StackTrack

An Automated Transactional Approach to Concurrent Memory Reclamation

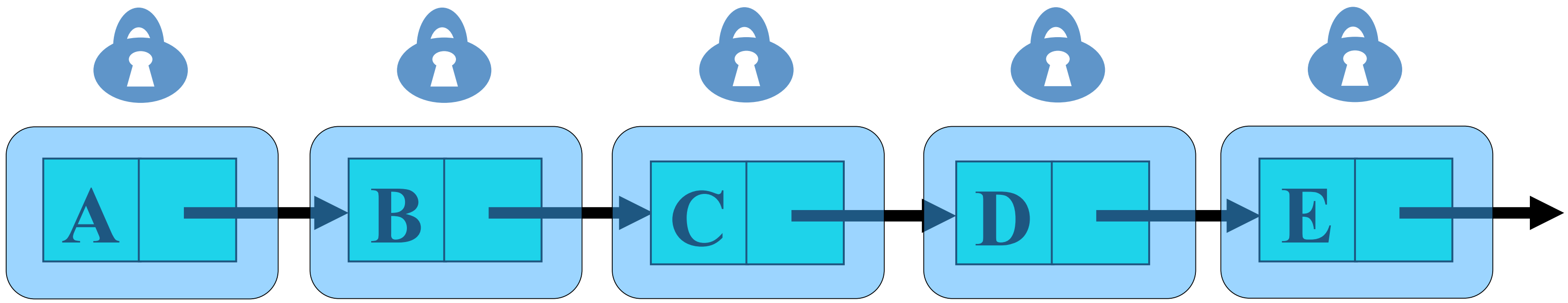
Dan Alistarh (MSR Cambridge)
Patrick Eugster (Purdue University)
Maurice Herlihy (Brown University)
Alexander Matveev (MIT)
Nir Shavit (MIT)

Concurrent Data Structures

- Memory Reclamation a big problem for efficient concurrent data-structures.
- Why?
  - To be efficient, operations must be designed in a certain way.
  - Let’s see an example

Concurrent List – First Try

- Consider a hand-over-hand locking design:

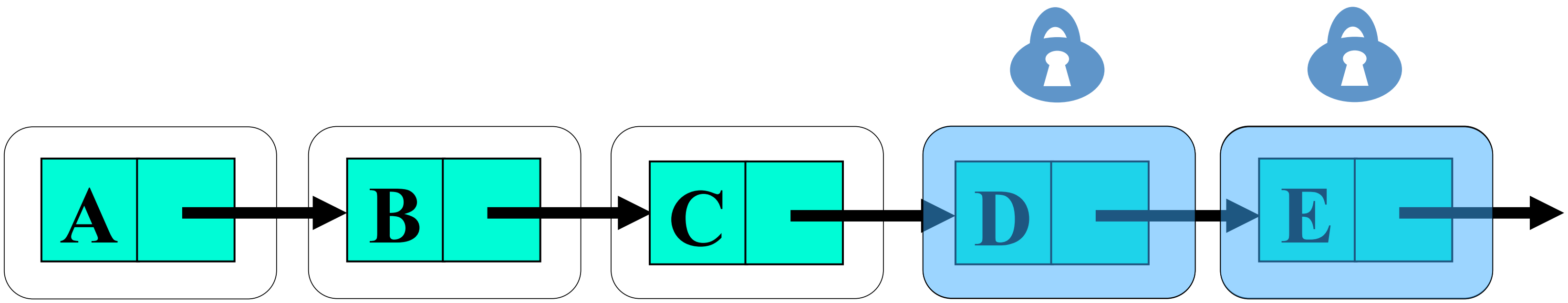


Very Inefficient

A synchronization operation for every node visited!

Concurrent List – Second Try

- Consider an optimistic design:



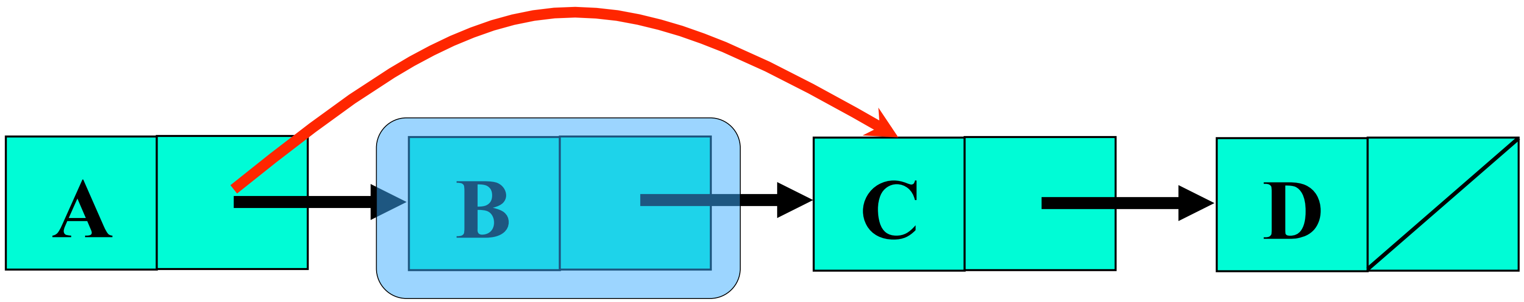
<u>Efficient</u>

A synchronization operation only for target nodes

Concurrent Data Structures

- Efficient concurrent data-structures, no matter if they use locks or not:
  - To be efficient, must avoid synchronizing while traversing
  - Like sequential algs: only read while traversing
  - But, this makes memory reclamation problematic
- Let's see an example

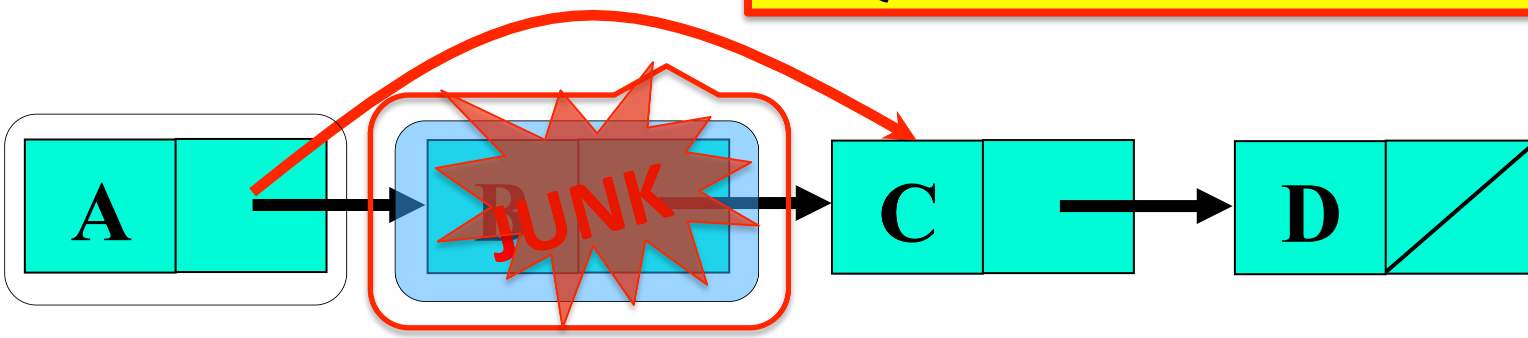
Memory Reclamation Problem




```
Thread P
b = a.next
a.next = c;

// b is disconnected
Free(b);
```

The Memory Reclamation Problem

The Problem:
P cannot detect Q, since Q's reads are invisible



Thread P
```
b = a.next
a.next = c;

// b is disconnected
Free(b);
```

Thread Q
```
b = a.next

// b is accessed
return b.Value + 2
```

SEGMENTATION FAULT

Memory Reclamation
Current Solutions

- **The problem:** We cannot free an object that has a reference to it by some thread.
- **The known solutions:** Actively track references of the threads to the memory objects.
  - Reads must be visible
  - But, we must have invisible reads to get good performance.

Memory Reclamation
Current Solutions

- **Existing Approaches:**
  1. Reference-counting

     [Detlefs et al., Gidenstam et al.]
  2. Quiescence-based

     [Harris, Hart et al.]
  3. Pointer-based

     [Michael, Herlihy et al.]

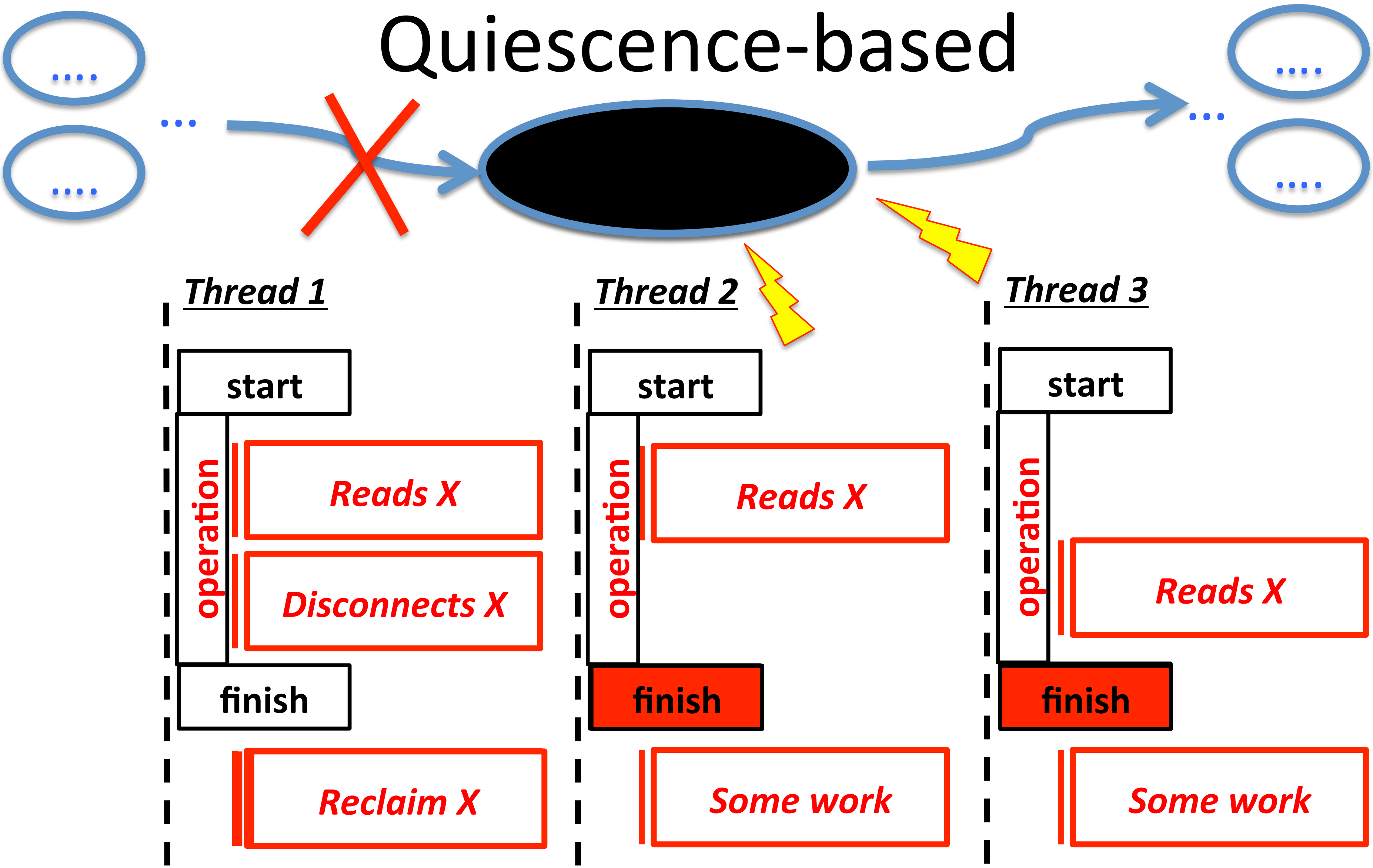
Reference-Counting

- <u>The idea:</u> Add a counter for every object that counts the number of references to it.
- Advantage:
  - Non-blocking
- Disadvantage:
  - Very inefficient
    - Every read must update a shared counter and do a memory fence

Quiescence-based

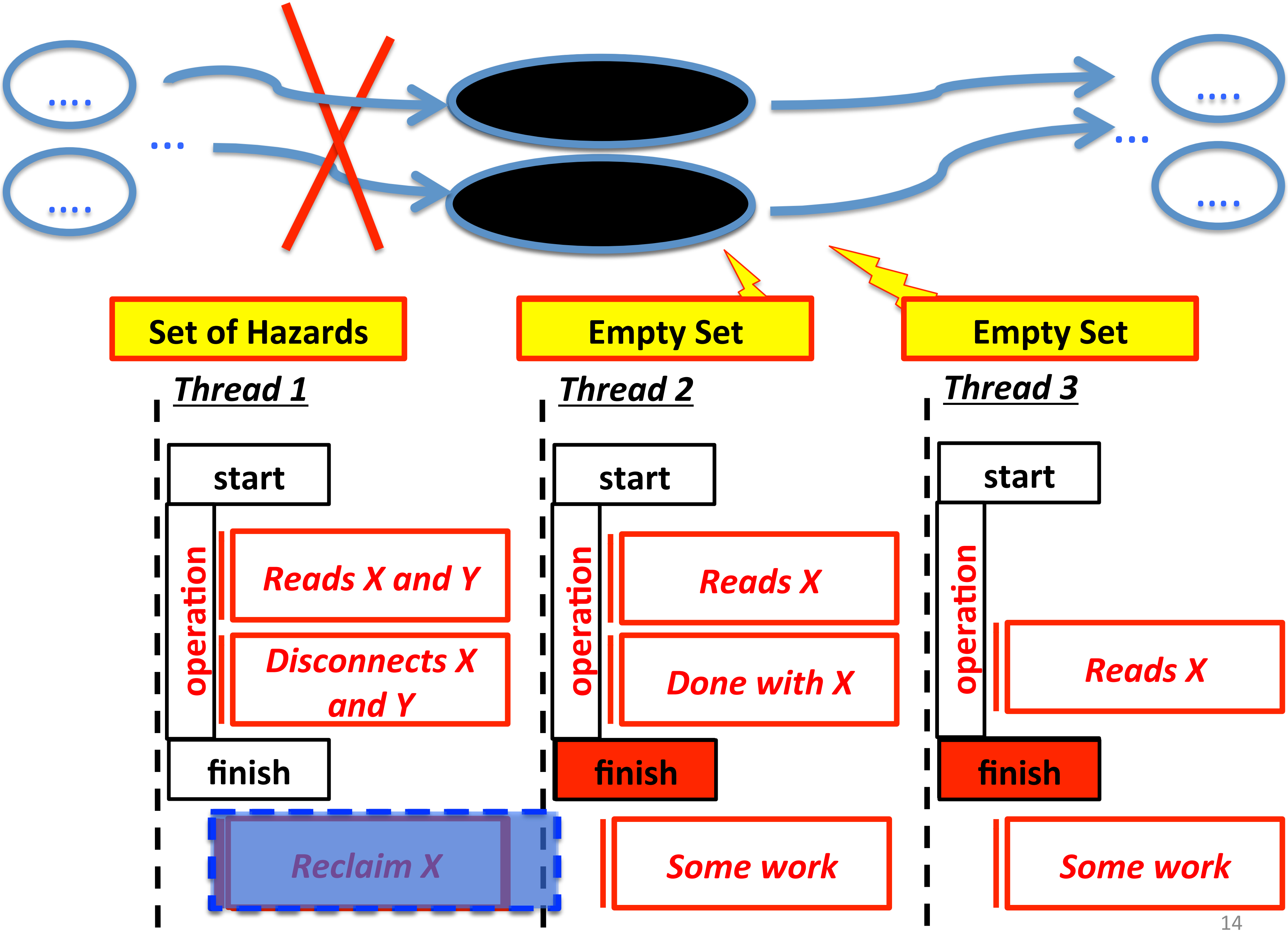
- The idea: track method calls.
- To reclaim, a thread waits for a quiescent state, in which all other threads finish their concurrent operation at least once.
- Advantage:
  - Efficient if threads are never delayed
- Disadvantage:
  - **Blocking:** If a thread blocks, unbounded amount of memory may be never freed.

Quiescence-based



Pointer-Based

- The idea: Track references by using special thread-local pointers. For example,
  - *Hazard Pointers* [Michael et al.]
  - *Pass-The-Buck* [Herlihy et al.]
  - *Drop-The-Anchor* [Braginsky et al.]
- Advantage:
  - Non-blocking
  - More efficient than reference counting.
- Disadvantage:
  - To be efficient, requires manual placement and verification of pointers.


....
...
....
....
...
....
Set of Hazards
Empty Set
Empty Set
Thread 1
Thread 2
Thread 3
start
operation
Reads X and Y
Disconnects X and Y
finish
Reclaim X
start
operation
Reads X
Done with X
finish
Some work
start
operation
Reads X
finish
Some work

Memory Reclamation
Current Solutions

•

Bad news for concurrent data-structures

- Very inefficient – sha[...] write for every read

2. Quiescence-based

Memory reclamation is too hard …

block.

3. Pointer-based

- Still not efficient enough: requires a memory fence per

No hope? …

Memory Reclamation

- Hardware Transactional Memory is a tool eliminating the need for locks
- Has been used to make reference counting faster [Dragojevic et al.].
- **New idea:** Use Hardware Transactional Memory (HTM) to track the references:
  - HTM is non blocking
  - HTM provides visible reads for free – no penalty

The StackTrack Algorithm

- **Main idea:** Use HTM to track thread local variables dynamically and atomically
  - No need to write the information about the references.
  - The reclaiming thread can simply scan the stacks of other threads (since they update atomically)

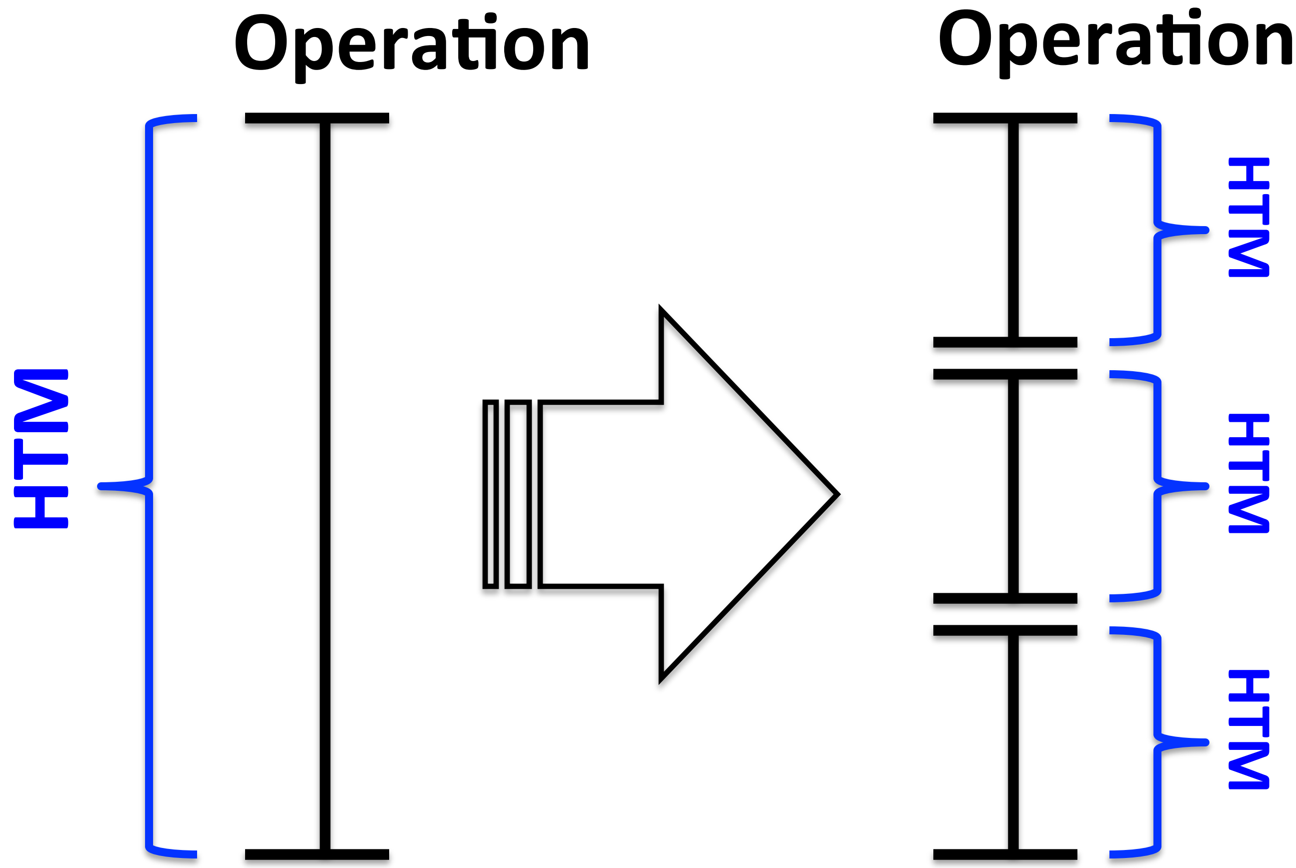
The StackTrack Algorithm

- Advantage:
  - Efficient and Automatic
- Disadvantage:
  - Reads must be transactional, so we depend on HTM performance.

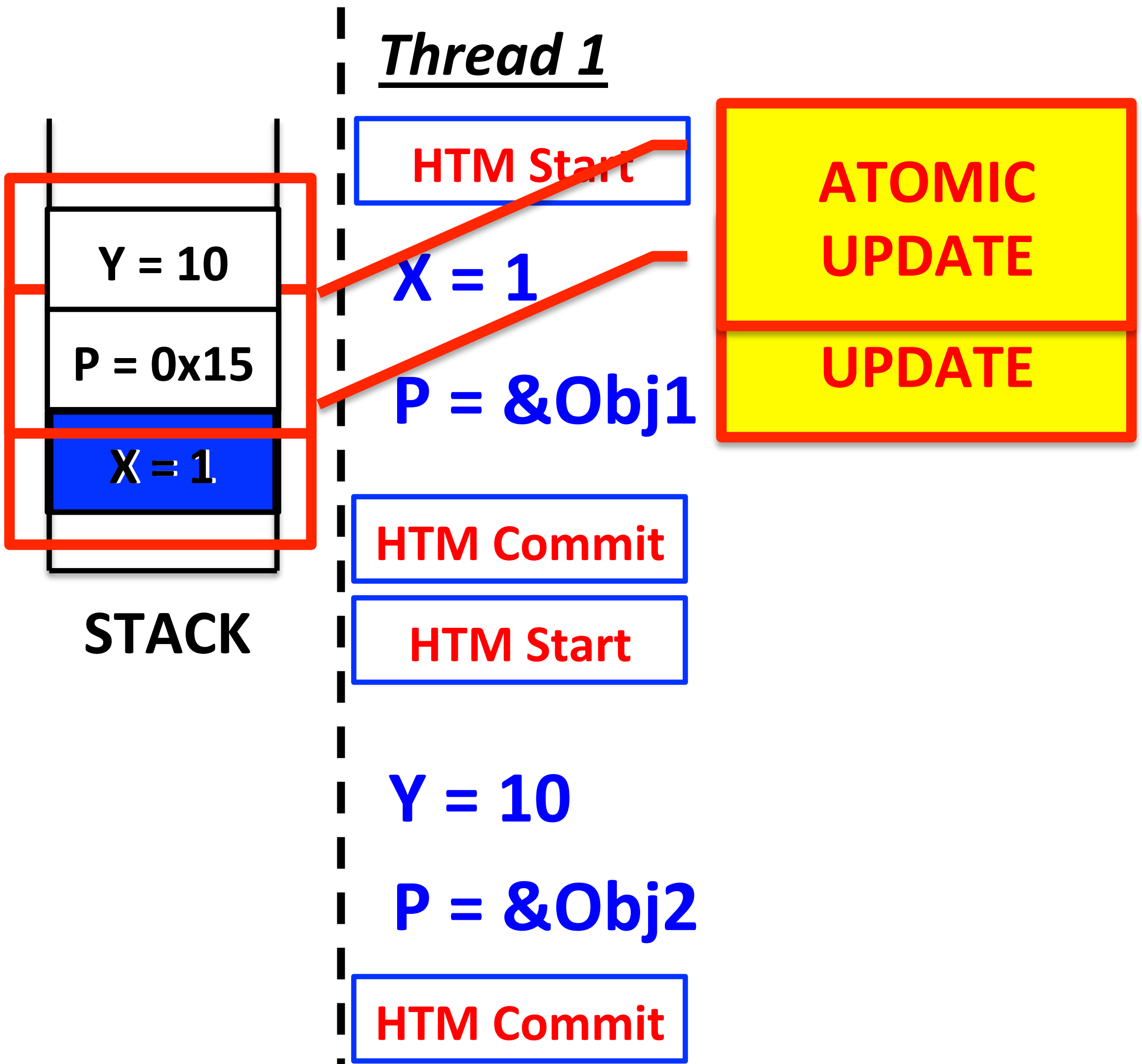
Adding HTM to the code

- **The problem:** How to apply HTM to the code?
- If we can execute a complete method call as one hardware transaction, then we are done.
- But, it is usually not possible, since HTM is limited in size.
- **Solution:** Split the operation into multiple hardware transactions.

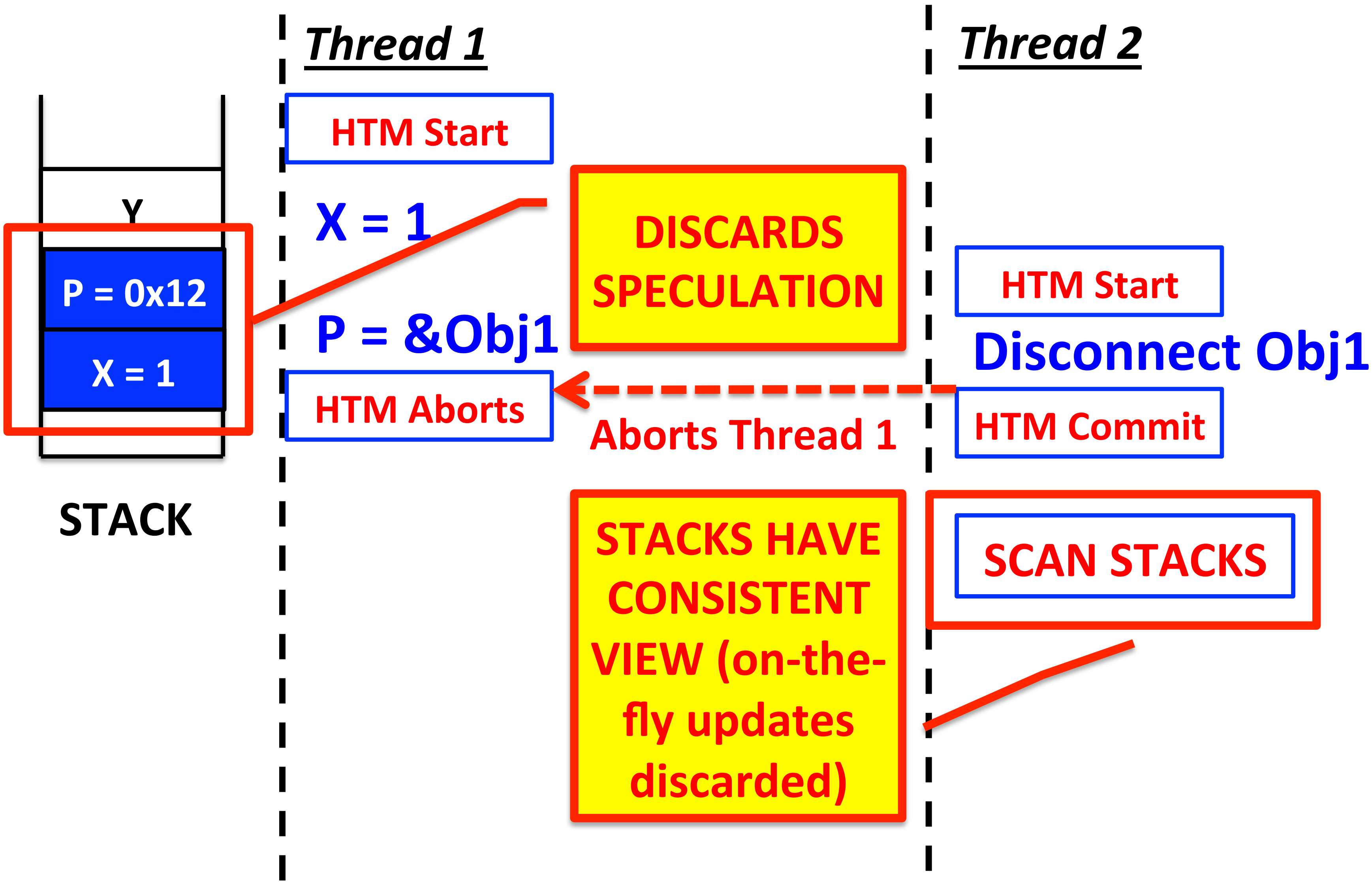
Splitting Transactions



Split HTM Execution (1)



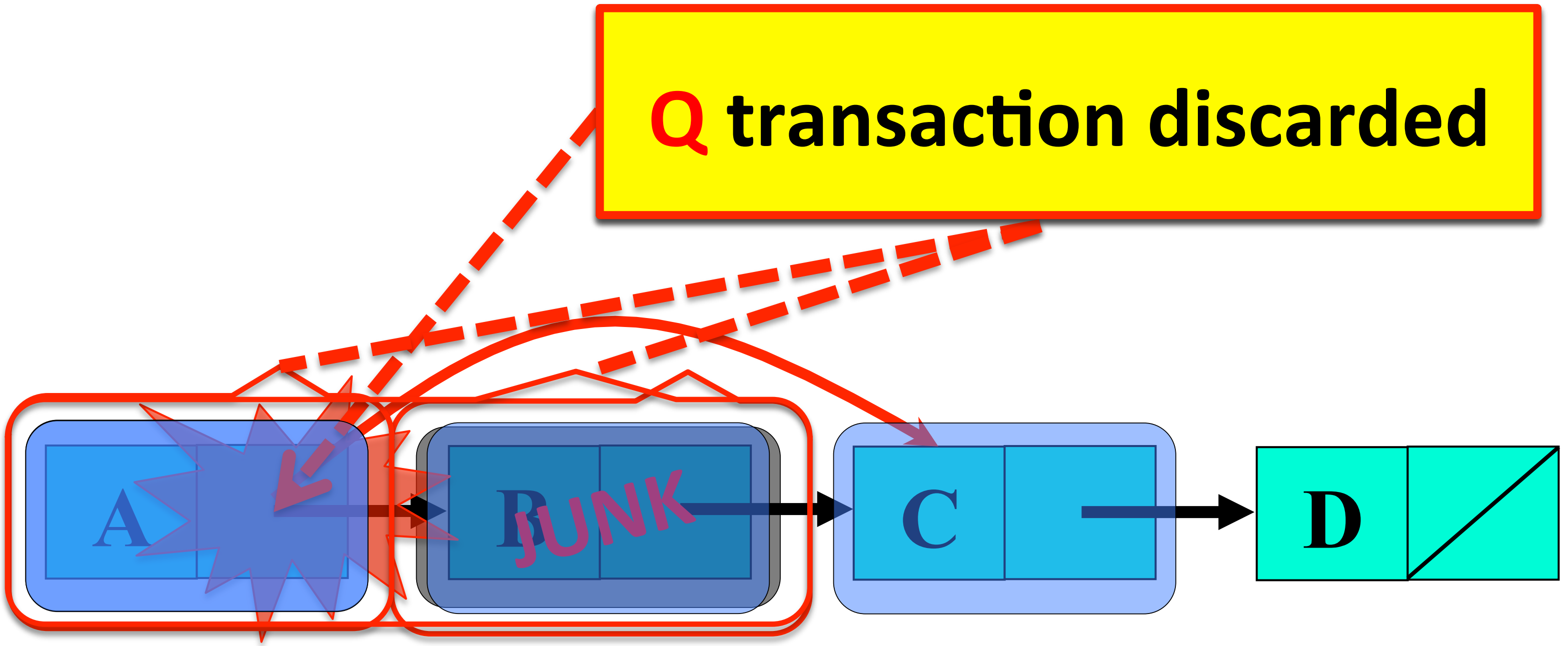
Split HTM Execution (2)



StackTrack

- All memory reclamation algorithms must coordinate the freeing of an object with concurrent reads of this object
- StackTrack avoids this!
- In StackTrack, concurrent reads of an object are speculative, and will abort when it is disconnected
- In StackTrack, freeing thread simply scans the stacks

Memory Reclamation Problem



Thread P

```
b = a.next
a.next = c;

// b is disconnected
Free(b);
```

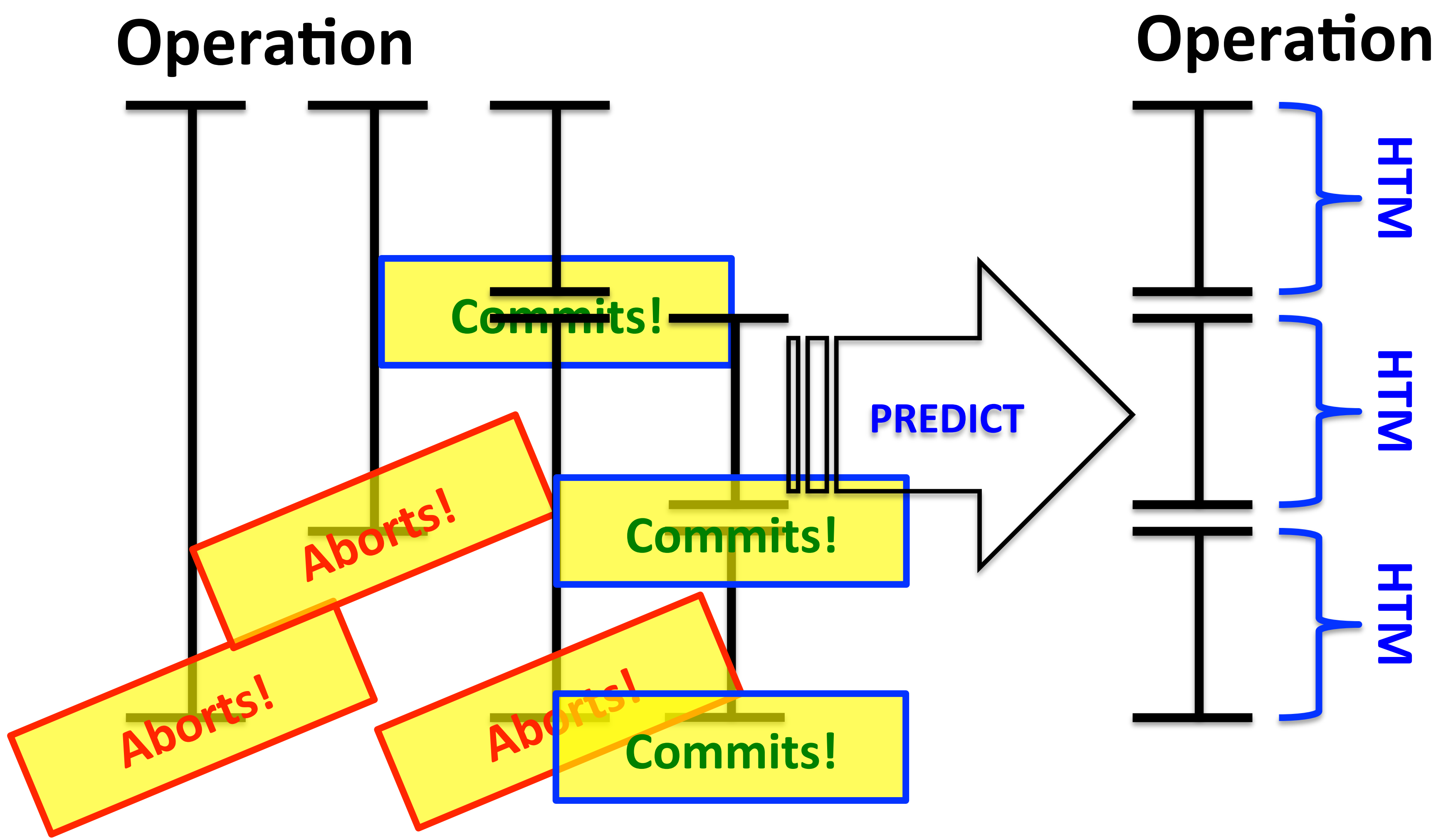
Thread Q

```
b = a.next

HTM restart

b = a.next
```

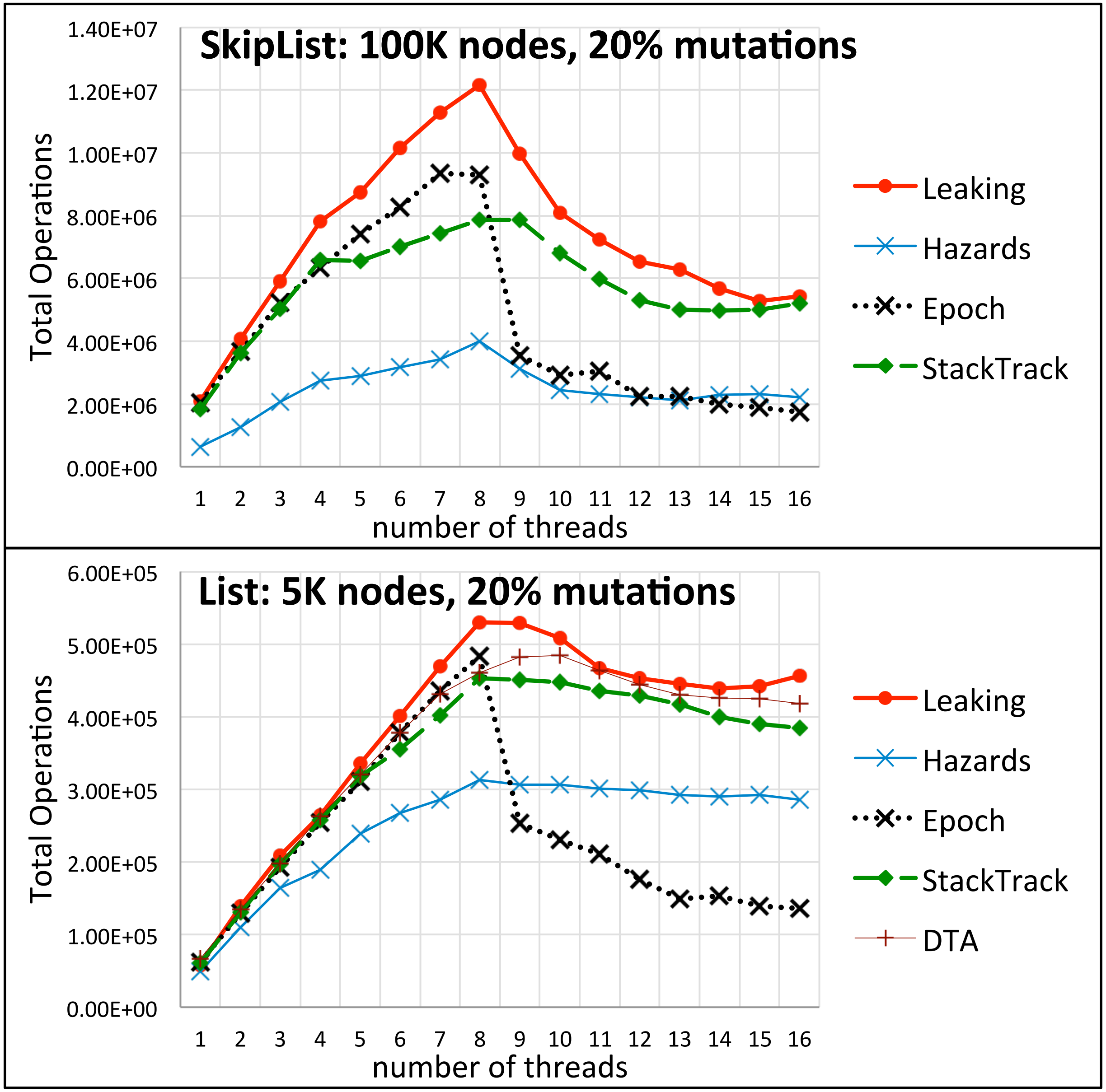
Automation of Splitting

- Do the splitting on the level of basic code blocks:
  - Inject a call to a split checkpoint function for every basic code block
  - The split checkpoint function counts the current number of blocks encountered
  - When its equal to the expected length, the HTM splits by executing an HTM commit and HTM start.

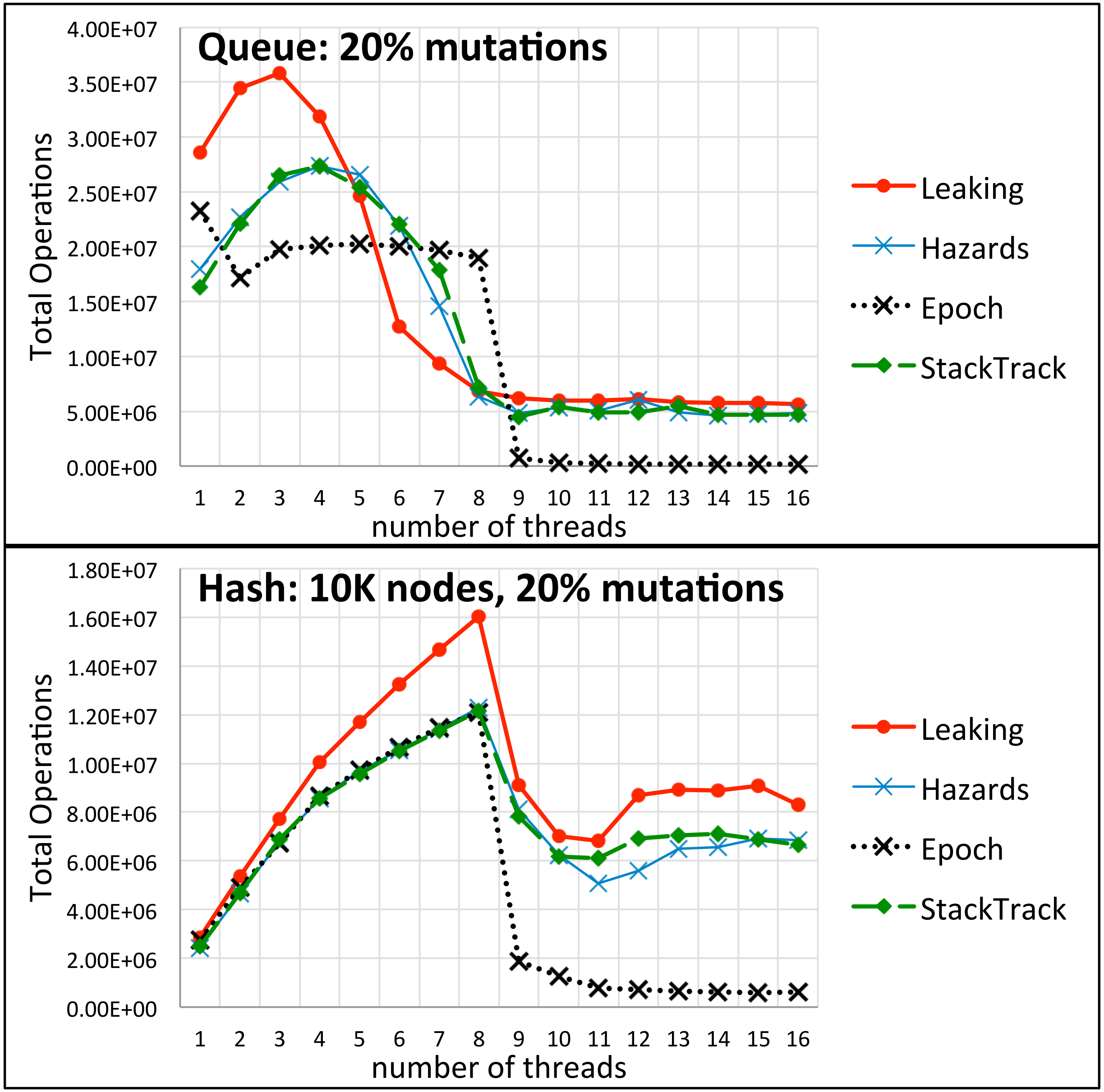
Splitting Transactions



Performance 1

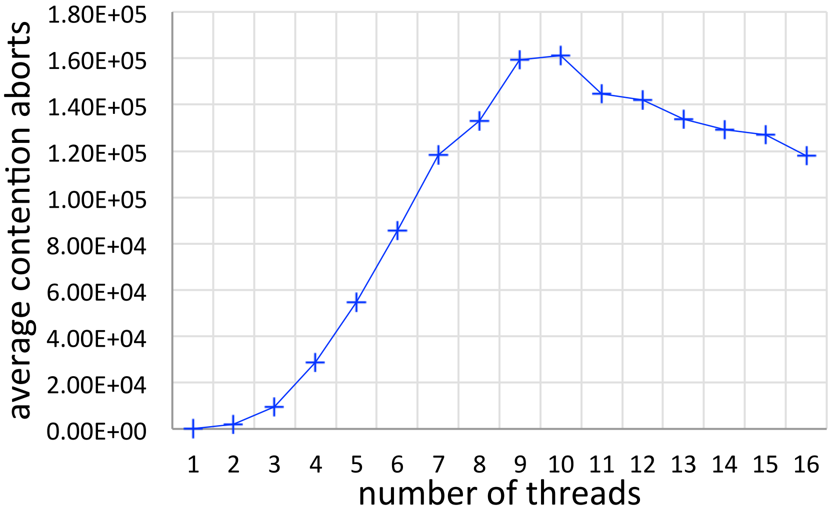


Performance 2

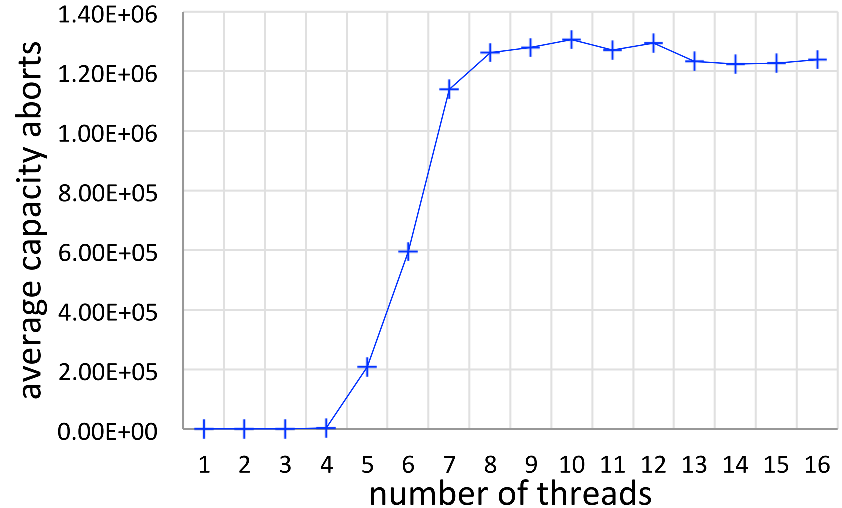


Performance Analysis

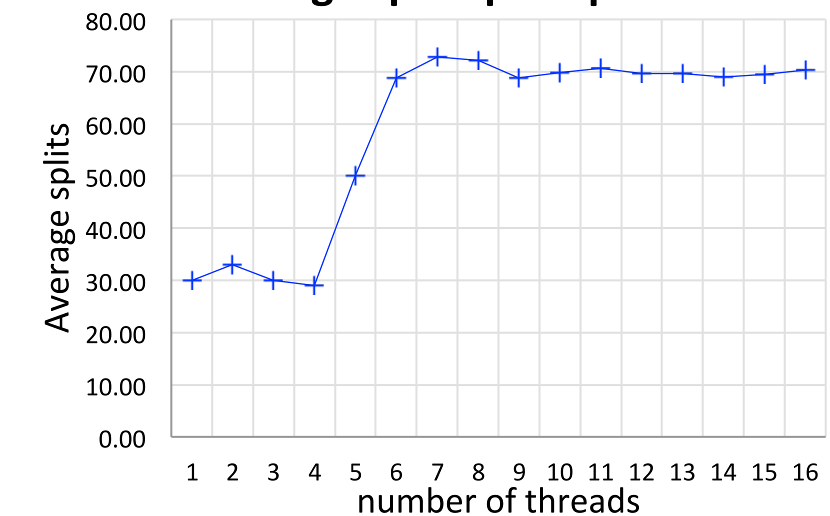
List: HTM average contention aborts



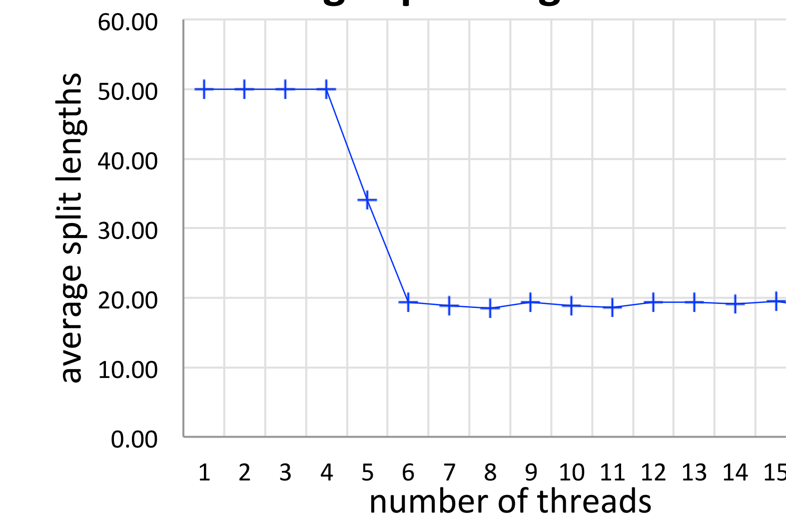
List: HTM average capacity aborts



List: HTM average splits per operation



List: HTM average split lengths



StackTrack

- A New Approach to Memory Reclamation
- Leverages HTM in a new way
- For the 1st time in concurrent data structure design, allows
  - efficient memory reclamation
  - without explicit programmer intervention

Thank You