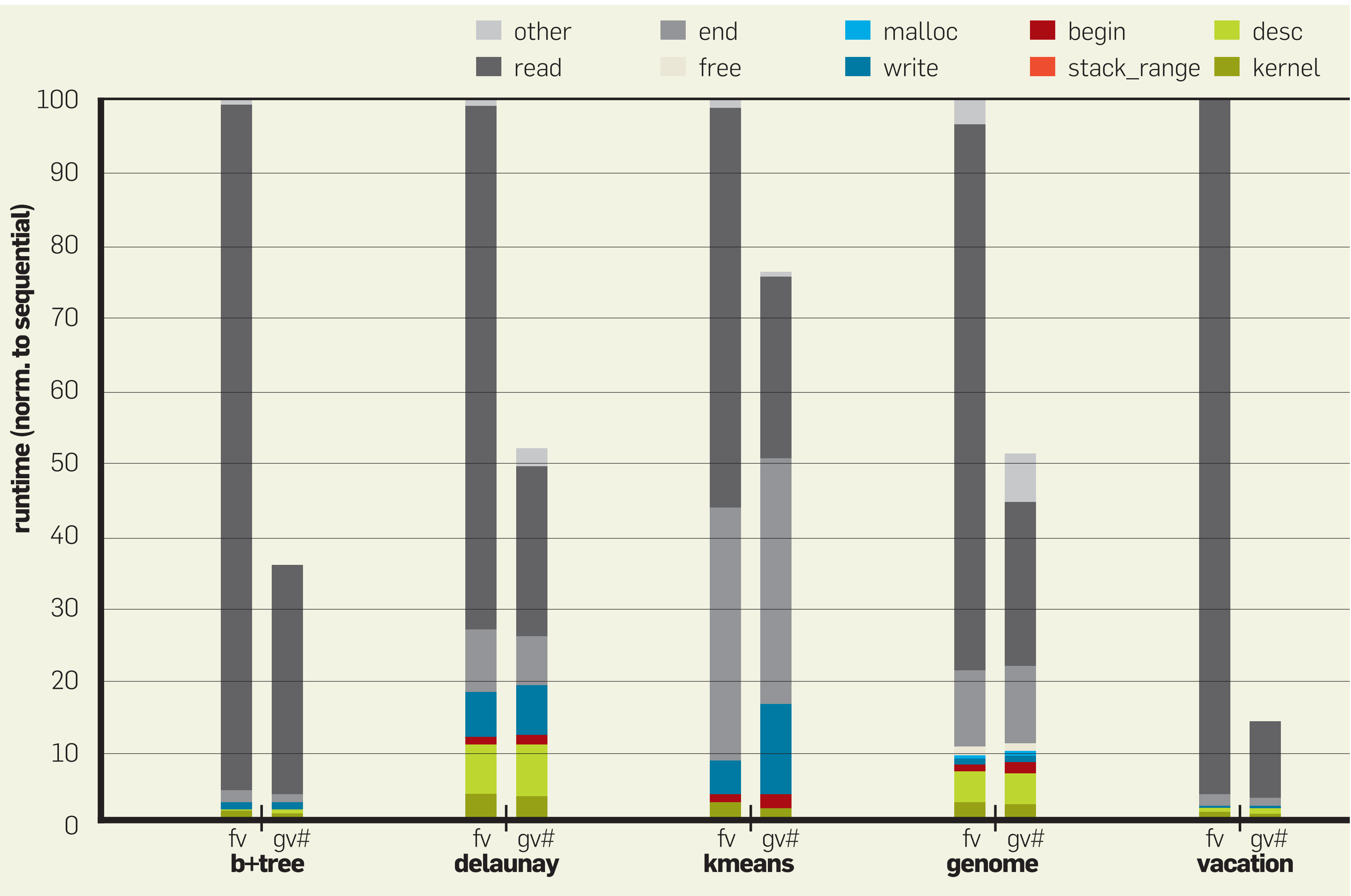
Scalability of the delaunay application. Baseline is the sequential code without synchronization.

Source: [CascavalEtAl'08]



Single-thread overhead of fv and gv# algorithms for different applications.

Source: [CascavalEtAl'08]



Fraction of components in STM single-thread overhead.

Source: [CascavalEtAl'08]

Performance: Take-away

- Top-contributors to overhead:
  - read barrier (read)
  - commit (end)
- Hardware can help to accelerate read-set validation
  - Intel's architecture with thread-local mark bits in cache [SahaEtAl'06]
  - Even then: significant overheads remain that cannot be attributed to a single source / optimization opportunity

Remember This

- TM is a real step forward in parallel programming methodology
- TM does not solve parallel programming menace
  - Focus on task-parallel shared memory
  - Parallel still more difficult than sequential programming
  - Buggy programs are easily possible
- TM is a hot research area. Challenges:
  - Language integration: TM semantics, debugging, ...
  - For STM: performance, performance, performance

Sources

[Herlihy&Shavit'08] Maurice Herlihy, Nir Shavit: Companion Slides "The Art of Multiprocessor Programming", Licence: http://creativecommons.org/licenses/by-sa/3.0/

[Amarasinghe'07] Saman Amarasinghe: Lecture on "Introduction to Parallel Architectures", MIT 2007.

[GrossmannEtAl'06] Dan Grossmann, Jeremy Manson, William Pugh: Lecture on "What do high-level memory models mean for transactions?", MSPC, 2006.

[Scherer&Scott'04] William N. Scherer III, Michael Scott: "Contention management in dynamic software transactional memory", CSJP 2004.

[CacsavalEtAl'08] C. Cascaval et al.: "Software transactional memory: Why is it only a research toy", Communications of the ACM 51/11, 2008.

[SahaEtAl'06] B. Saha et al.: "Architectural support for software transactional memory", IEEE MICRO, 2006.

Sources

[KulkarniEtAl'06] M. Kulkarni, L. P. Chew, K. Pingali: "Using transactions in delaunay mesh generation", WTW, 2006.

[RossbachEtAl'09] Ch. Rossbach, O. S. Hofmann, E. Witchel: "Is transactional programming actually easier?", WDDD, 2009.

[LuEtAl'08] S. Lu, S. Park, E. Seao, Y. Zhou: "Learning from mistakes – A comprehensive study on real world concurrency bug characteristics", ASPLOS, 2008.

[McDonaldEtAl'06] A. McDonald, et al.: "Architectural Semantics for Practical Transactional Memory" , ISCA, 2006.

Vielen Dank!

Christoph von Praun
Geoorg-Simon-Ohm Hochschule, Nürnberg

Backup

Delaunay Mesh Refinement

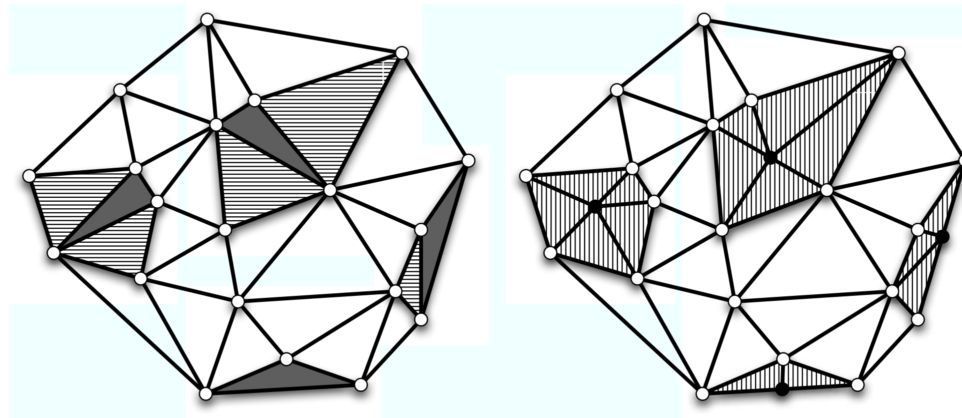


Figure 4. An example of processing several elements in parallel. The left mesh is the original mesh, while the right mesh represents the refinement. In the left mesh, the *dark grey* triangles represent the "bad" elements, while the *horizontally shaded* are the other elements in the cavity. In the right mesh, the the *black* points are the newly added points and *vertically shaded* triangles are the newly created elements.

Source: [KulkarniEtAl'06]

This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free**:
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions**:
  - **Attribution**. You must attribute the work to "The Art of Multiprocessor Programming" (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike**. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.