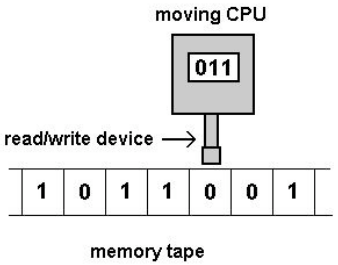
On the Benefits of Being Optimistic and Relaxed

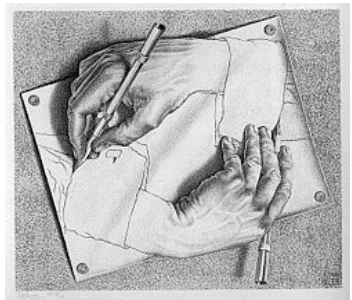
Petr Kuznetsov

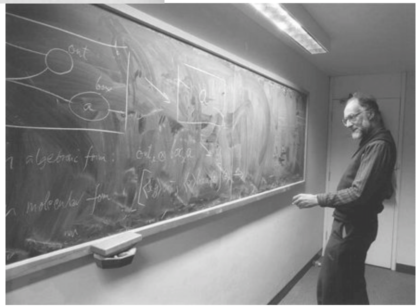
INFRES, Télécom ParisTech

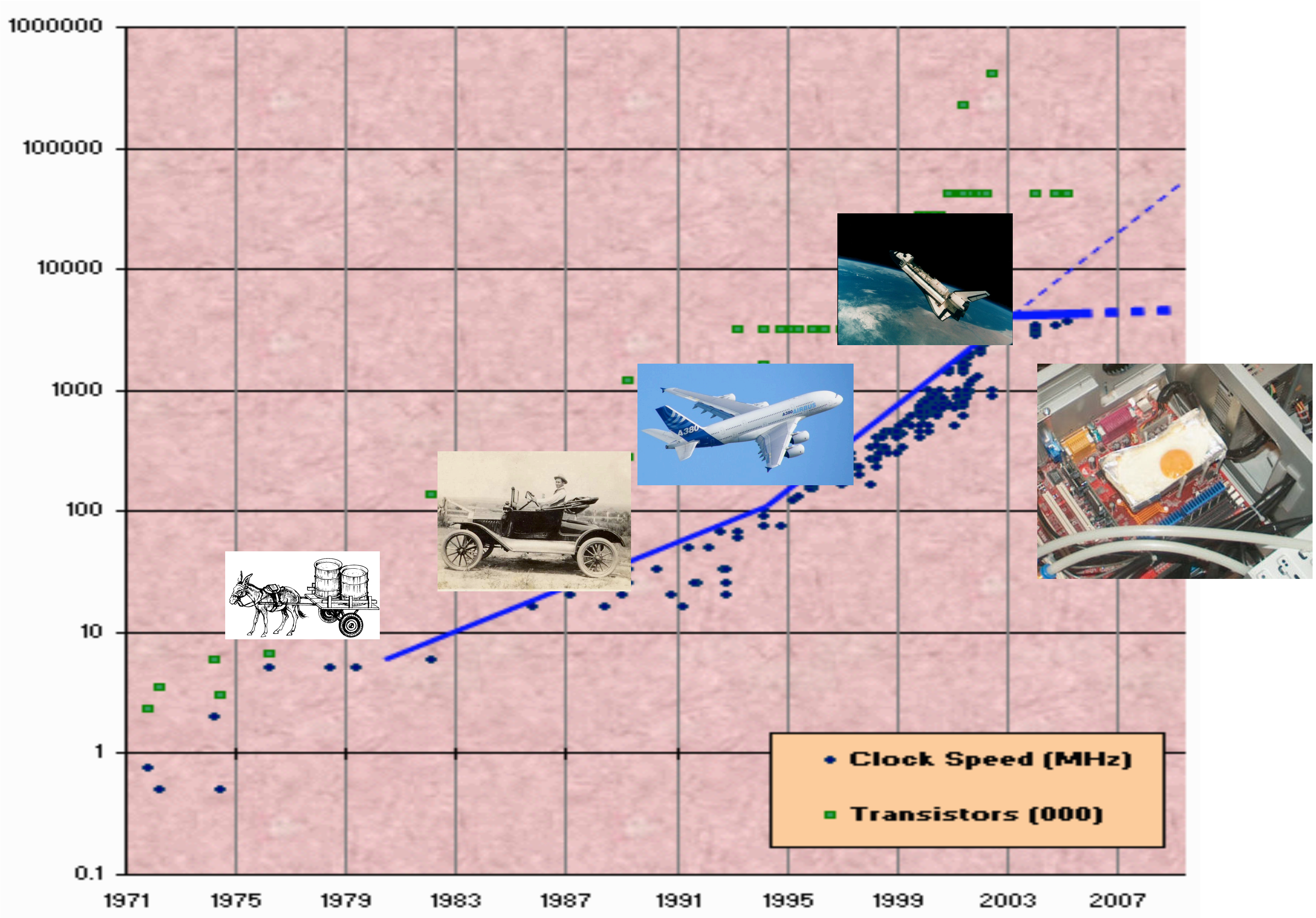
Joint work with Srivatsan Ravi (TU Berlin) and Vincent Gramoli (U Sydney)

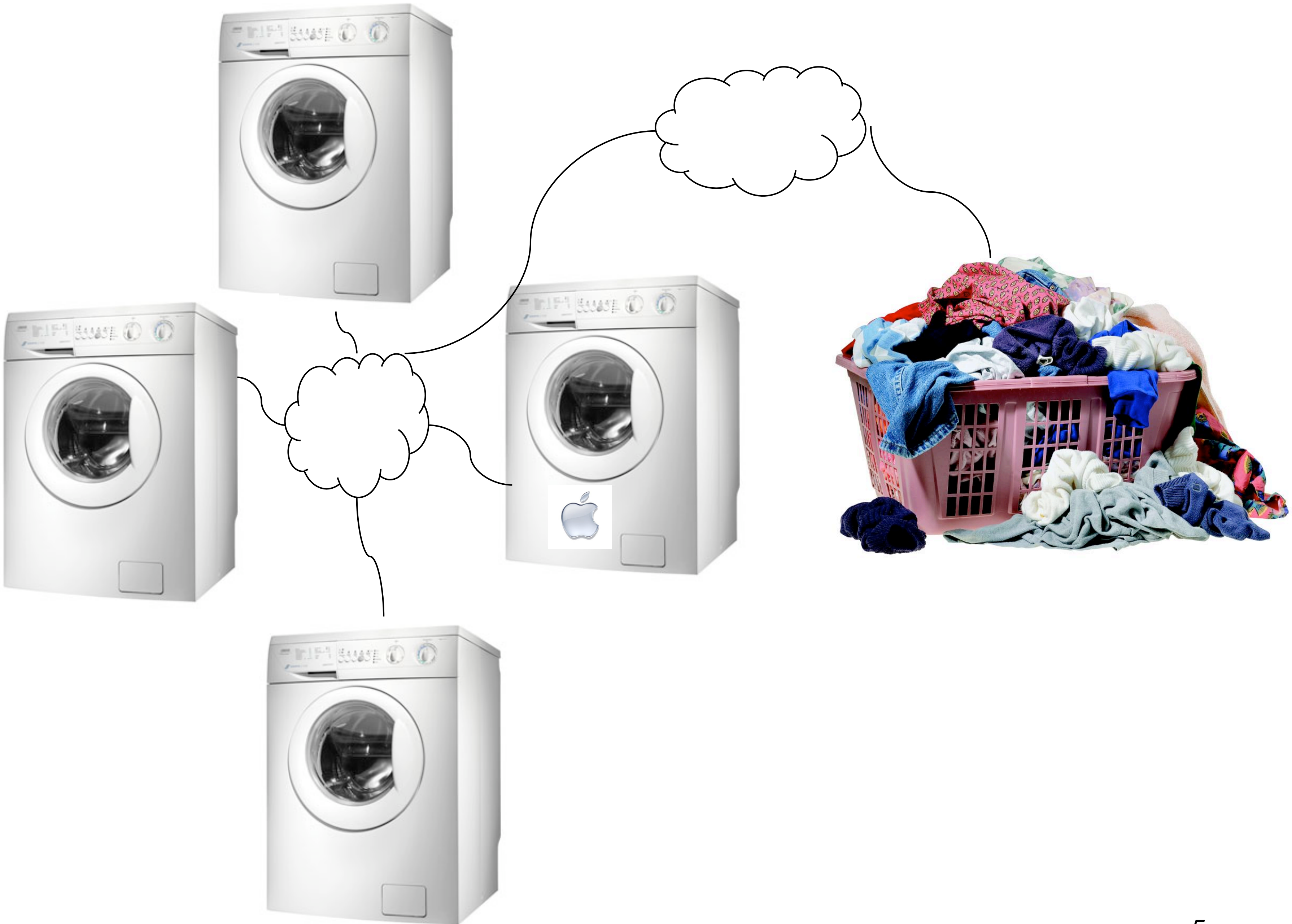
What is computing?

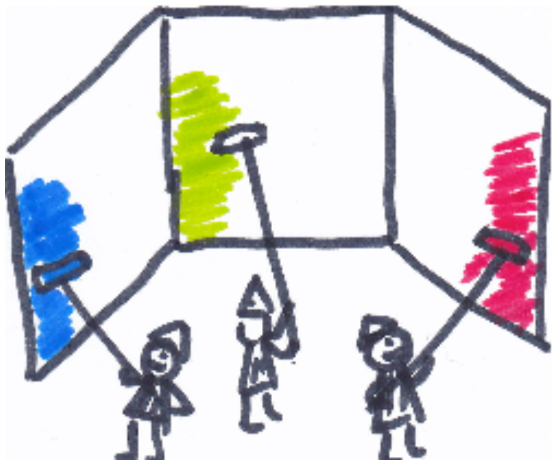


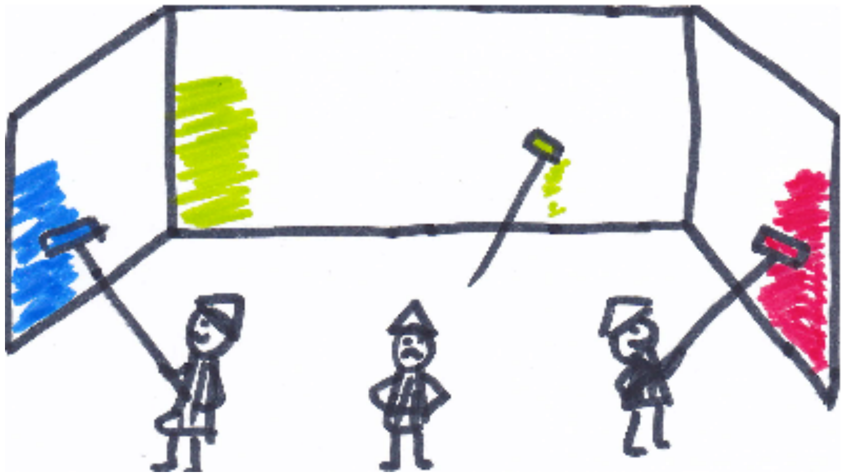





1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
Clock Speed (MHz)
Transistors (000)







$$S = \frac{1}{1 - p + p/n}$$

Amdahl's Law

- *p* – fraction of the work that can be done in parallel (no synchronization), 1-p for synchronization
- *n* - the number of processors

$$S = \frac{1}{1 - p + p/n}$$

For n=9, p=9/10, S=5! S<9, regardless of n!

Minimizing synchronization costs is crucial!

But…

- Concurrent programming is hard
  - ✓A new algorithm worth a PhD (or a paper at least)
- Sequential programming is "easy"
  - ✓e.g., for data structures (queues, trees, skip lists, hash tables,…)
- What about a "wrapper" that allows for running sequential operations concurrently?
  - ✓Let the wrapper care about conflicts
- How? Locks, transactional memory…

Our contribution

- What it means to share a sequential program
  ✓ Locally serializable (LS) linearizability

- What it means for sharing to be efficient
  ✓ Relative concurrency of different synchronization techniques (e.g., locks vs. TMs)

- What are the benefits of being relaxed and optimistic
  ✓ Type-specific (relaxed) consistency and transactional (optimistic) concurrency control supersede both

Correctly sharing sequential code?

Given a sequential implementation P of a data structure type T, provide a concurrent one that:

- Locally appears sequential – the user simply runs the sequential code of P
  - ✓Local serializability
- Globally makes sense – the high-level operations give consistent global order wrt T
  - ✓Linearizability [HW90]

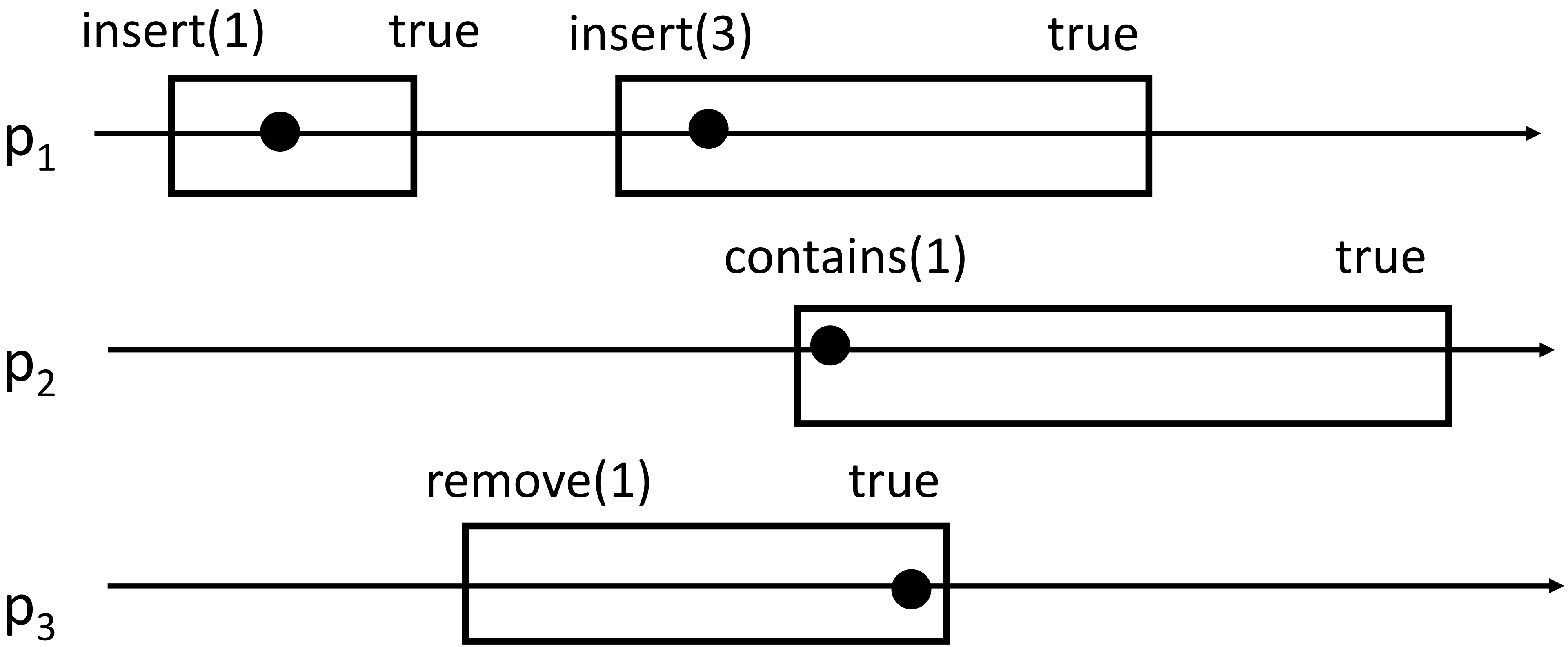
Example: Integer Set

Type: Integer Set:

- *boolean insert(x)*
- *boolean remove(x)*
- *boolean contains(x)*

Implemented sequentially as a sorted linked list:

Linearizable histories



The history is equivalent to a legal sequential history on a set (real-time order preserved)

Linked-list for Integer Set: sequential implementation

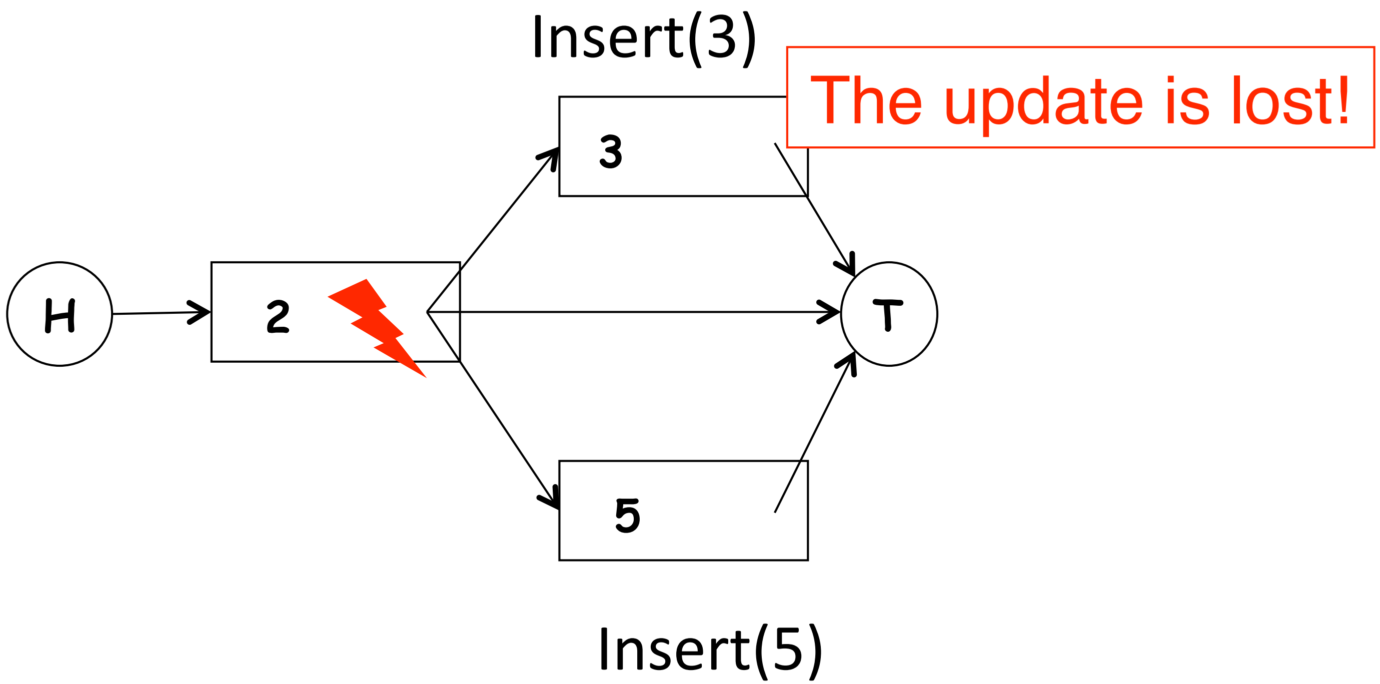
Locate(s):
 $prev \leftarrow Head$
 $curr \leftarrow prev.next$
 while $(curr.val < s \vee curr = Tail)$ **do**
  $prev \leftarrow curr$
  $curr \leftarrow curr.next$
 end while
 return $\langle prev, curr \rangle$

Contains(s):
 $\langle prev, curr \rangle \leftarrow$ Locate(s)
 if $curr.val = s$ **then**
  $result \leftarrow$ true
 else
  $result \leftarrow$ false
 return $result$

Insert(s):
 $\langle prev, curr \rangle \leftarrow$ Locate(s)
 if $curr.val \neq s$ **then**
  $X_s \leftarrow$ new-node(s)
  $X_s.next \leftarrow curr$
  $prev.next \leftarrow X_s$
 return ok

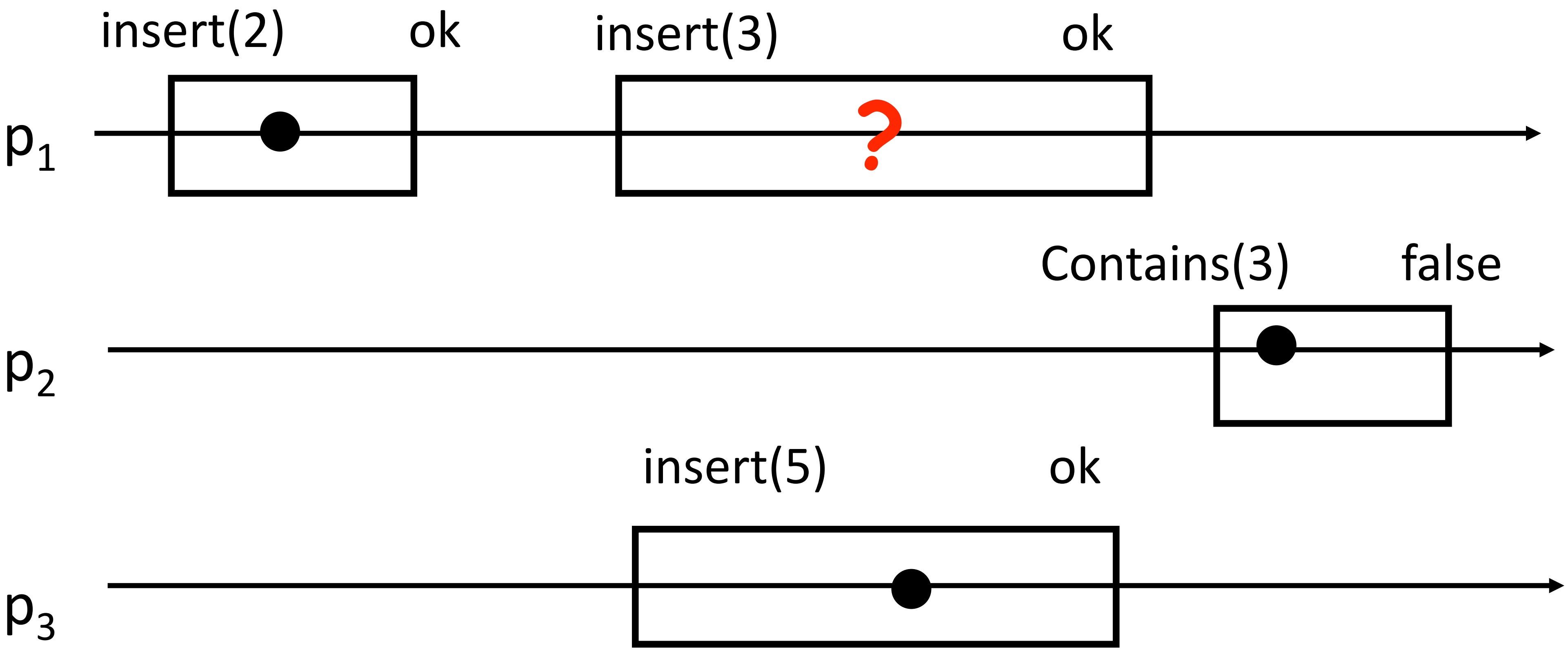
Remove(s):
 $\langle prev, curr \rangle \leftarrow$ Locate(s)
 if $curr.val = s$ **then**
  $curr.marked \leftarrow$ true
  // Mark node for removal
  $prev.next \leftarrow curr.next$
 return ok

As is?



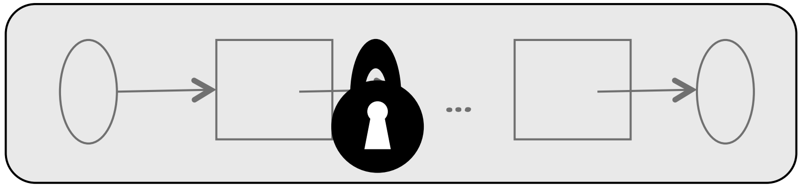
Not LS-linearizable: locally serializable,

but not linearizable…



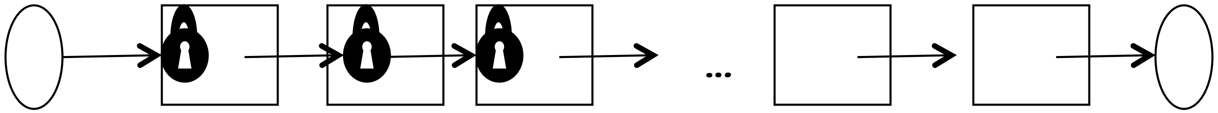
- protect data items (critical sections),
- provide roll-backs (transactions)

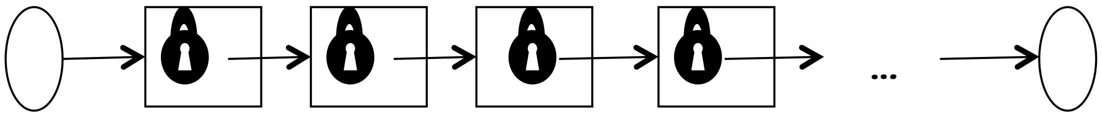
Locking schemes for a linked-list



Coarse-grained locking

2-phase locking





Hand-over-hand locking: relaxed for specific data structures

Optimistic wrapper for a linked-list

```
startTxn
Contains(s):
    ⟨prev, curr⟩ ← Locate(s)
    if curr.val = s then
        result ← true
    else
        result ← false
    return result
tryCommit
```

```
startTxn

Insert(s):
    ⟨prev, curr⟩ ← Locate(s)
    if curr.val ≠ s then
        X_s ← new-node(s)
        X_s.next ← curr
        prev.next ← X_s
    return ok
tryCommit
```

```
startTxn

Remove(s):
    ⟨prev, curr⟩ ← Locate(s)
    if curr.val = s then
        curr.marked ← true
        // Mark node for removal
        prev.next ← curr.next
    return ok

tryCommit
```

Plenty of STMs exist:

- Opaque
- Strictly serializable
- Elastic
- …

What about efficiency?

"Amount of concurrency": the sets of accepted schedules (orderings of sequential steps)

- ✓Each (correct) implementation accepts a subset of schedules
- ✓The more schedules are accepted the better

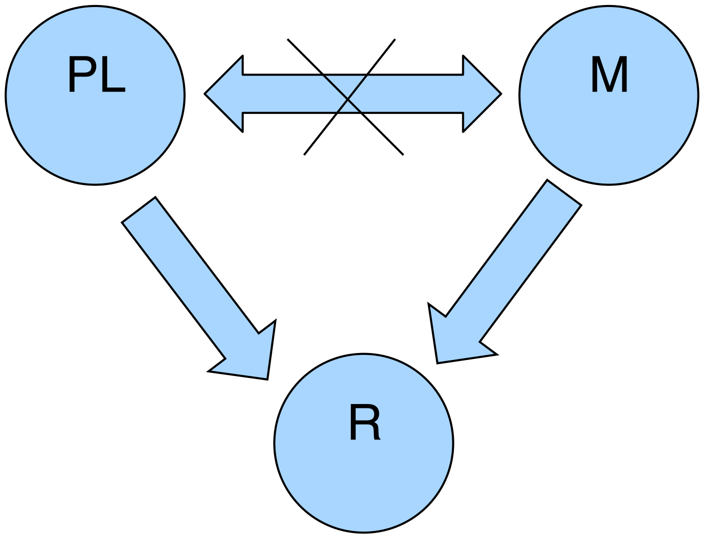
Which technique provides most concurrency?

Relaxation vs. Optimism

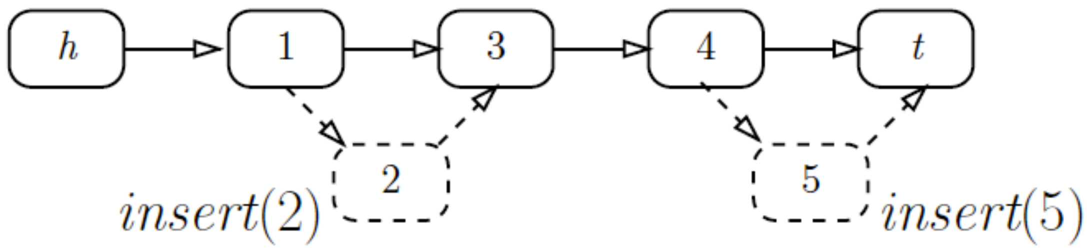
PL: deadlock-free (fine-grained) lock-based

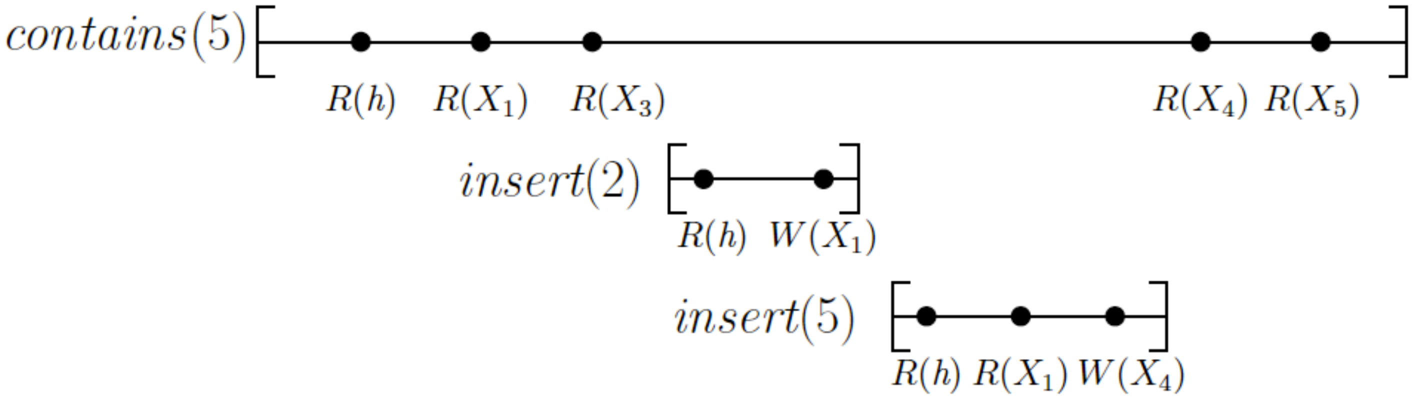
M: strongly consistent (serializable) TMs

R: relaxed (data-type-aware) TMs

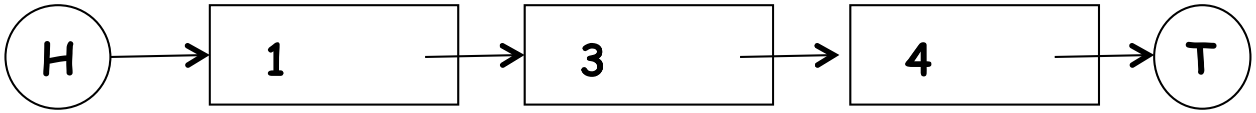


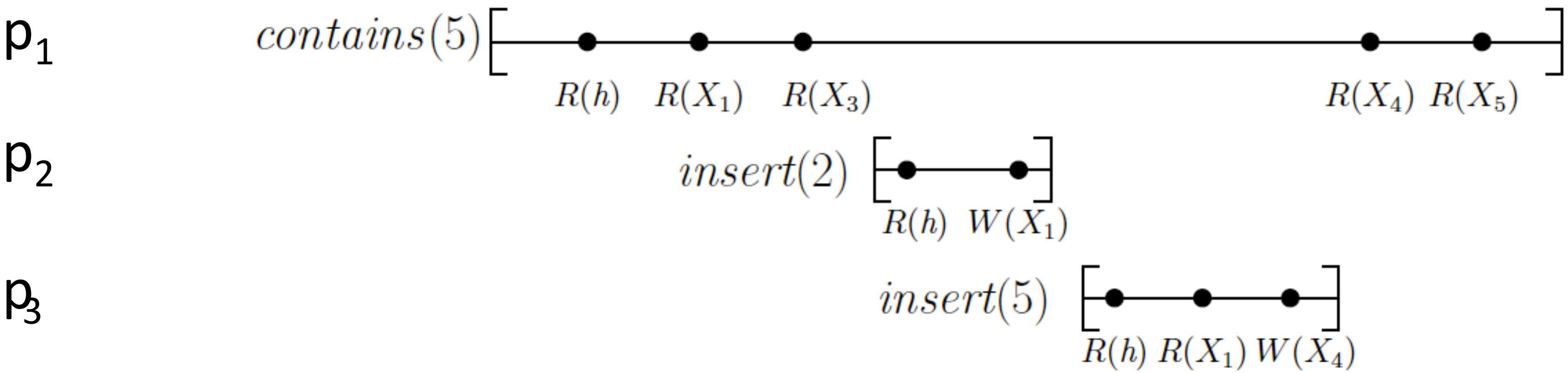
$M \not\preceq_{(LL,set)} PL$





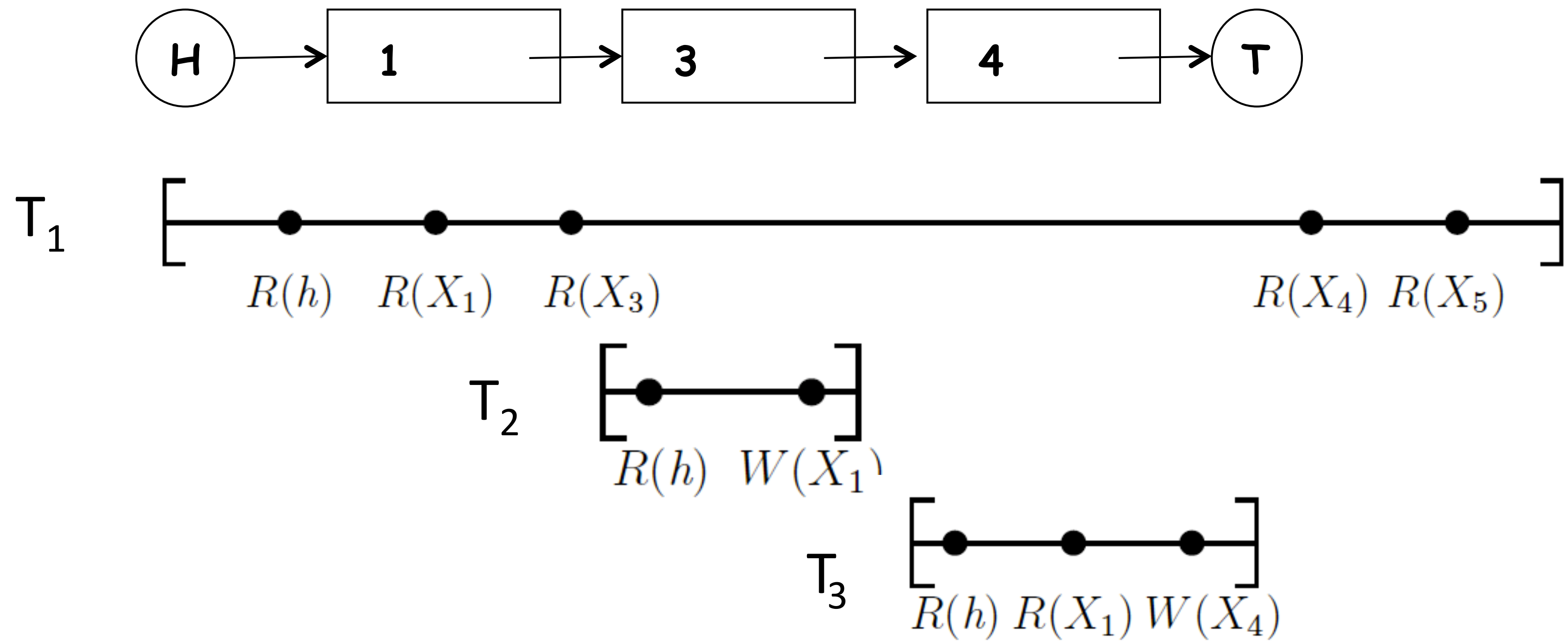
Accepted by locks





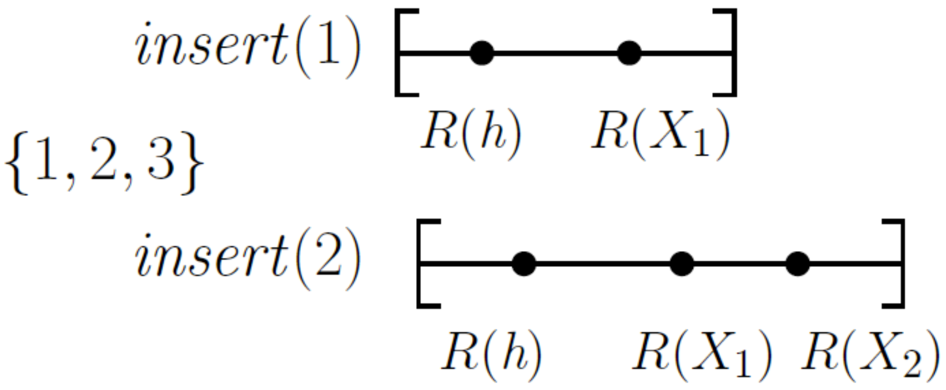
- Accepted by hand-over-hand
  - ✓ p_1 and p_3 are consistent with different serializations

But not serializable!



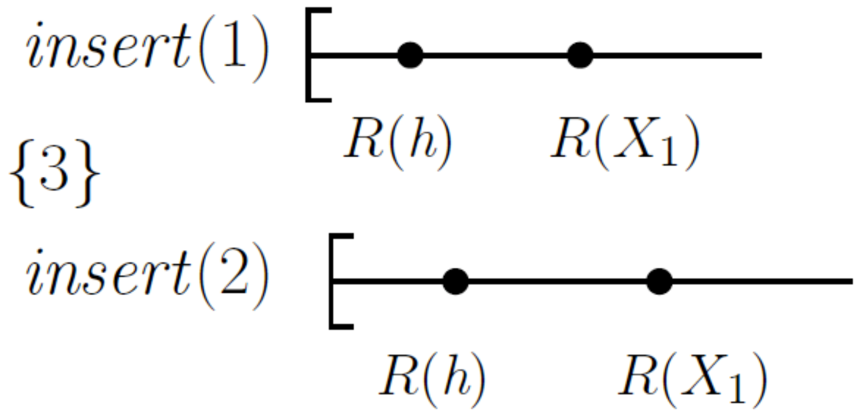
- T_1->T_2 (T_1 read X_1 before T_2 updated it)
- T_2->T_3 (T_3 sees the effect of T_2)
- T_3-> T_1 (T_1 sees the effect T_3)

$PL \not\preceq_{(LL,set)} M$



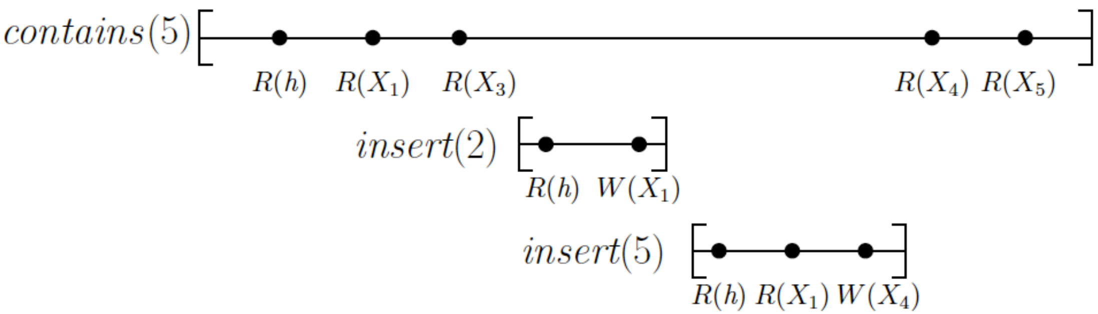
No writes: accepted by M

- Both ops are about to write: rejected by M (at least one txn aborts)
- But must be accepted by PL too!

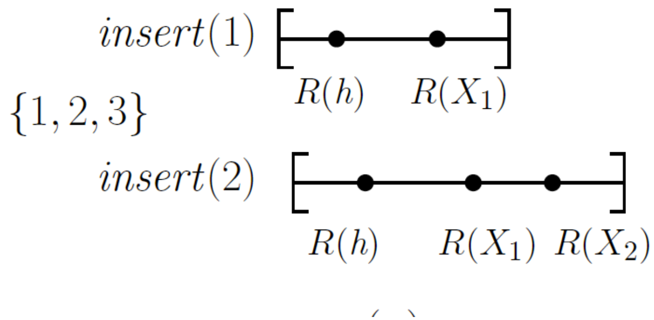


$$M \prec_{(LL,set)} R \qquad PL \prec_{(LL,set)} R$$

- *R* accepts every observable correct schedule of *(LL,set)*
- *R* accepts the linearizable but not serializable schedules of non-conflicting updates (1)
- *R* accepts the potentially conflicting-update schedule (2)



(1)



(2)

Results and implications

- What is a correct concurrent wrapper?
  - ✓LS-linearizability
- How to measure relative efficiency of concurrent wrappers?
  - ✓Accepted schedule sets
- Benefits of relaxation and optimism formally captured
- A language to reason about the “best” synchronization technique, the “most suitable” data structure

Open questions

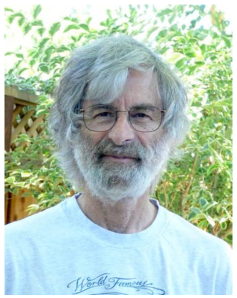
- Extend the results to more general classes of types, data structures
- Workload analysis: which schedules are relevant?
- What concurrency tells us? The cost of concurrency?

Details: http://arxiv.org/abs/1203.4751

Distributed ≠ Parallel

- The main challenge is efficient and robust synchronization

- “you know you have a distributed system when the crash of a computer you’ve never heard of stops you from getting any work done” (Lamport)



Merci beaucoup!