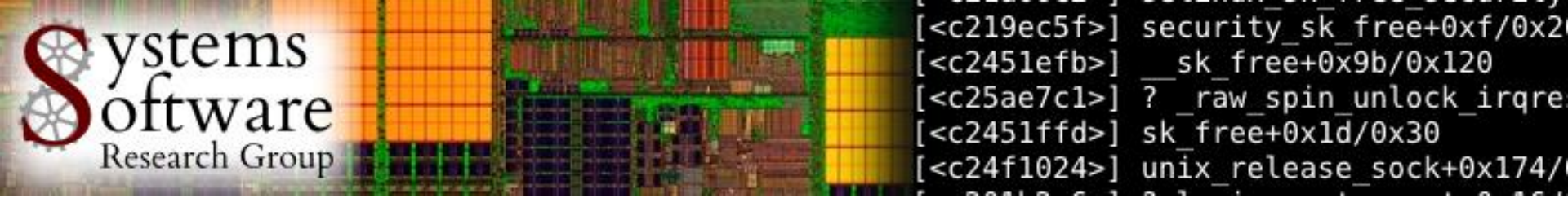


On Real-Time STM Concurrency Control for Embedded Software with Improved Schedulability

Mohammed Elshambakey and Binoy Ravindran

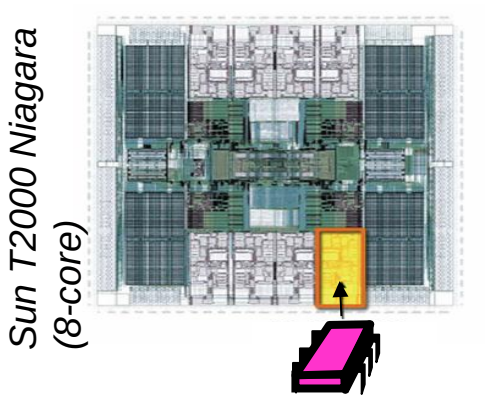
Virginia Tech
USA
{shambake,binoy}@vt.edu

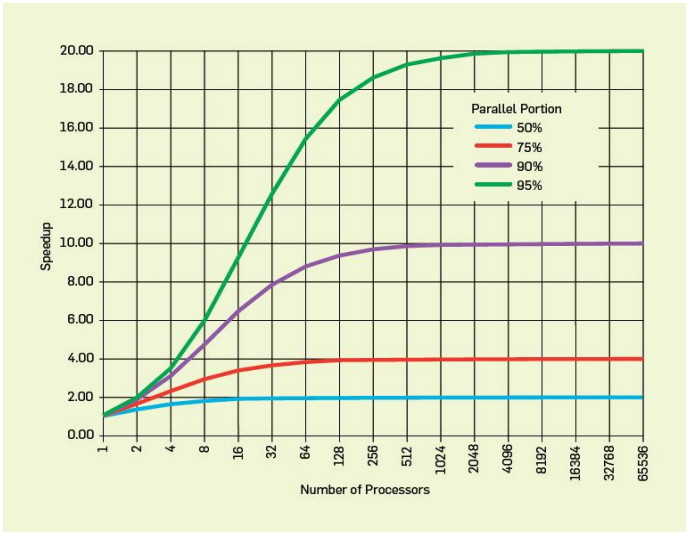
ASP-DAC 2013

Virginia Tech

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*





- Significant implications for embedded real-time software
  - Inherently concurrent – react to multiple streams of sensor input and control multiple actuators
  - Often concurrently read/write shared data objects

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
- Fine-grained performance
- Composable

- First HTM, then STM, later HyTM

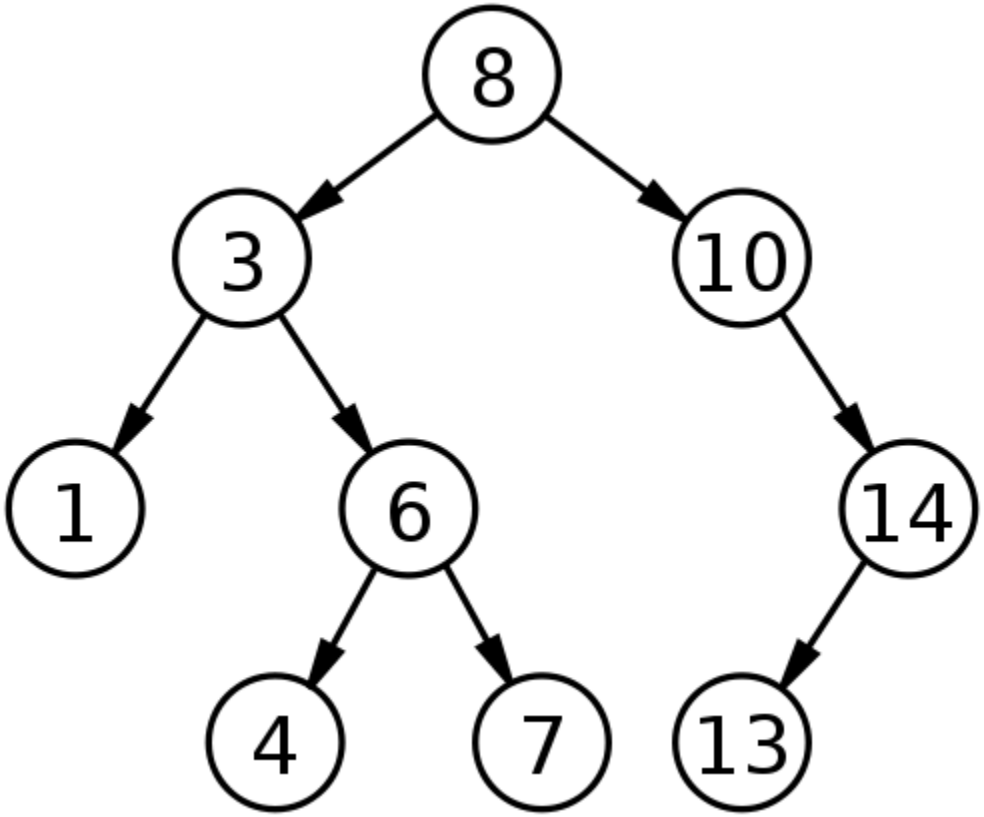
```
public boolean add(int item) {
  Node pred, curr;
  atomic {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;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

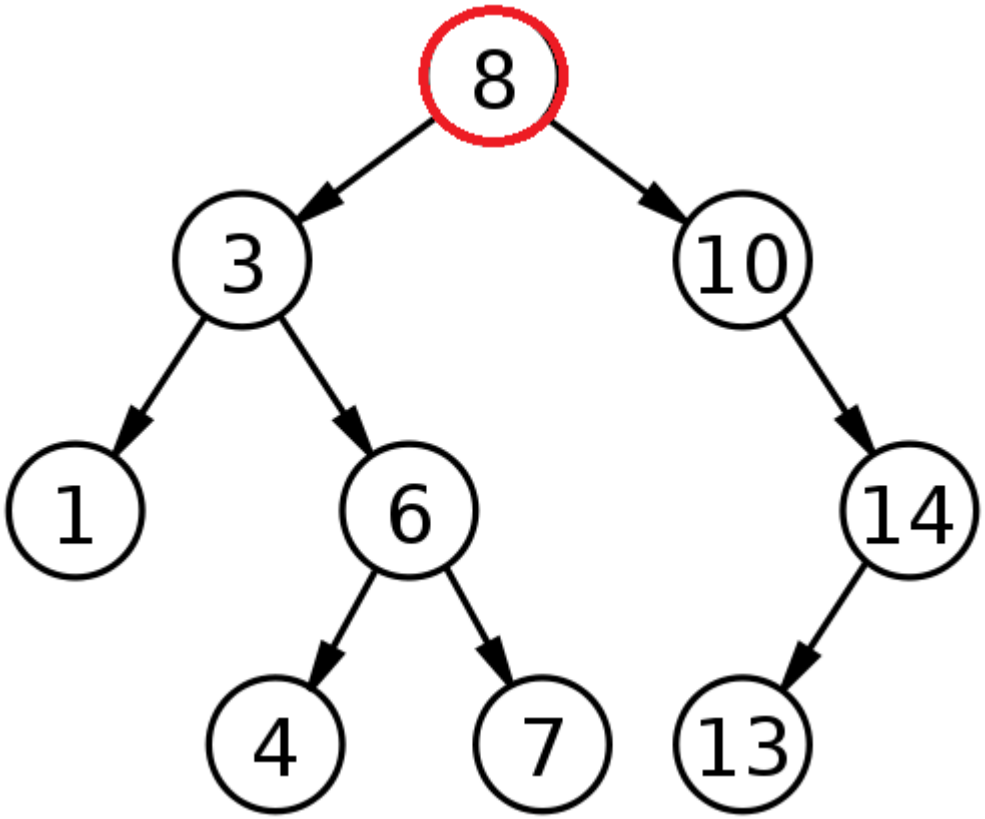
N. Shavit and D. Touitou (1995). Software Transactional Memory. *PODC*. pp. 204—213.

How does TM work?

- Idea: speculate
- Example: add 9 and 15 concurrently



Thread A adds 9 and thread B adds 15



Thread A

Read-set: 8

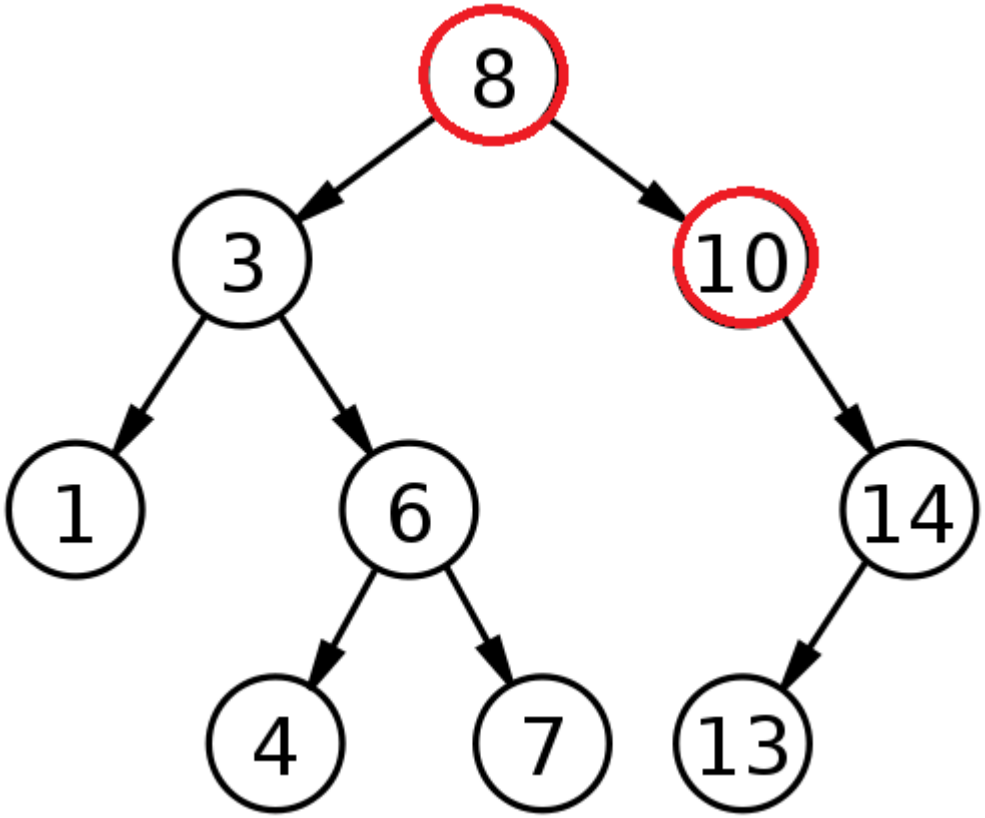
Write-set:

Thread B

Read-set: 8

Write-set:

Thread A adds 9 and thread B adds 15



Thread A

Read-set: 8,10

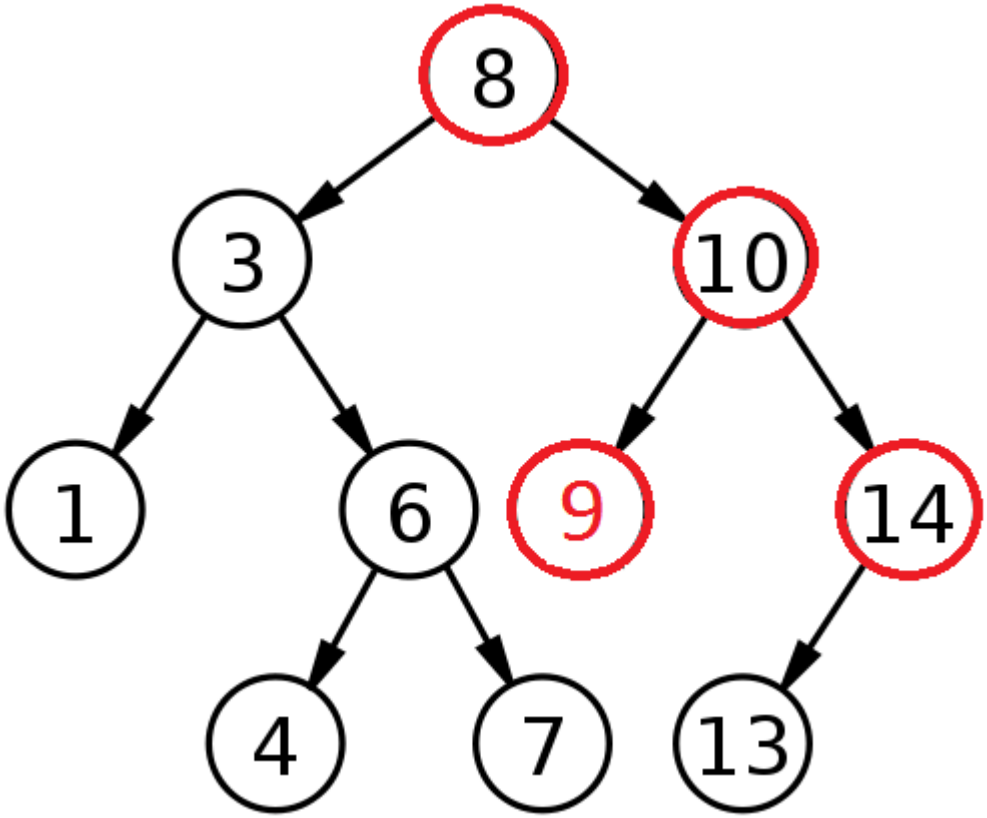
Write-set:

Thread B

Read-set: 8,10

Write-set:

Thread A adds 9 and thread B adds 15



Thread A

Read-set: 8,10

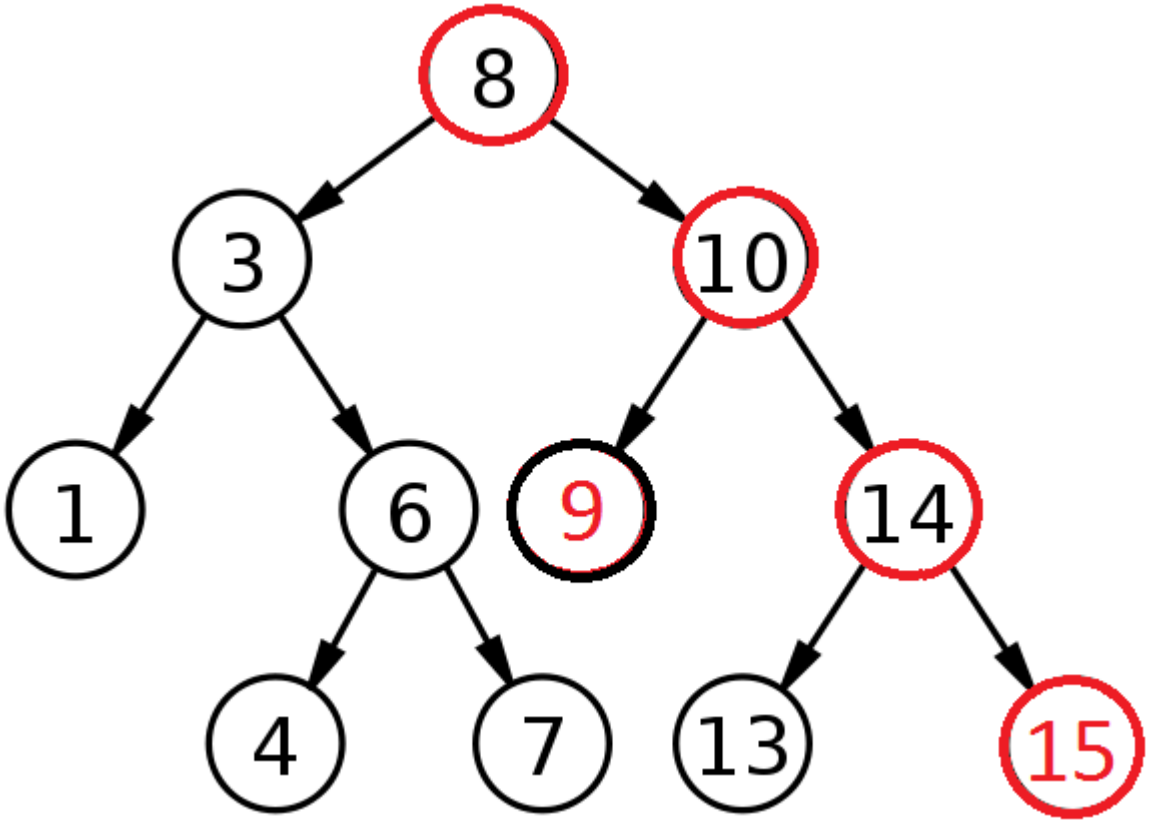
Write-set: 10 (LC)

Thread B

Read-set: 8,10,14

Write-set:

Thread A adds 9 and thread B adds 15



Thread A

Read-set: 8,10

Write-set: 10 (LC)

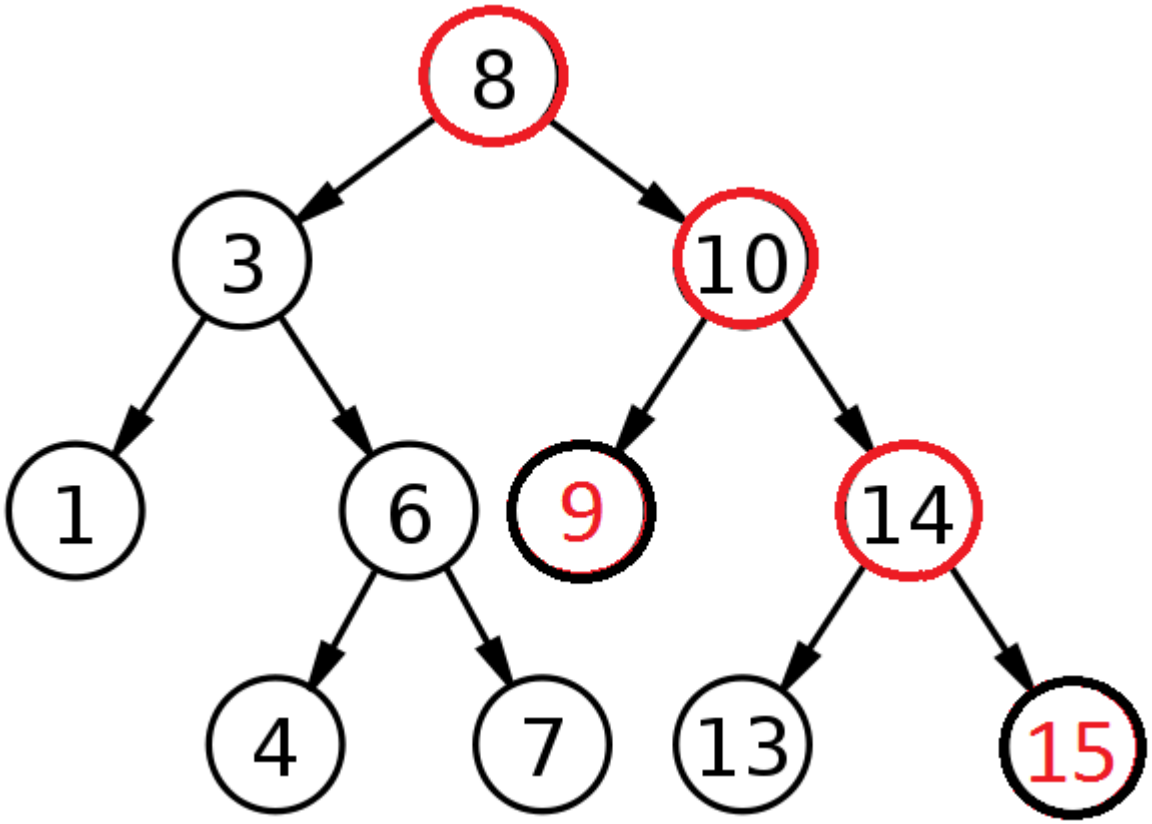
(Committed successfully)

Thread B

Read-set: 8,10,14

Write-set: 14 (RC)

Thread A adds 9 and thread B adds 15



Thread A

Read-set: 8,10

Write-set: 10 (LC)

(Committed successfully)

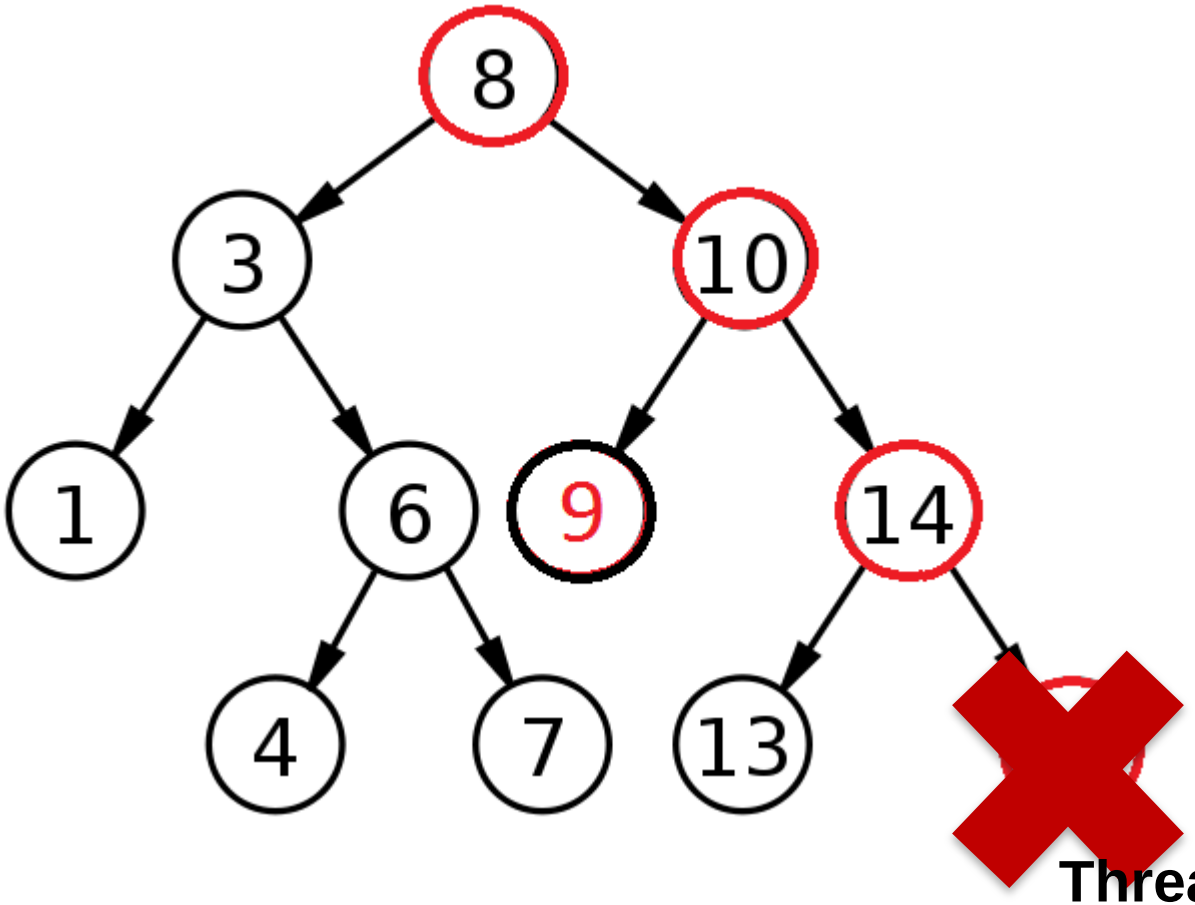
Thread B

Read-set: 8,10,14

Write-set: 14 (RC)

(Committed successfully)

Object-based granularity causes conflict (in this case)



Thread A

Read-set: 8,10

Write-set: 10

(Committed successfully)

Write-after-read

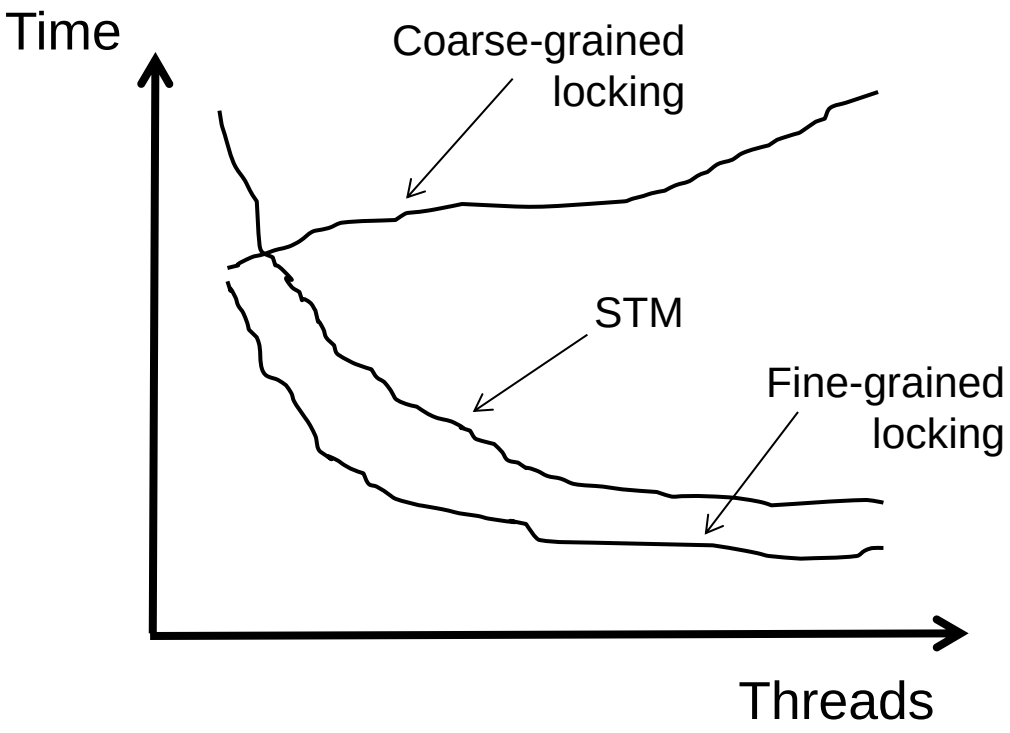
Thread B

Read-set: 8,10,14

Write-set: 14

(Conflict, so abort and retry)

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



- 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`

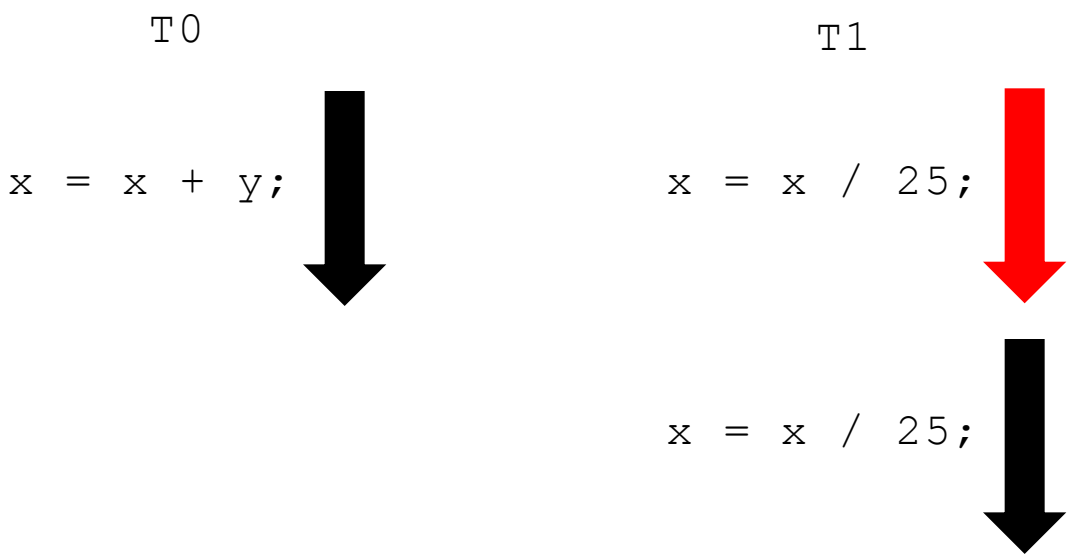
```
atomic{
    x = x / 25;
}
```

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 determines transactional progress

Conflict resolution or contention management



Which transaction to abort?

- Polite: familiar exponential backoff
- Greedy: favor those with an earlier start time
- Karma: ….

(Starvation may occur)

Wait-free progress is necessary for real-time STM

- Bound retry cost to satisfy time constraints
- Rely on existing versioning and conflict detection techniques

- Contention management directly affects transactional progress
  - Starvation-freedom is necessary

Paper's contribution

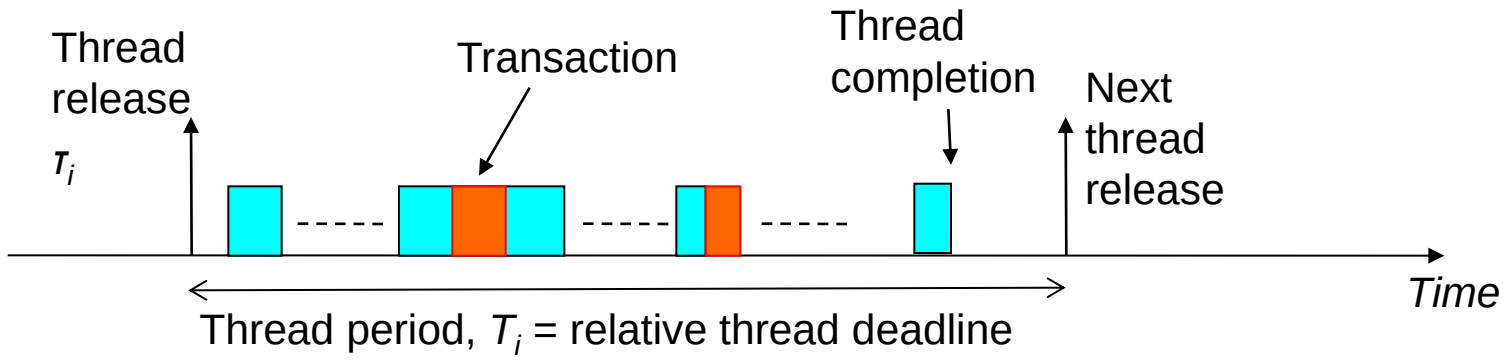
- Prior research have developed real-time contention managers, with bounded retry costs and response times
  - Earliest deadline first contention manager (ECM)
  - Rate monotonic contention manager (RCM)
  - Length-based contention manager (LCM)

M. El-Shambakey and B. Ravindran. STM concurrency control for embedded real-time software with tighter time bounds. In *DAC*, pages 437–446, 2012

- But they restrict to *one* object access per transaction
- Paper presents new contention manager: PNF
  - Allows multiple objects per transaction
  - Bounds retry costs and response times
- Formal comparison with past CMs and lock-free
- Implementation

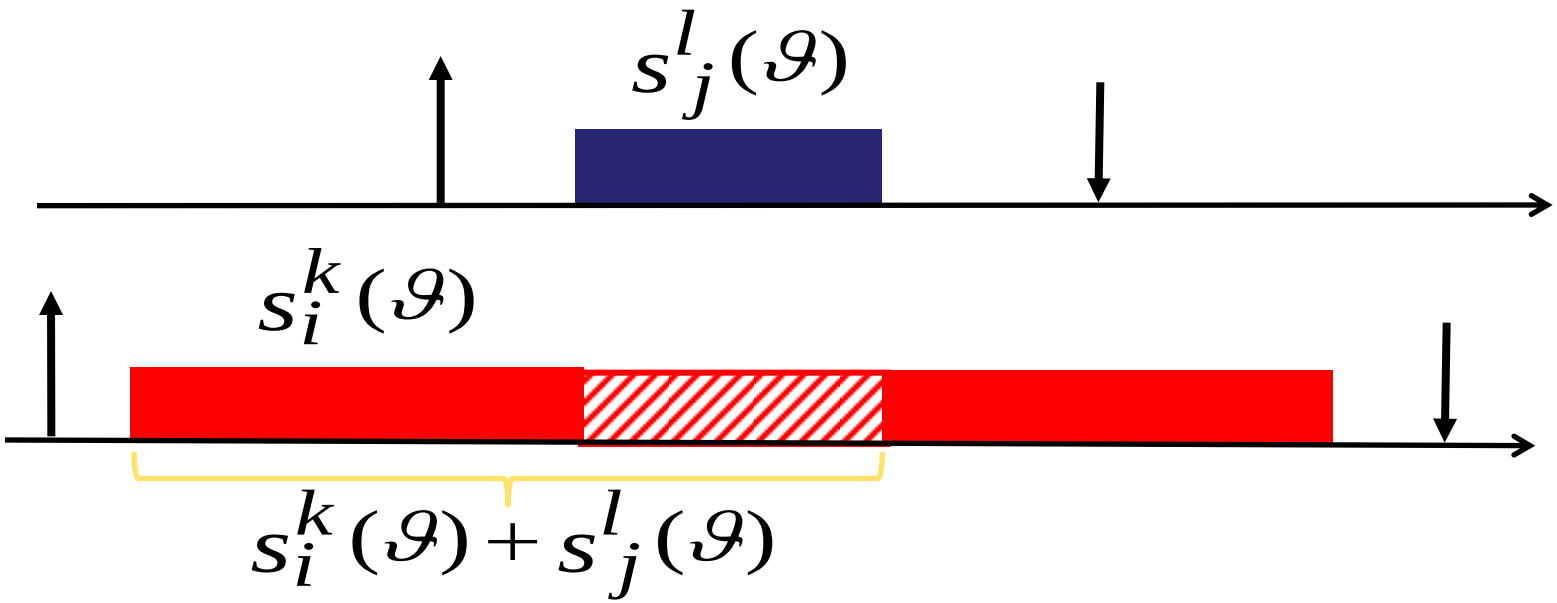
Model is traditional real-time model

- Threads arrive periodically, with deadlines equal to periods



- Threads subsume transactions
- Execute on a CMP
- Two schedulers: global EDF (G-EDF) and global RMA (G-RMA)
- Total thread utilization demand satisfies G-EDF (G-RMA)'s schedulable utilization bound

Earliest Deadline CM



$$RC(T_i) \leq \sum_{\theta \in \theta_i} \left(\left(\sum_{\tau_j \in \gamma_i(\theta)} \left(\left\lceil \frac{T_i}{T_j} \right\rceil \sum_{\forall s_j^l(\theta)} len\left(s_j^l(\theta) + s_{max}(\theta)\right) \right) \right) - s_{max}(\theta) + s_{i_{max}}(\theta) \right)$$

- γ_i: tasks sharing objects θ with τ_i
- $s_j^l(\theta)$: l^{th} atomic section of τ_j
- $s_{max}(\theta)$: longest atomic section in all tasks
- $s_{i_{max}}(\theta)$: longest atomic section in τ_i accessing θ

Thread execution time is inflated with worst-case transactional retry cost

- Each interference may cause a retry
- C_{ji} is the inflated WCET of τ_j relative to τ_i:

$$c_{ji} = c_j - \left(\sum_{\theta \in (\theta_j \wedge \theta_i)} len\left(s_j(\theta)\right) \right) + RC(T_{ji})$$

  - C_j : WCET of τ_j without retries
  - θ: shared object between τ_j and τ_i
  - $len(s_j(\theta))$: length of all atomic sections in τ_j that accesses θ
  - $RC(\tau_{ji})$: retry cost of τ_j without considering τ_i

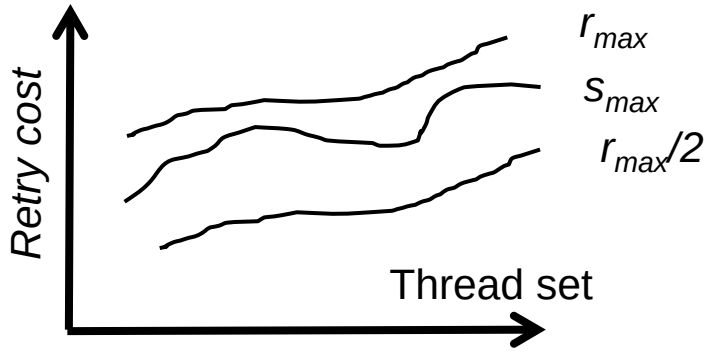
- G-EDF (G-RMA)'s response time analysis can now be used to determine schedulability of ECM

$$R_i^{up} = c_i + RC(T_i) + \left\lfloor \frac{1}{m} \sum_{j \neq i} W_{ij}(R_i^{up}) \right\rfloor$$

ECM can be formally compared with G-EDF/lock-free synchronization

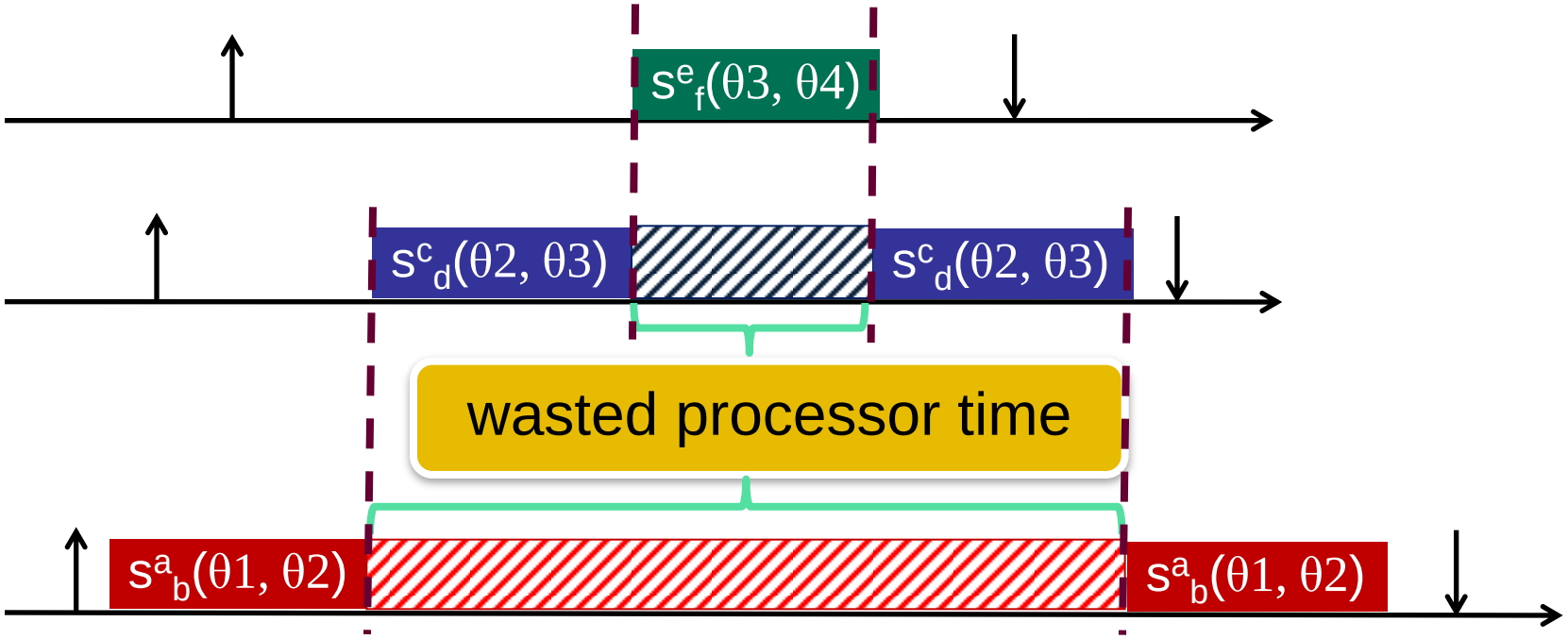
- Utilization demand-based schedulability (i.e., $U_i = C_i/T_i$)
  - Less demand is better
- ECM is better than G-EDF/lock-free if $s_{max} < r_{max}/2$
  - s_{max} : max cost of one transactional retry
  - r_{max} : max cost of one lock-free retry
- With low # conflicting tasks, s_{max} approaches r_{max}

STM can tolerate higher retry costs than lock-free and still be competitive



Priority CM with Negative value and First access (PNF)

- ECM, RCM, and LCM suffer from transitive retry

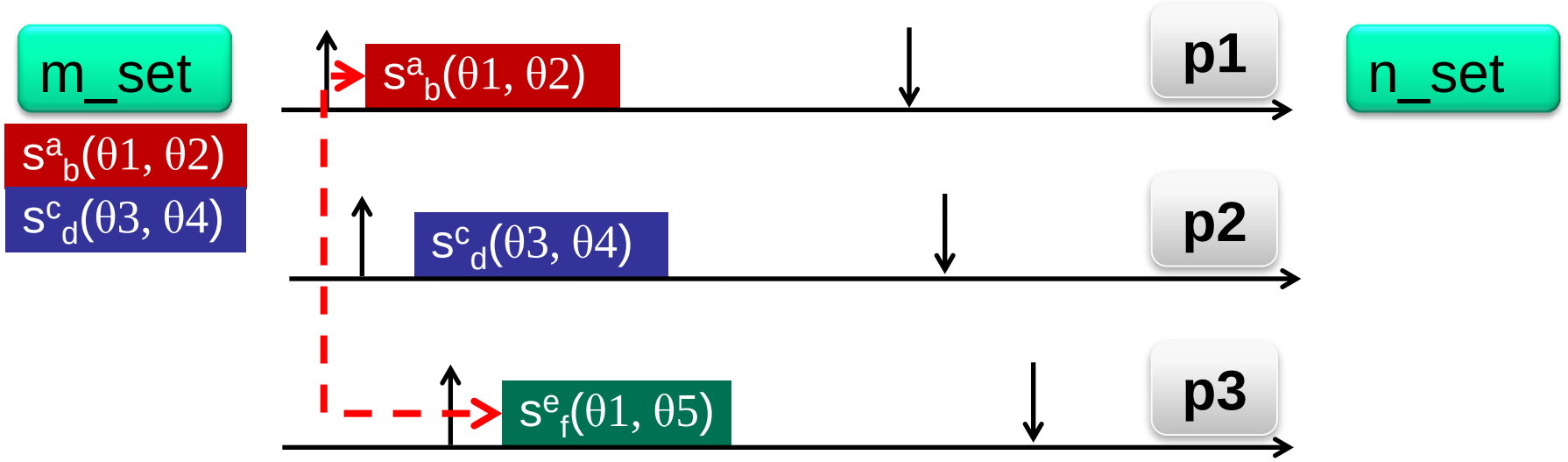


- Significantly wasted processor time for lower priority jobs

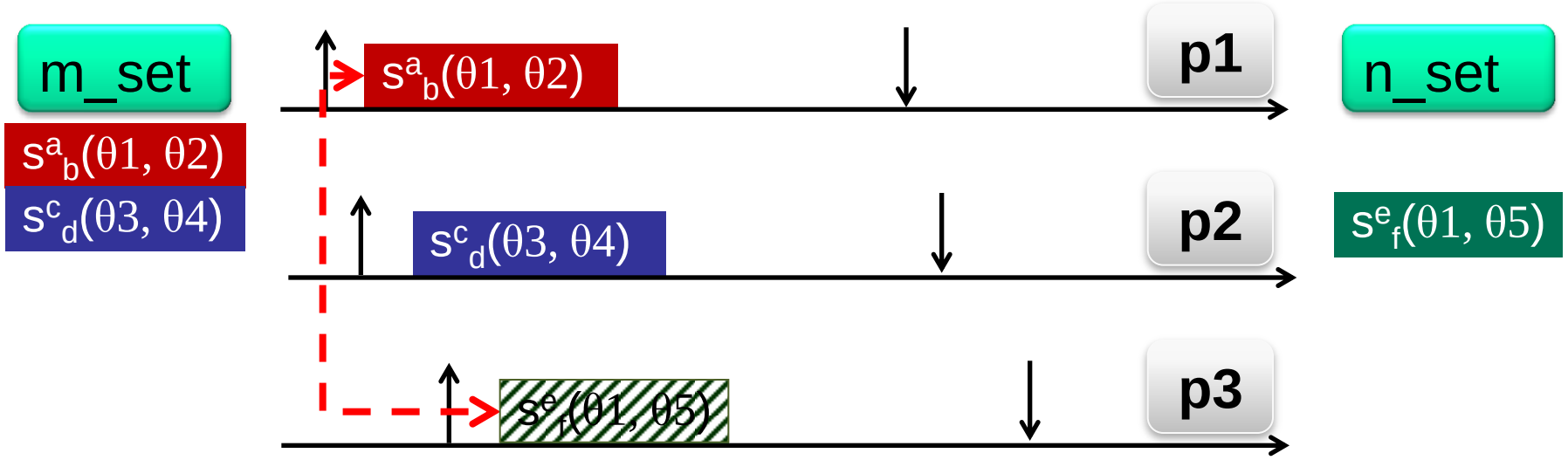
Design goals of PNF…

- Allow multiple objects per transaction
- Tight bound for transitive retry
- Reduce wasted processor time

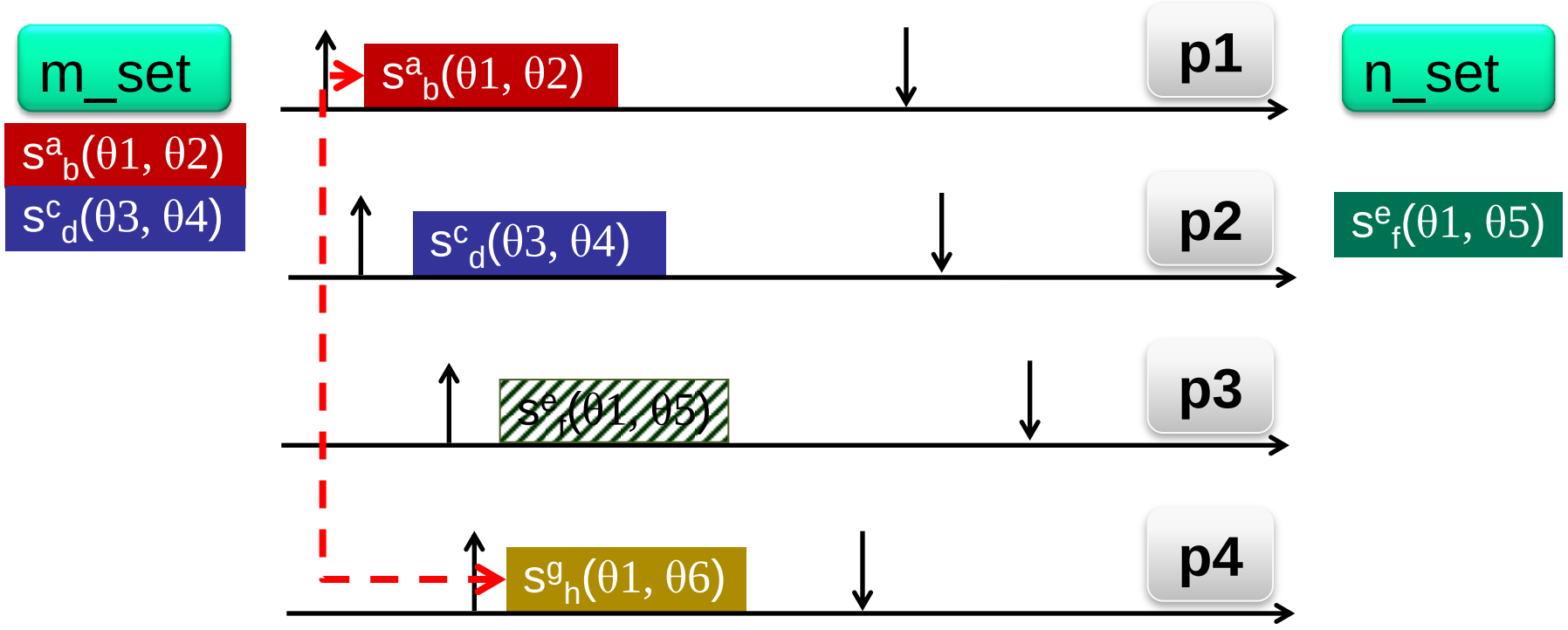
PNF design rationale



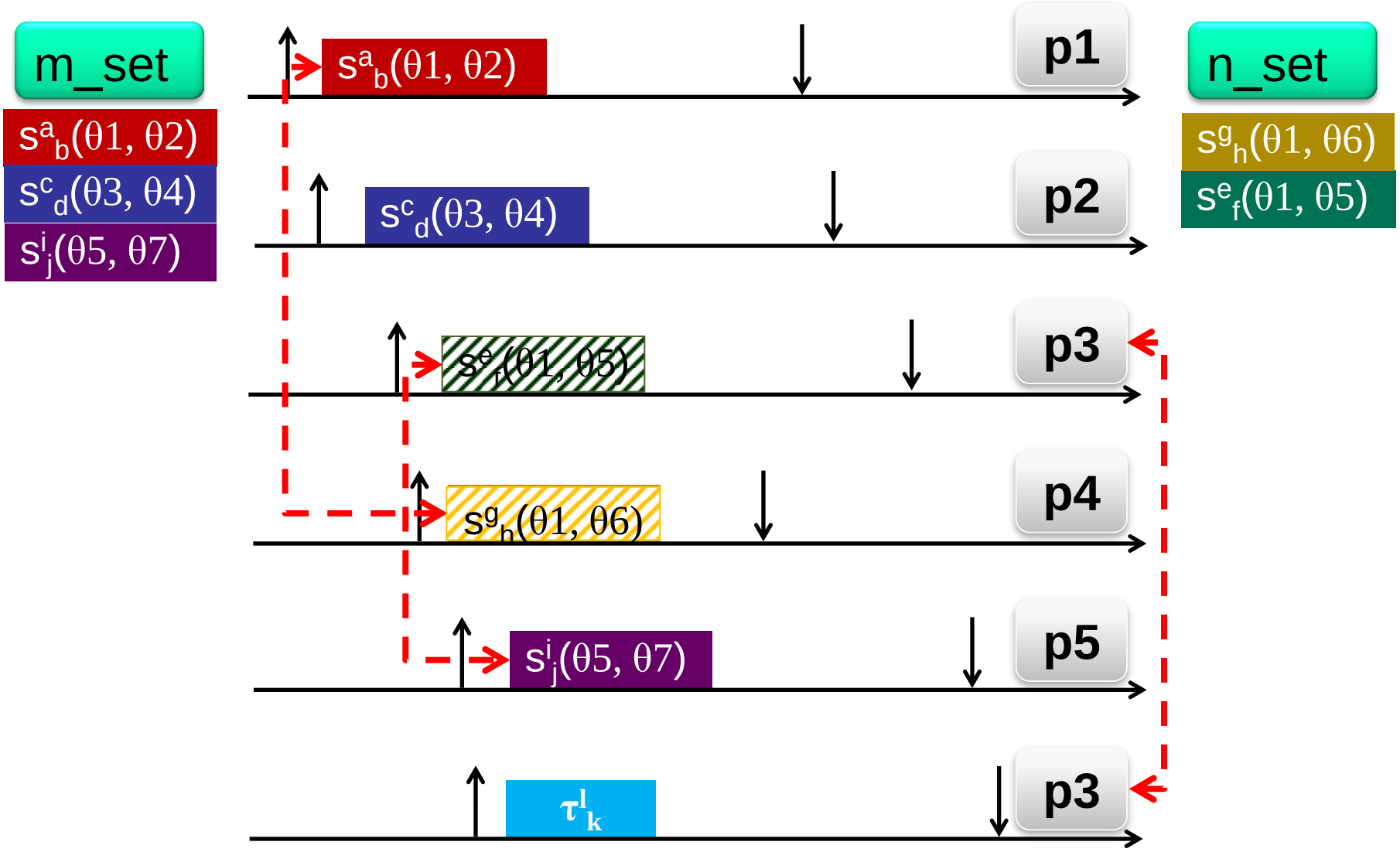
PNF rationale



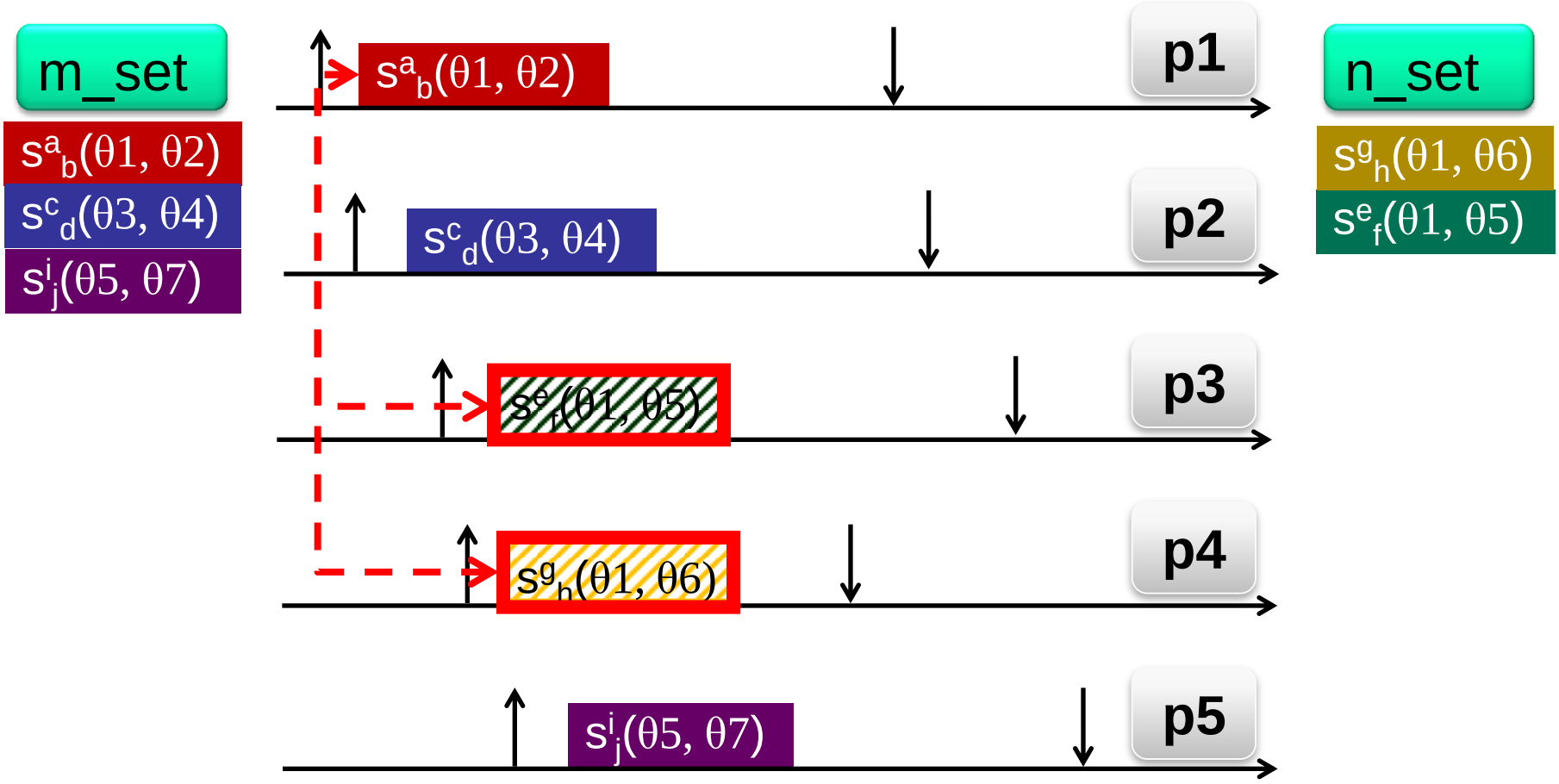
PNF rationale



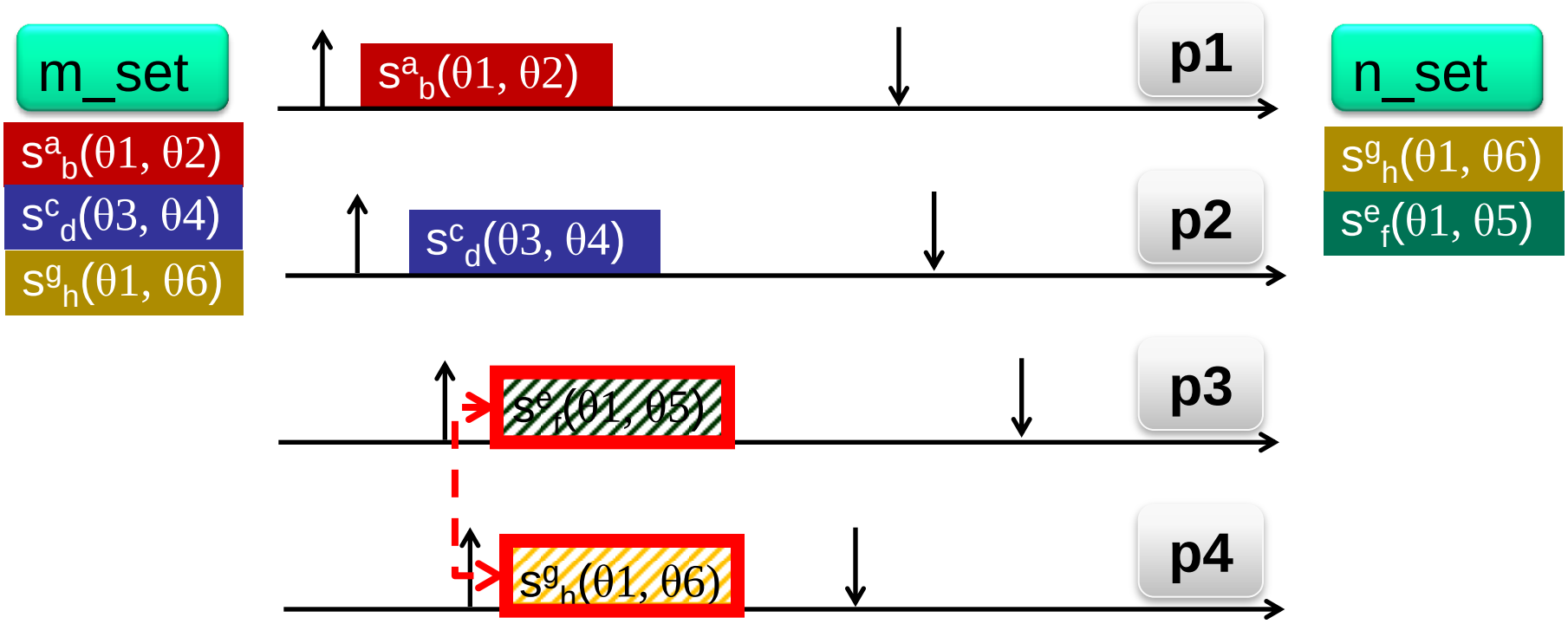
PNF rationale



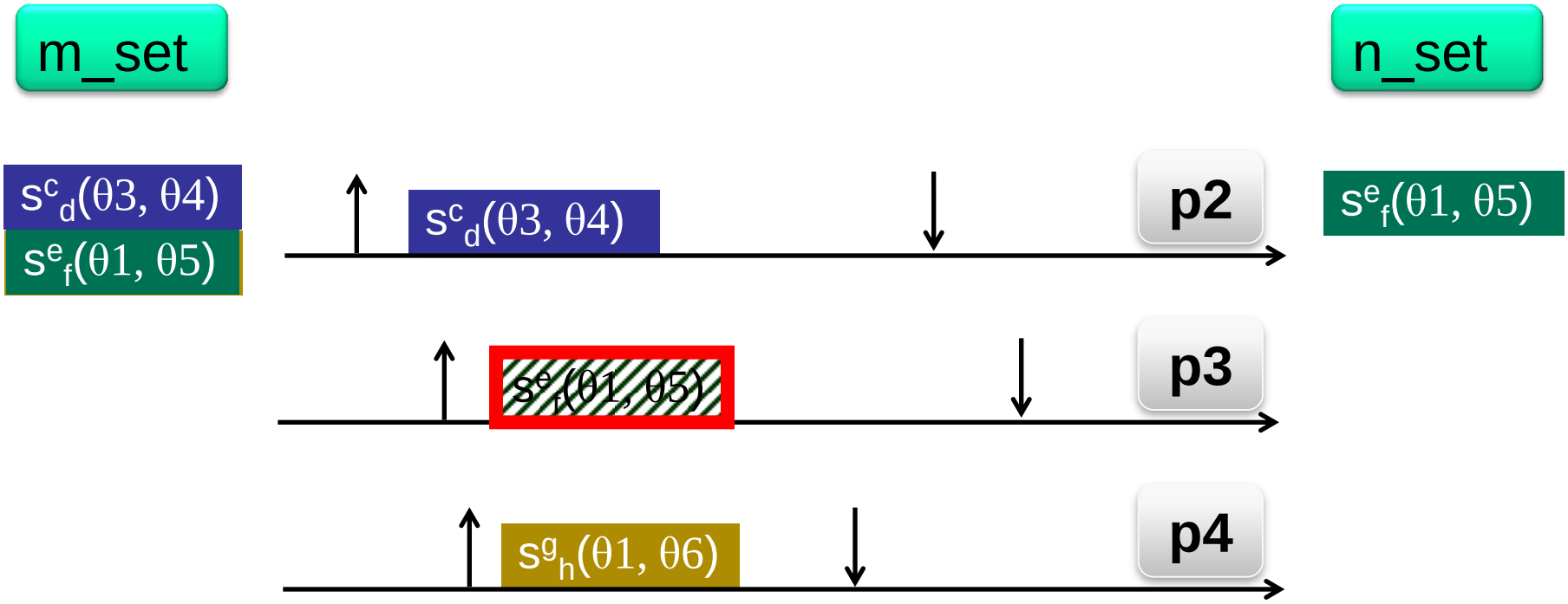
PNF rationale



PNF rationale



PNF rationale



PNF's retry cost and response time bounds

- Retry cost bound during interval *L*

$$RC(L) \leq \sum_{\tau_j \in \gamma_i} \left(\sum_{\theta \in \theta_i} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \bar{s_j^l}(\theta)} len\left(\bar{s_j^l}(\theta) \right) \right) \right)$$

- Blocking time bound due to lower priority jobs

$$D(\tau_i^x) \leq \left\lfloor \frac{1}{m} \sum_{\forall \bar{\tau_j^l}} \left(\left(\left\lceil \frac{L}{T_j} \right\rceil + 1 \right) \sum_{\forall \ddot{s_j^h}} len\left(\ddot{s_j^h} \right) \right) \right\rfloor$$

- Response time bound of τ_i^x

$$R_i^{up} \leq c_i + RC(L) + D(\tau_i^x) + \left\lfloor \frac{1}{m} \sum_{\forall j \neq i, p_j > pi} W_{ij}(R_i^{up}) \right\rfloor$$

Schedulability comparison

In the absence of transitive retry:

- PNF $\geq$ ECM when conflicting atomic sections have equal lengths
- PNF $\geq$ RCM when a large number of tasks heavily conflict

- PNF $\geq$ G-EDF/LCM's if the conflicting atomic section lengths are approximately equal and all α terms approach 1
- PNF $\geq$ G-RMA/LCM's if:
  - lower priority tasks suffer increasing number of conflicts from higher priority tasks
  - Lengths of the atomic sections increase as task priorities increase

- PNF $\geq$ lock-free if $s_{max}/r_{max} \leq 1$
  - Better than ECM! (and also other past CMs)

Implementation studies: experimental settings

- Implemented ECM, RCM, LCM, and PNF
- 3 task sets

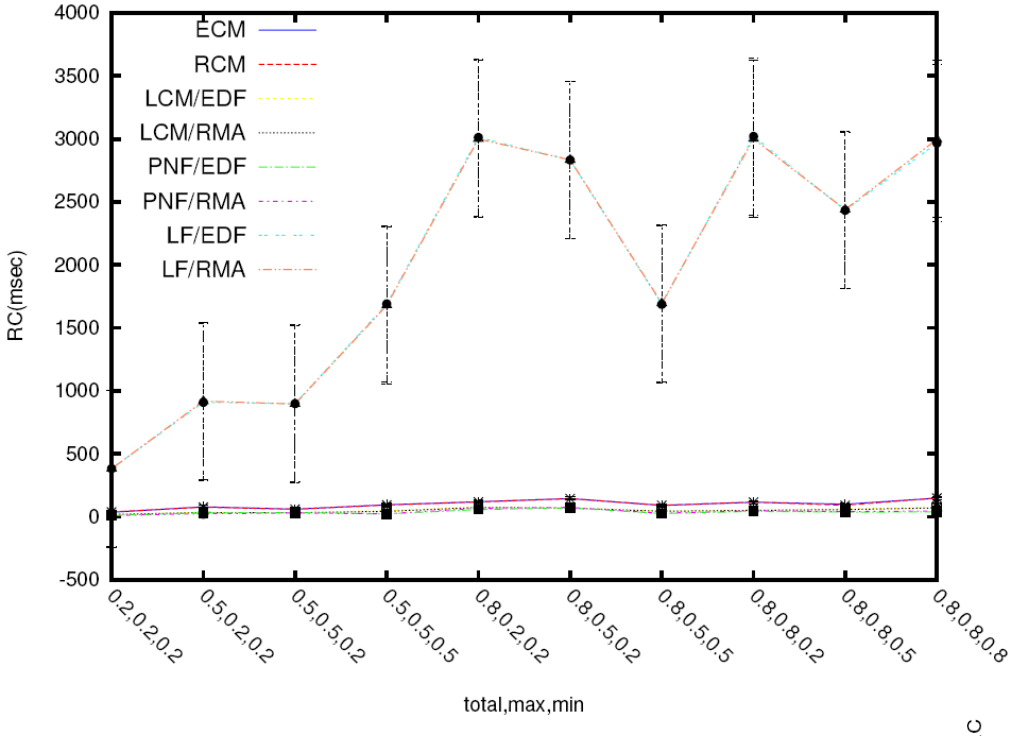
- ChronOS real-time Linux kernel (chronoslinux.org)
- 8-core, 2GHz AMD Opteron
- RSTM CAS for lock-free

$P_i(\mu s)$	$c_i(\mu s)$
1000000	227000
1500000	410000
3000000	299000
5000000	500000

$P_i(\mu s)$	$c_i(\mu s)$
1500000	961000
1875000	175000
2500000	205000
3000000	129000
3750000	117000
5000000	269000
7500000	118000
15000000	609000

$P_i(\mu s)$	$c_i(\mu s)$
375000	9000
400000	8000
500000	8000
600000	14000
625000	375000
750000	19000
1000000	26000
1200000	17000
1250000	21000
1500000	33000
1875000	39000
2000000	43000
2500000	18000
3000000	90000
3750000	28000
5000000	126000
7500000	231000
10000000	407000
15000000	261000
30000000	369000
375000	8000
30000000	407000

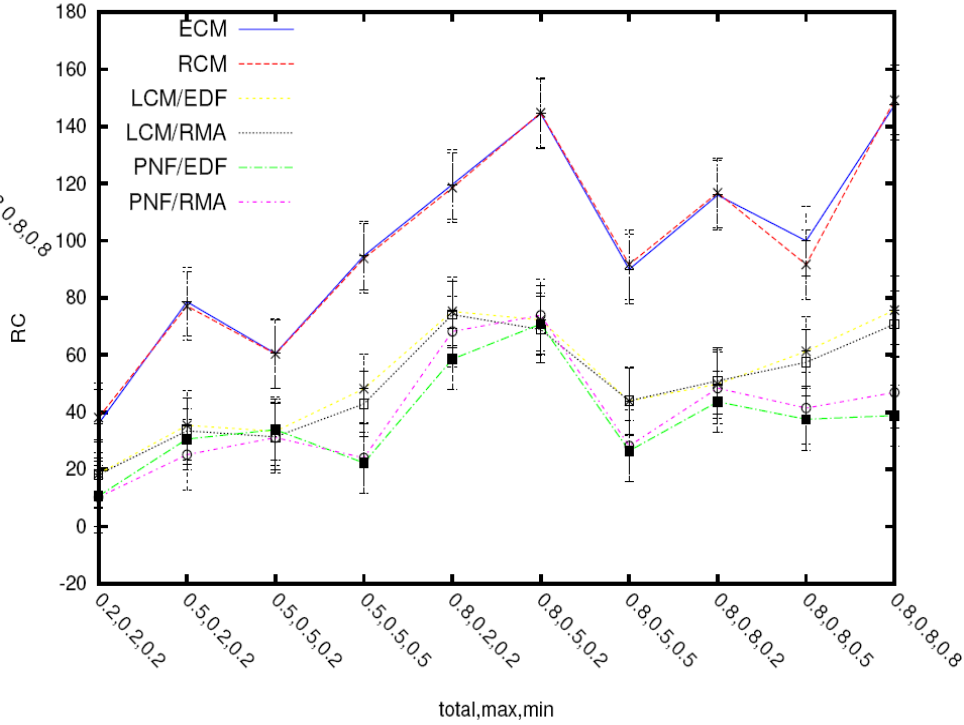
Implementation studies: retry cost under single shared object



4 tasks

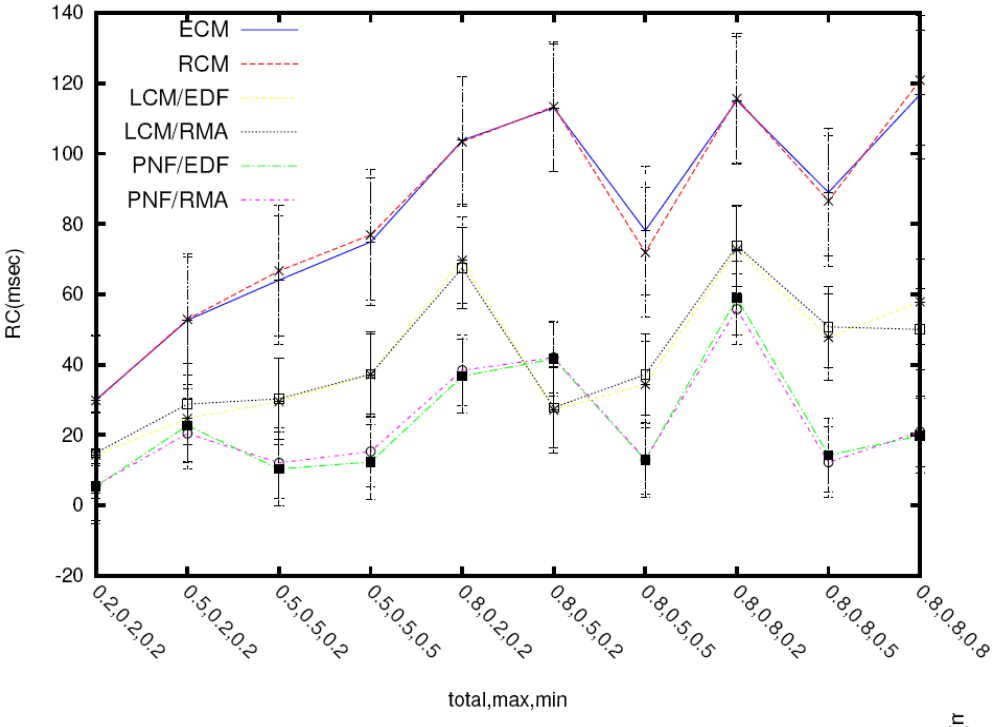
1 shared object per transaction

CMs and lock-free



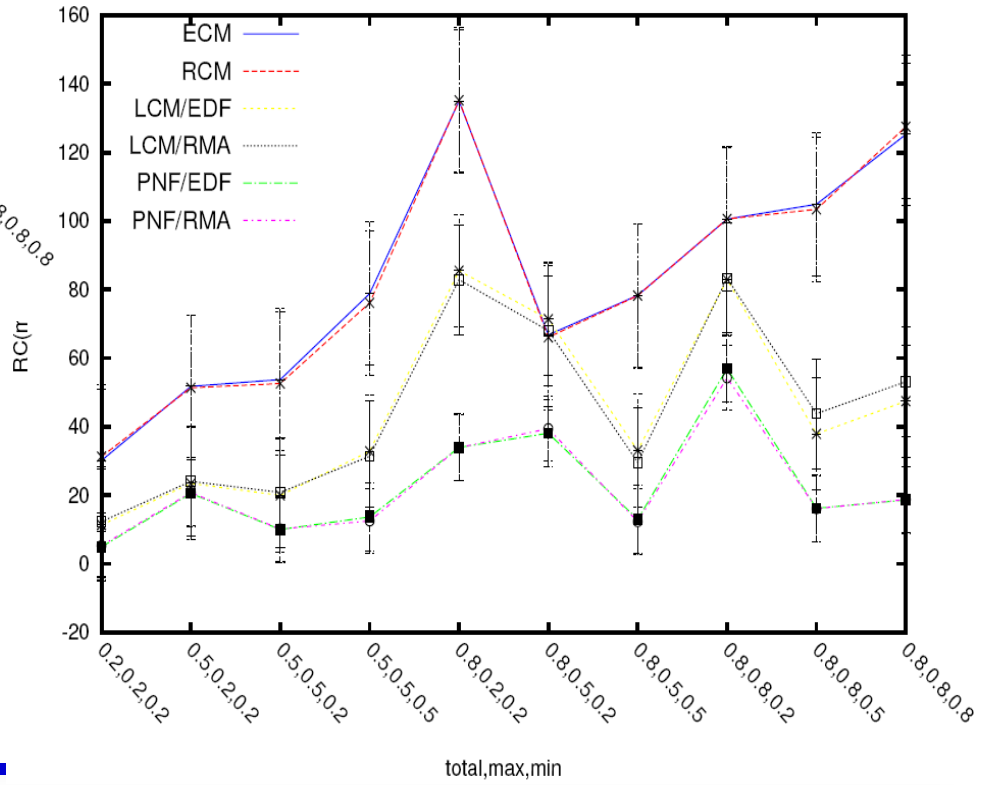
Retry cost measured by varying
max transaction length,
min transaction length, and
transactions: (total, max, min)

Implementation studies: retry cost under multiple shared objects



4 tasks
20 shared objects per transaction
40% write

4 tasks
40 shared objects per transaction
80% write



Conclusions

- Presented a real-time STM contention manager: PNF
  - Allows multiple objects per transaction; avoids transitive retry
  - Bounded retry costs and response times
- Schedulability comparisons established PNF's superiority
- Implementation confirmed PNF's superiority

- Allows reaping STM's programmability and composability advantages for a broader range than previously possible

- On the negative side:
  - Relatively complex implementation
  - Greater priority inversion
  - Must declare all transactional objects at transaction-start

FBLT: A Real-Time Contention Manager with Improved Real-Time Schedulability, M. El-Shambakey and B. Ravindran, *DATE 2013*