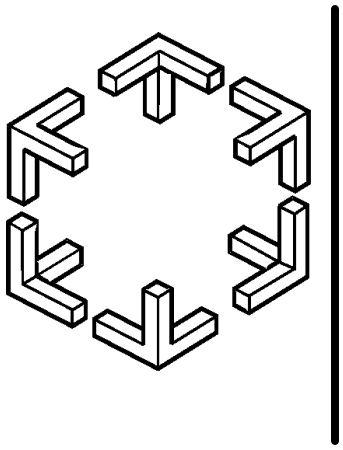
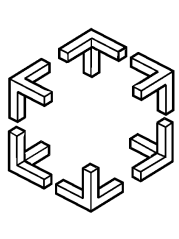


Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting

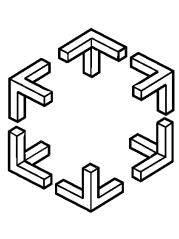
Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell and Philippas Tsigas

Distributed Computing and Systems group,
Department of Computer Science and Engineering,
Chalmers University of Technology



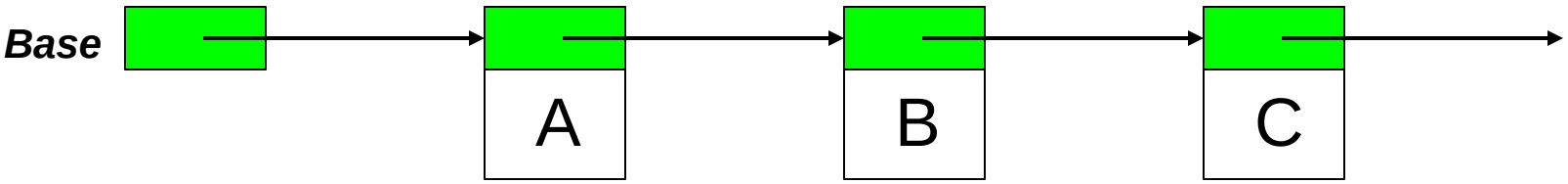
Outline

- Introduction
  - The Problem
  - Lock-free synchronization
- Our solution
  - Idea
  - Properties
- Experiments
- Conclusions

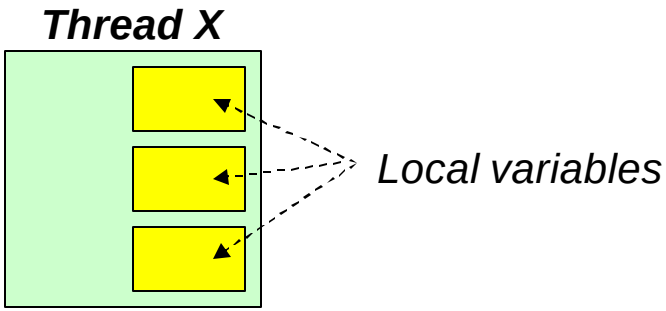


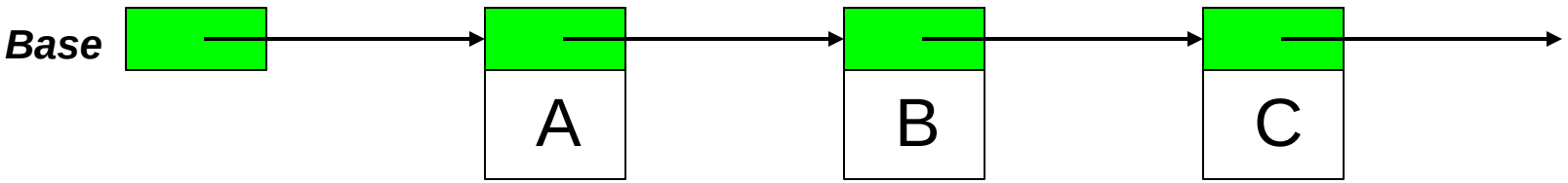
The Lock-Free Memory Reclamation Problem

- Concurrent shared data structure
  - Dynamic use of shared memory
  - Concurrent and overlapping operations by threads or processes

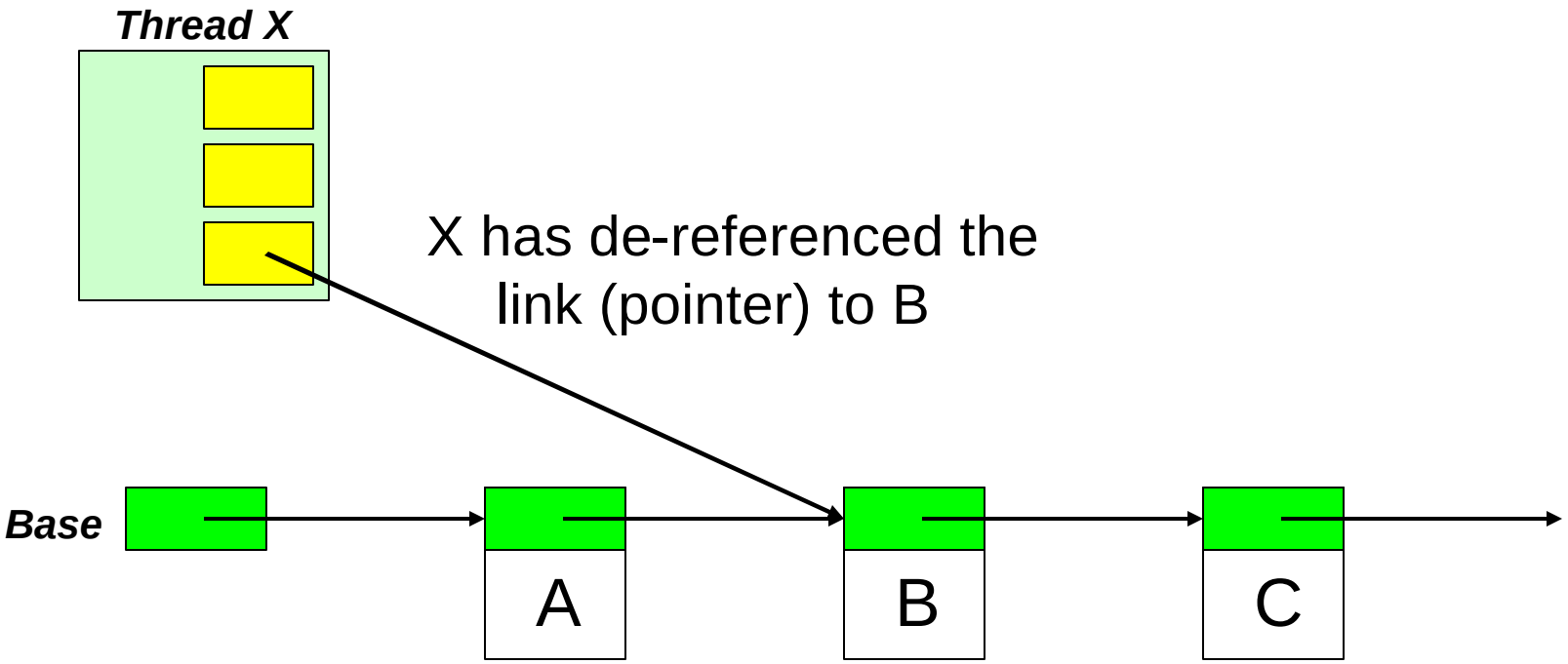


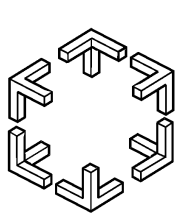
The Lock-Free Memory Reclamation Problem





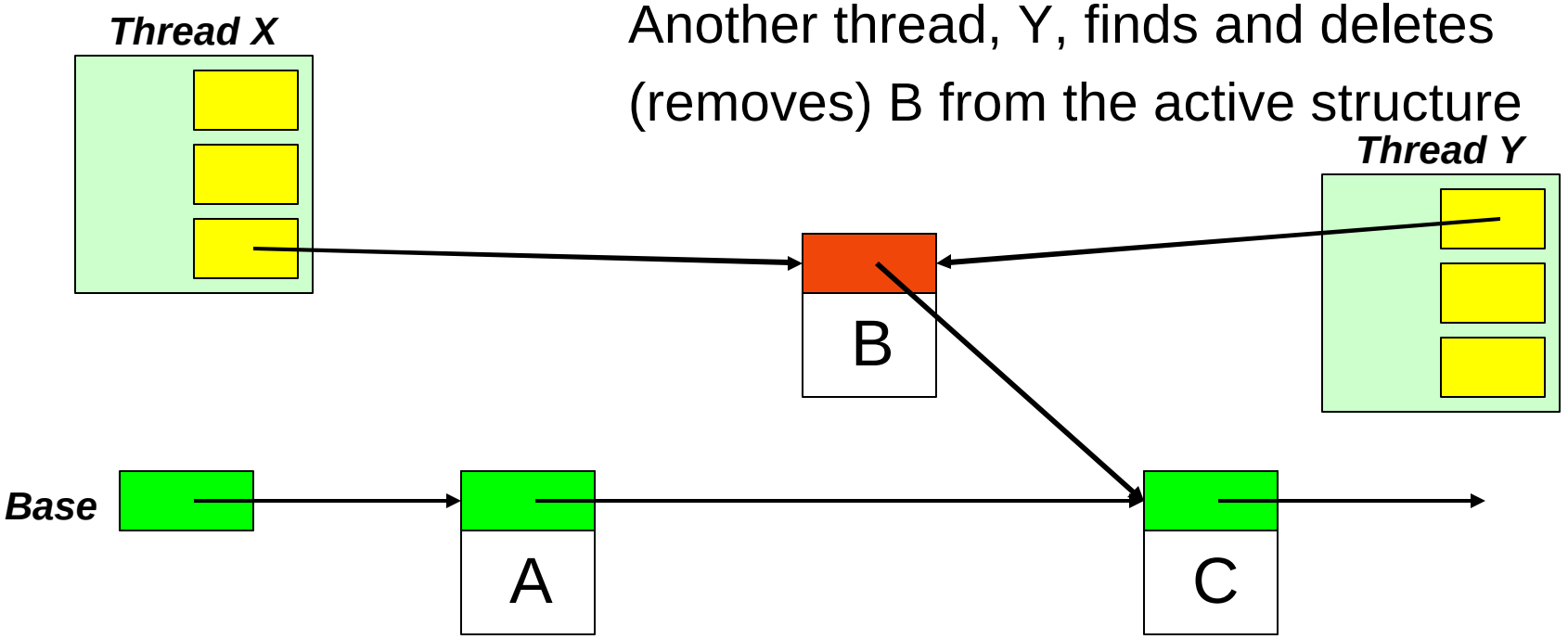
The Lock-Free Memory Reclamation Problem





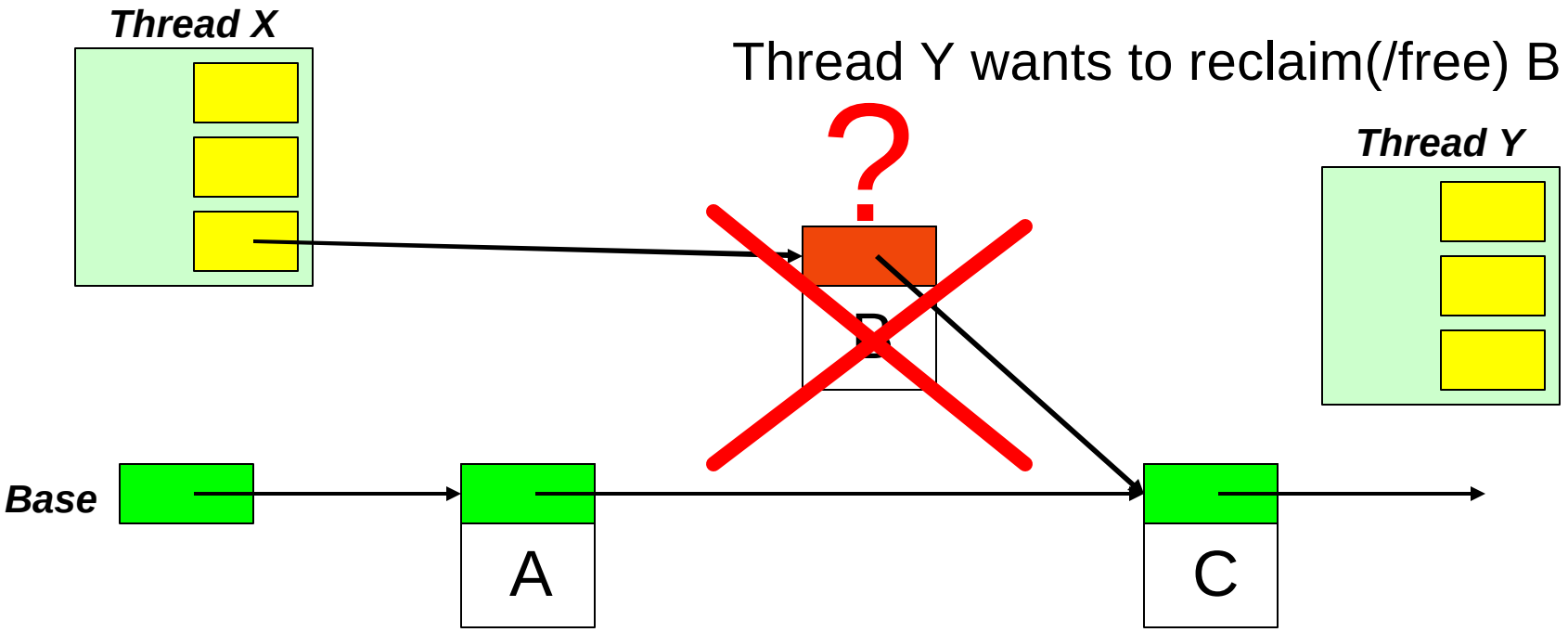
The Lock-Free Memory Reclamation Problem

Another thread, Y, finds and deletes (removes) B from the active structure



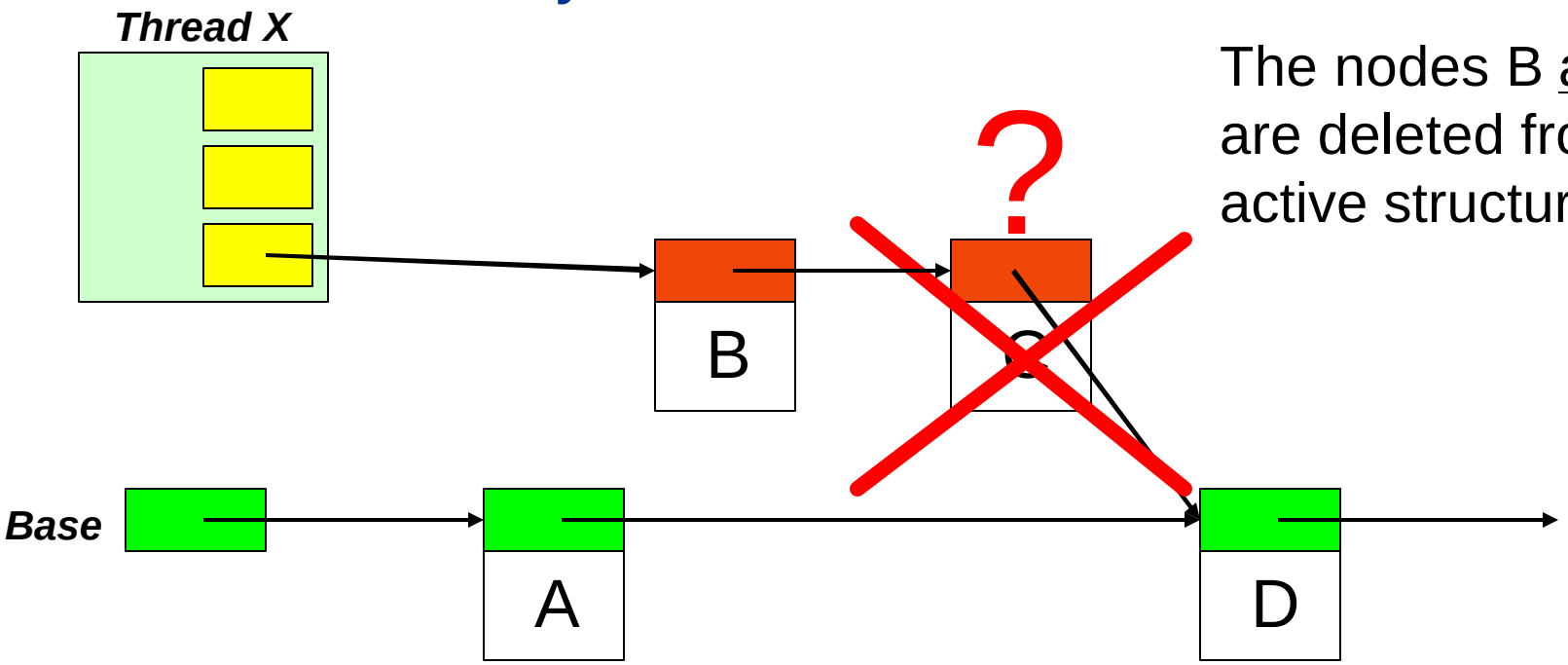
The Lock-Free Memory Reclamation Problem

Property I: A (de-)referenced node is not reclaimed

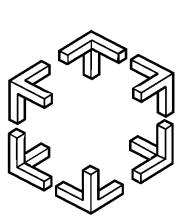


The Lock-Free Memory Reclamation Problem

Property II: Links in a (de-)referenced node should always be de-referencable.



The nodes B and C are deleted from the active structure.

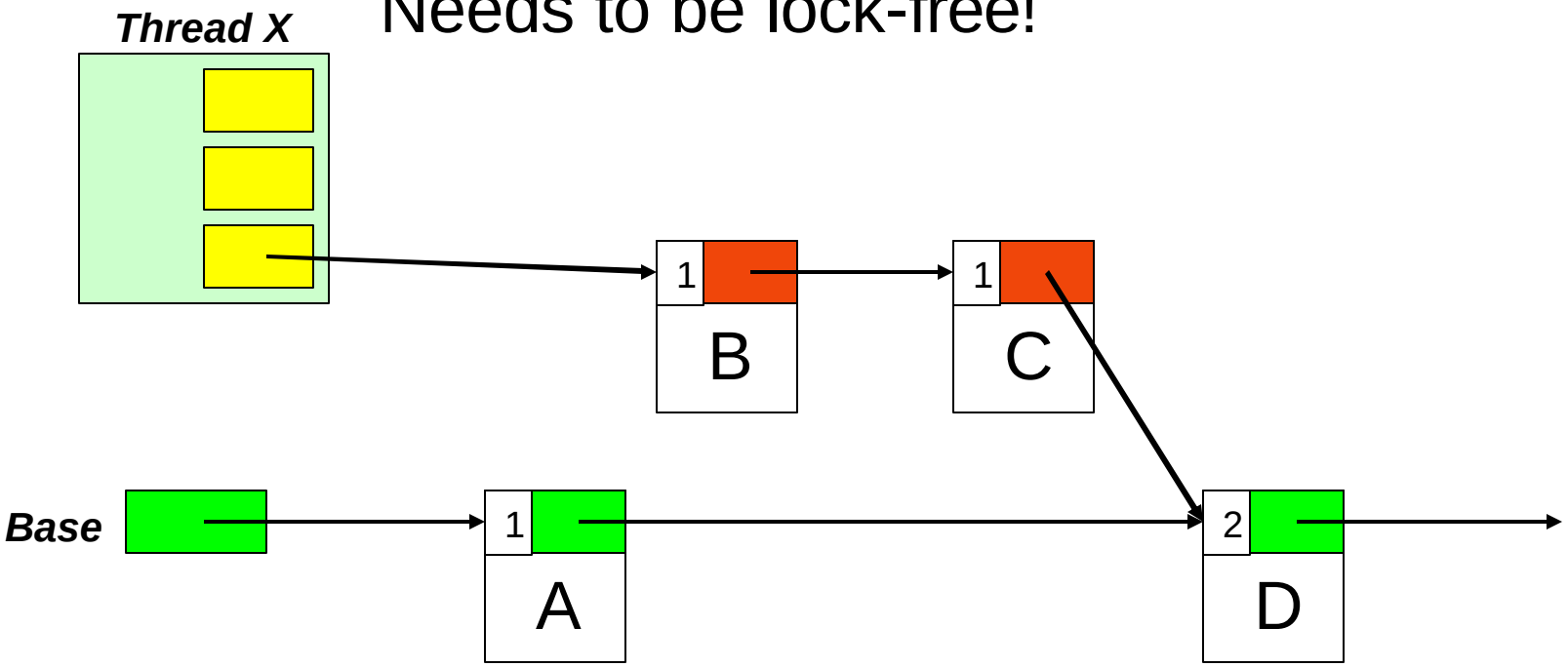


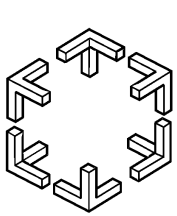
The Lock-Free Memory Reclamation Problem

Solutions?

- Garbage collection?
- Reference counting?

Needs to be lock-free!



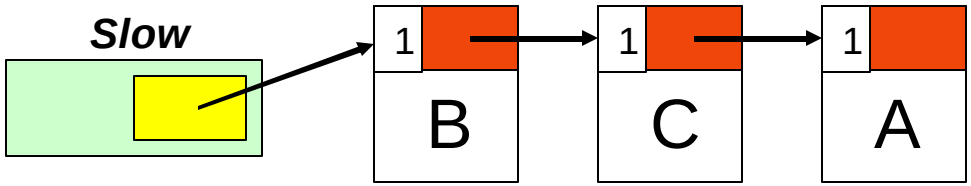


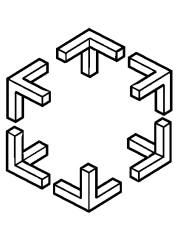
Lock-free synchronization

- A lock-free shared data structure
  - Allows concurrent operations without enforcing mutual exclusion (i.e. no locks)
  - Guarantees that at least one operation always makes progress
  - Avoids:
    - Blocking, deadlock and priority inversion
- Hardware synchronization primitives
  - Built into CPU and memory system
    - Typically: atomic read-modify-write instructions
  - Examples
    - Test-and-set, Compare-and-Swap, Load-Linked / Store-Conditional

Previous solutions

- Lock-free Reference Counting
  - Valois + Michael & Scott 1995
  - Detlefs et al. 2001
  - Herlihy et al. 2002
- Remaining issues
  - A slow thread might prevent reclamation
  - Cyclic garbage
  - Implementation practicality issues
    - Reference-count field MUST remain forever (Valois + Michael & Scott)
    - Needs double word CAS (Detlefs et al.)
    - Needs double width CAS (Herlihy, 2002)
    - Large overhead



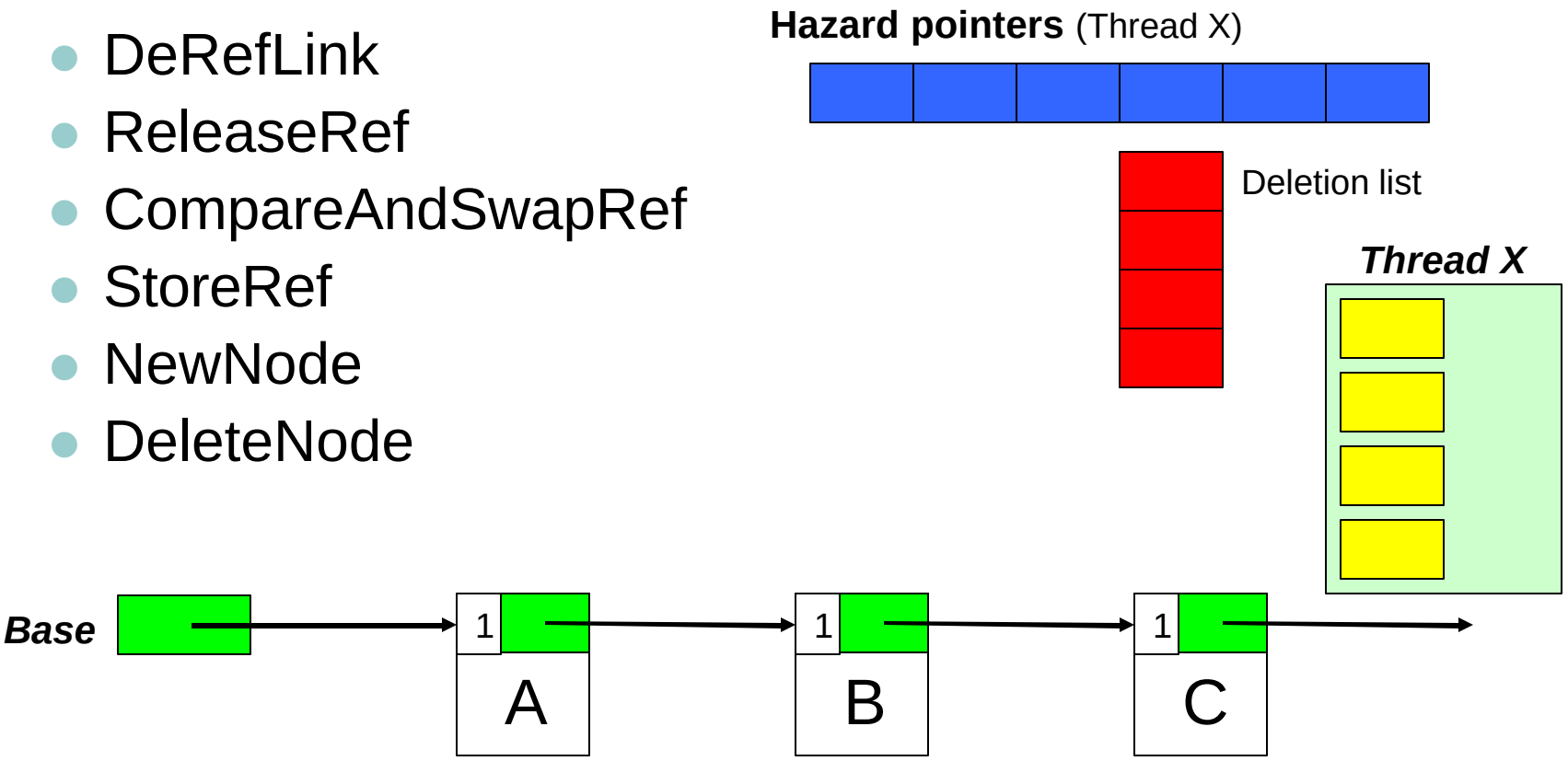


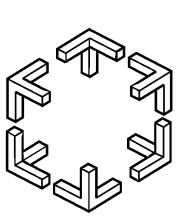
Our approach – The basic idea

- Combine the best of
  - Hazard pointers (Michael 2002)
    - Tracks references from threads
    - Fast de-reference
    - Upper bound on the amount of unreclaimed deleted nodes
    - Compatible with standard memory allocators
  - Reference counting
    - Tracks references from links in shared memory
      - Manages links within dynamic nodes
      - Safe to traverse links (also) in deleted nodes
- Practical
  - Uses only single-word Compare-And-Swap

The basic idea

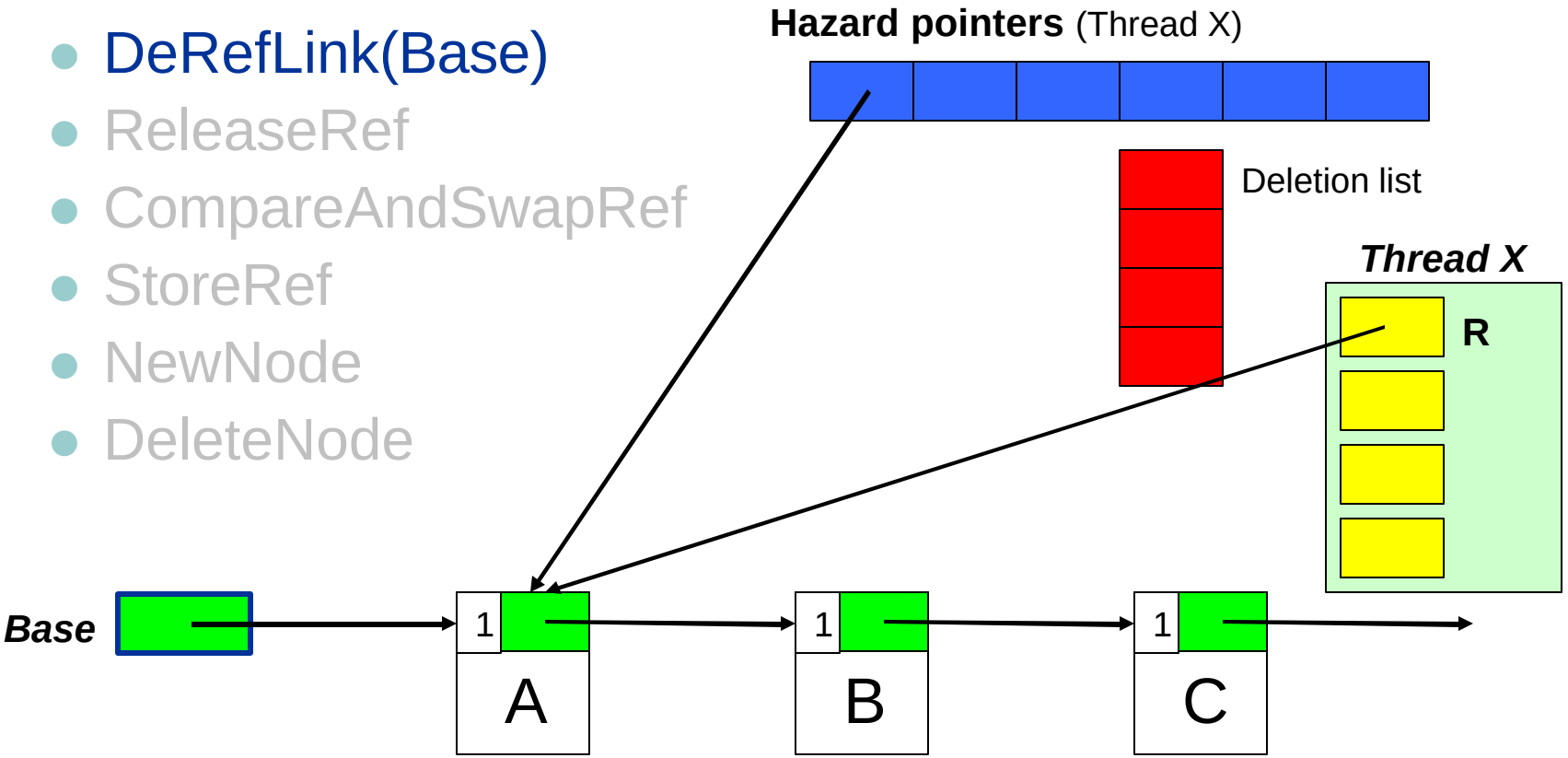
- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode





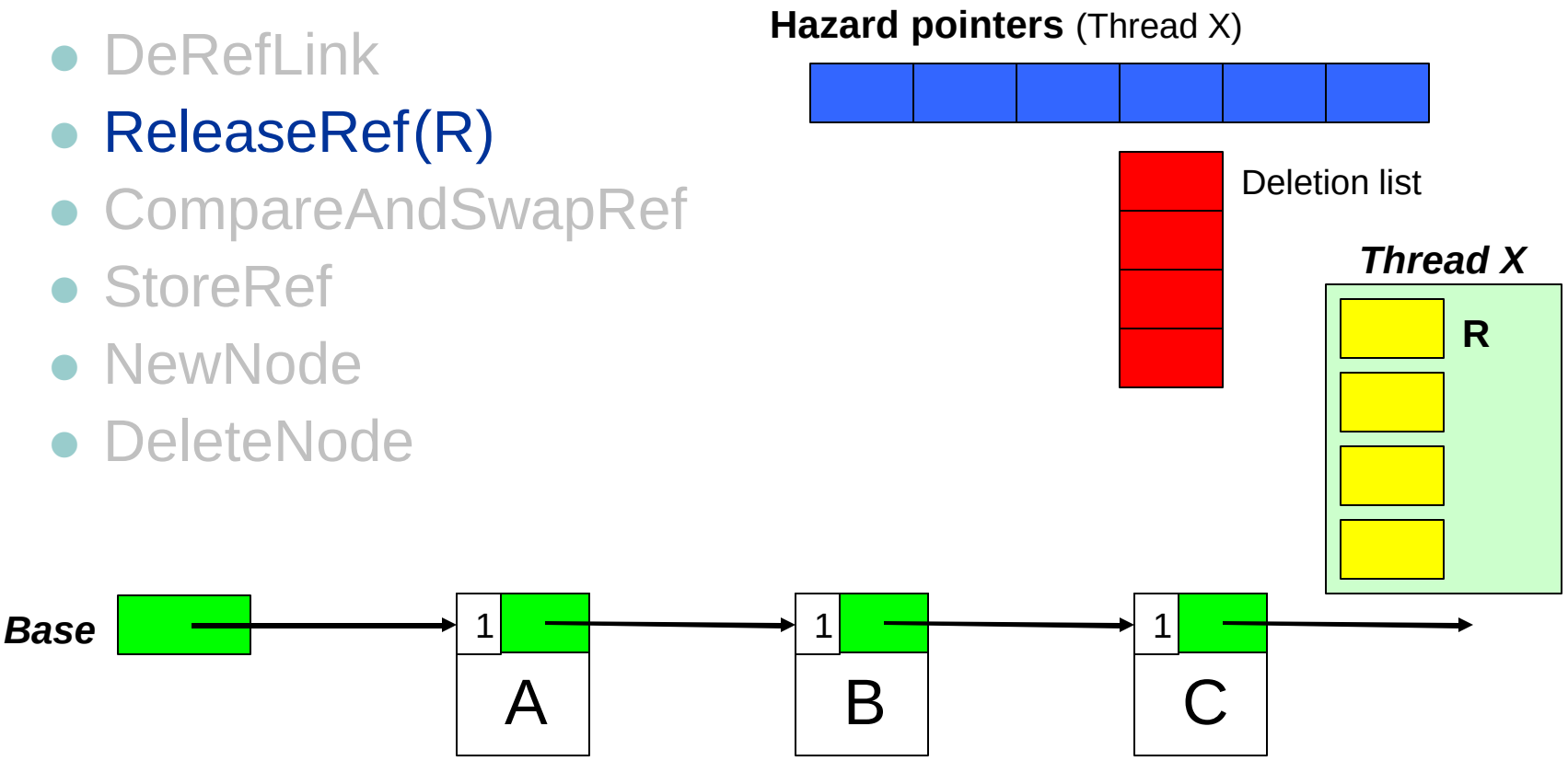
The basic idea

- API
  - DeRefLink(Base)
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode



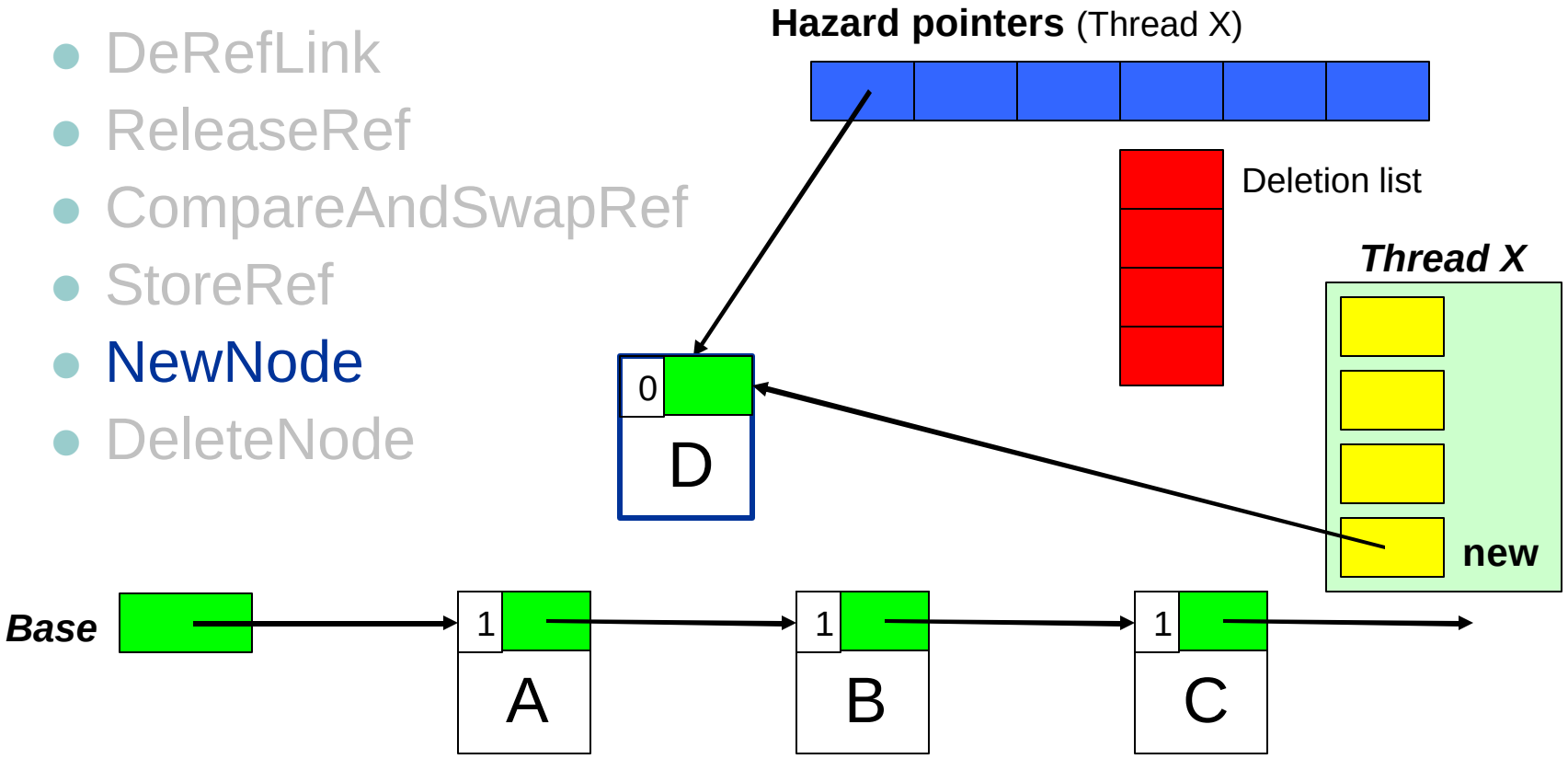
The basic idea

- API
  - DeRefLink
  - ReleaseRef(R)
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode



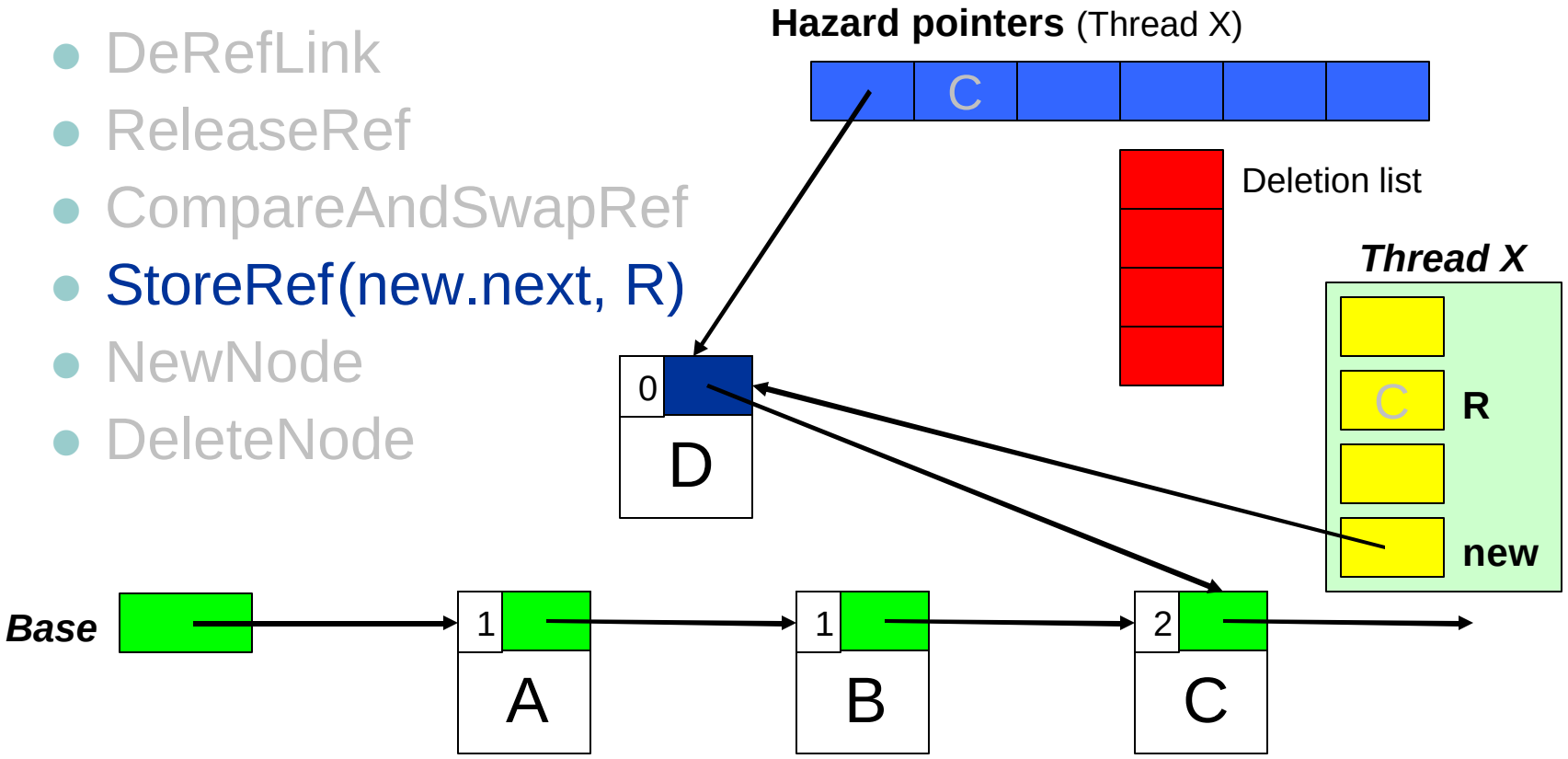
The basic idea

- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode



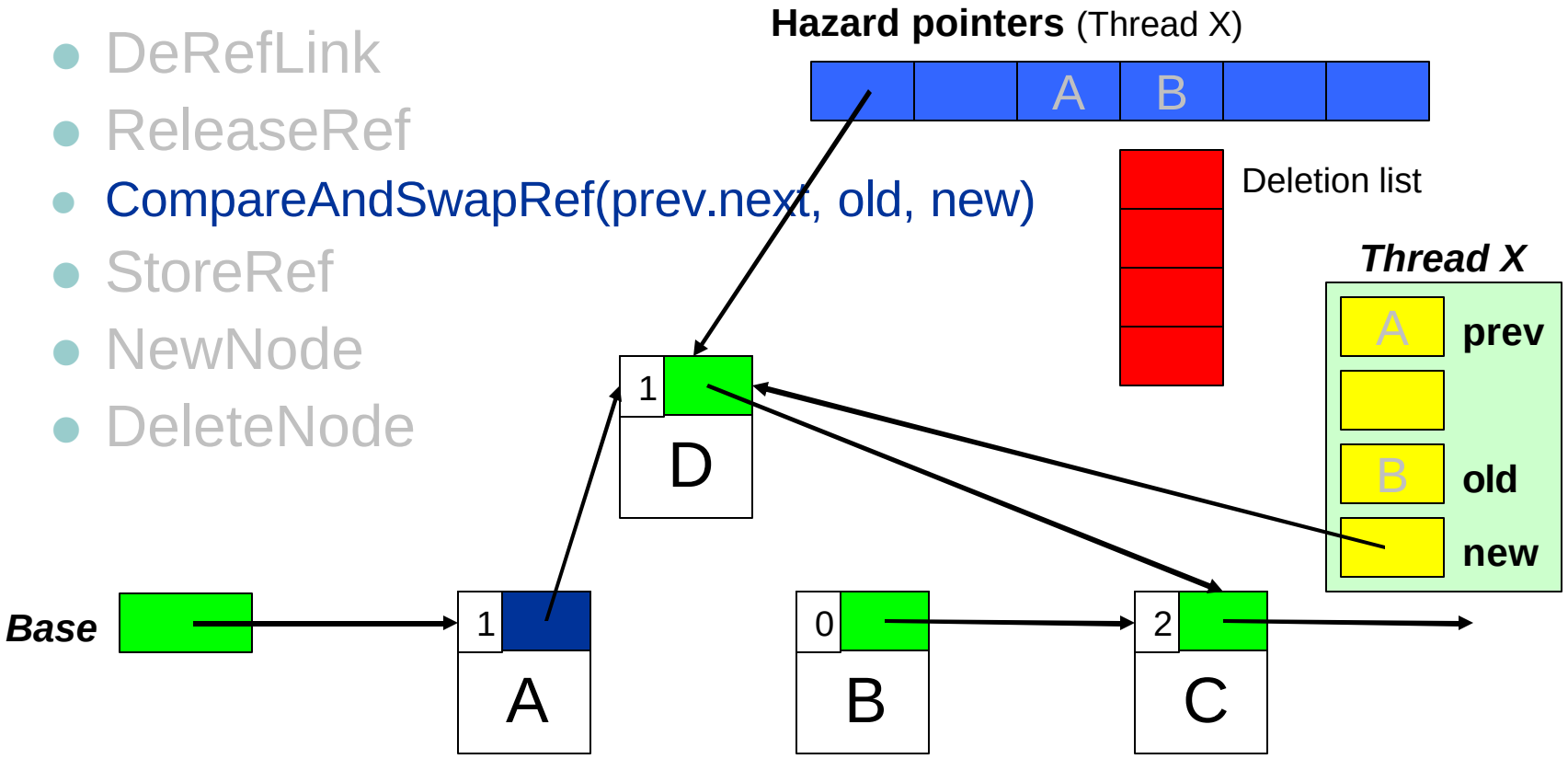
The basic idea

- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef(new.next, R)
  - NewNode
  - DeleteNode



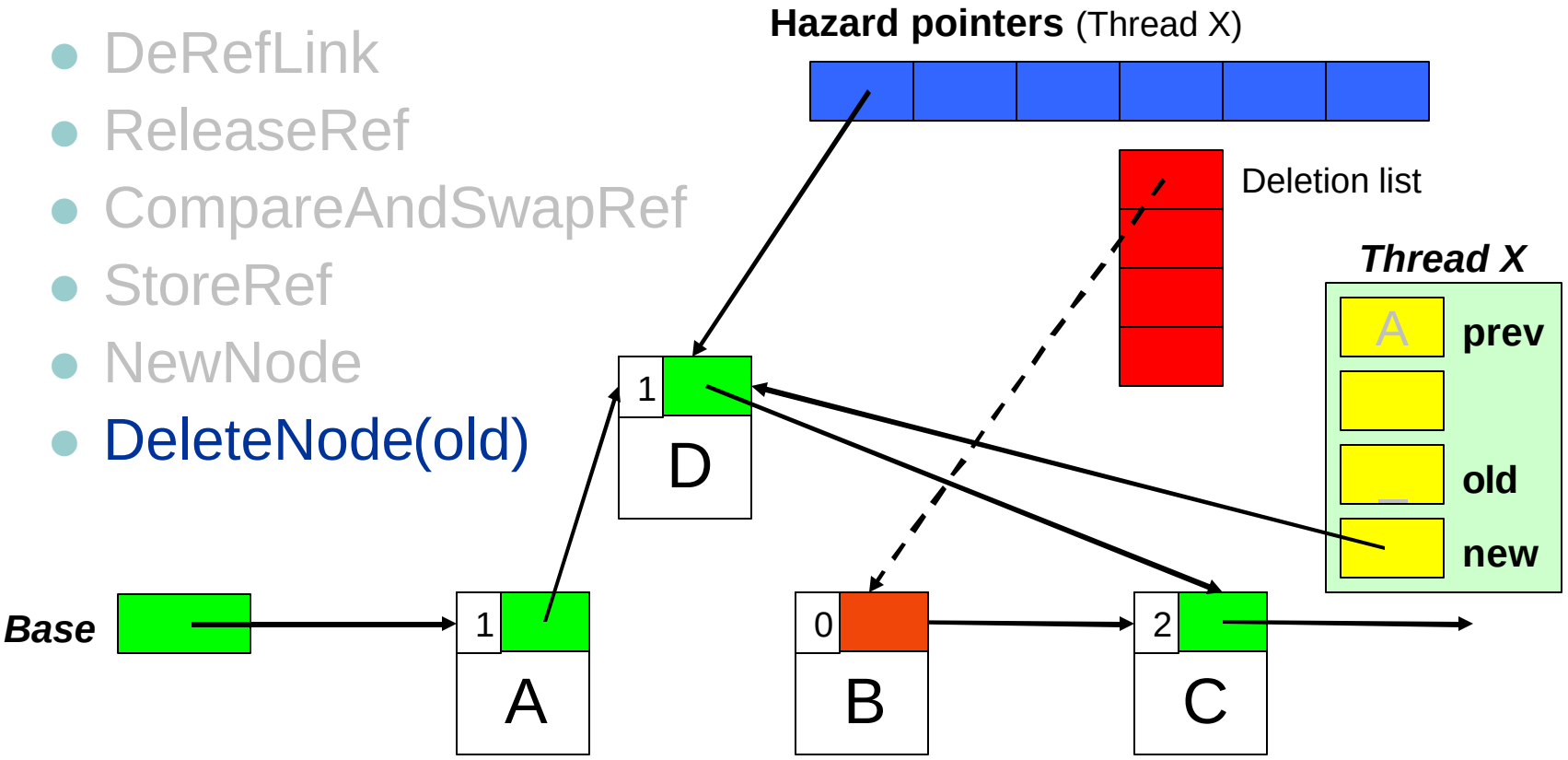
The basic idea

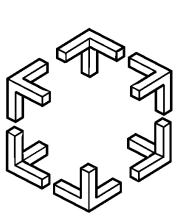
- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef(prev.next, old, new)
  - StoreRef
  - NewNode
  - DeleteNode



The basic idea

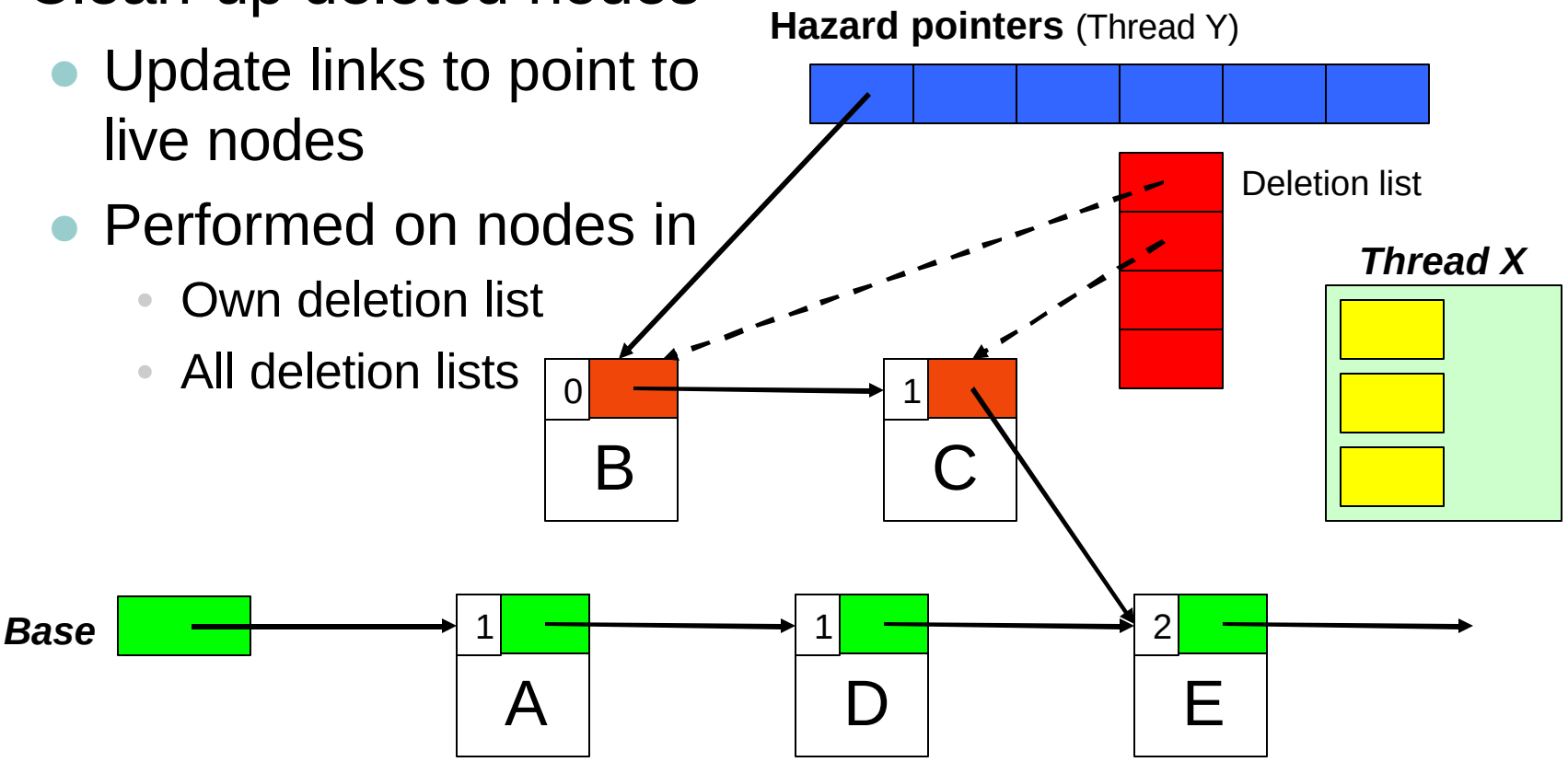
- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode(old)

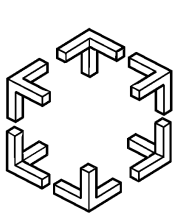




Breaking chains of garbage

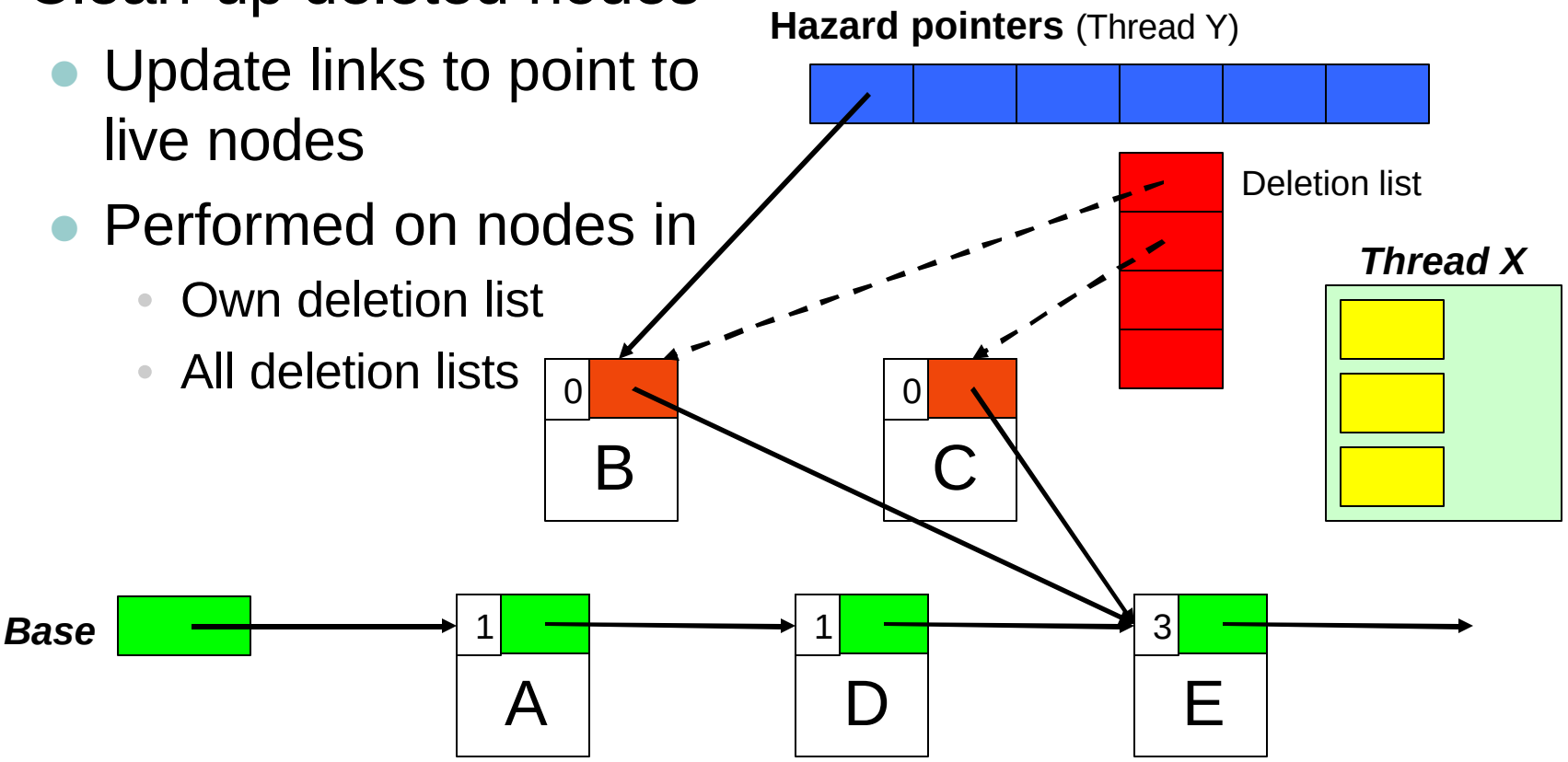
- Clean-up deleted nodes
  - Update links to point to live nodes
  - Performed on nodes in
    - Own deletion list
    - All deletion lists

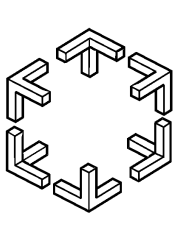




Breaking chains of garbage

- Clean-up deleted nodes
  - Update links to point to live nodes
  - Performed on nodes in
    - Own deletion list
    - All deletion lists

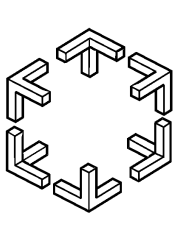




Bound on unreclaimed nodes

- A deleted node can be reclaimed when
  - The reference count is zero and
  - No hazard pointer is pointing to it and
  - There is no ongoing clean-up of this node

- With a rate relative to the number of threads of
  - Scanning hazard pointers
  - Cleaning up nodes as needed
- Then the maximum size of each deletion list depends on
  - The number of hazard pointers
  - The number of links per node
  - The number of threads



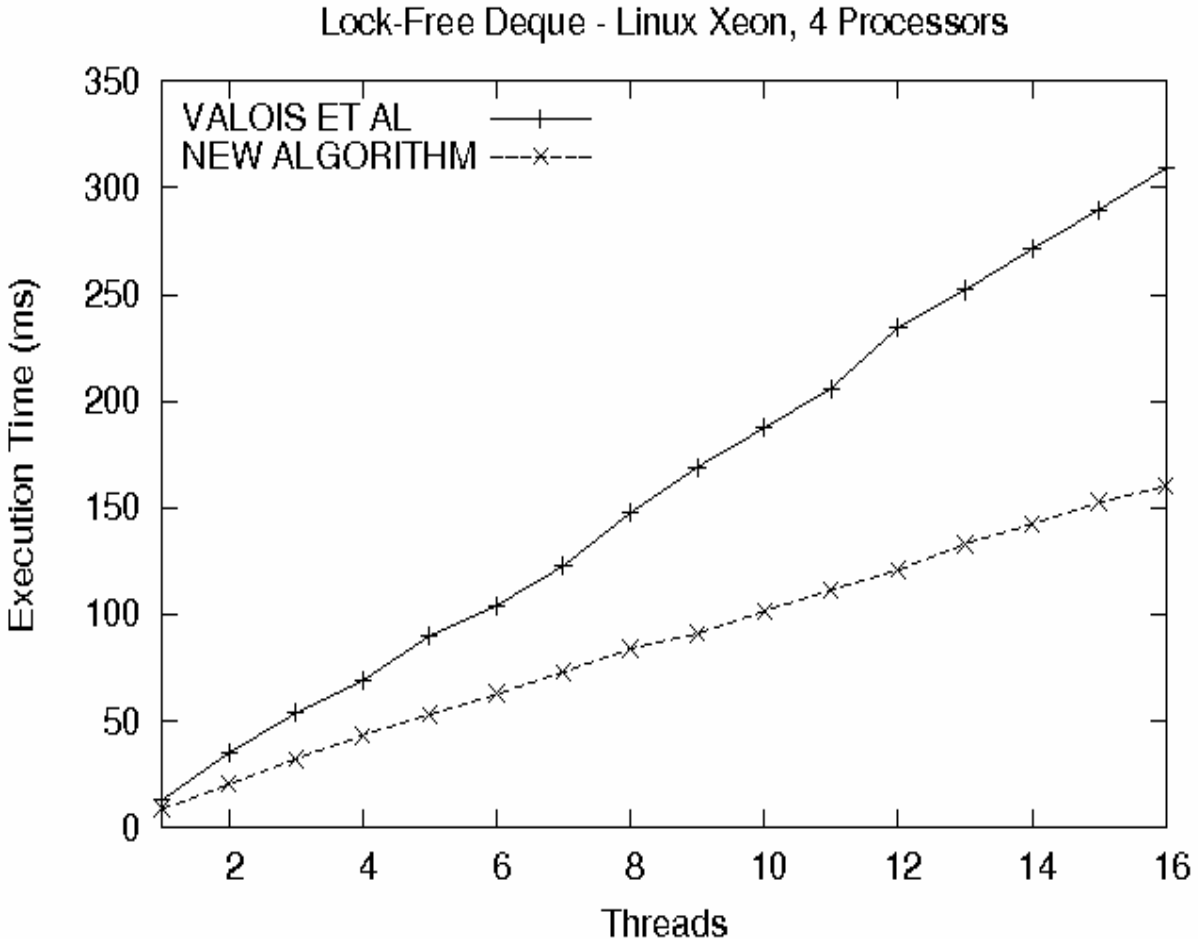
Experimental evaluation

- Lock-free deque (Sundell and Tsigas 2004)

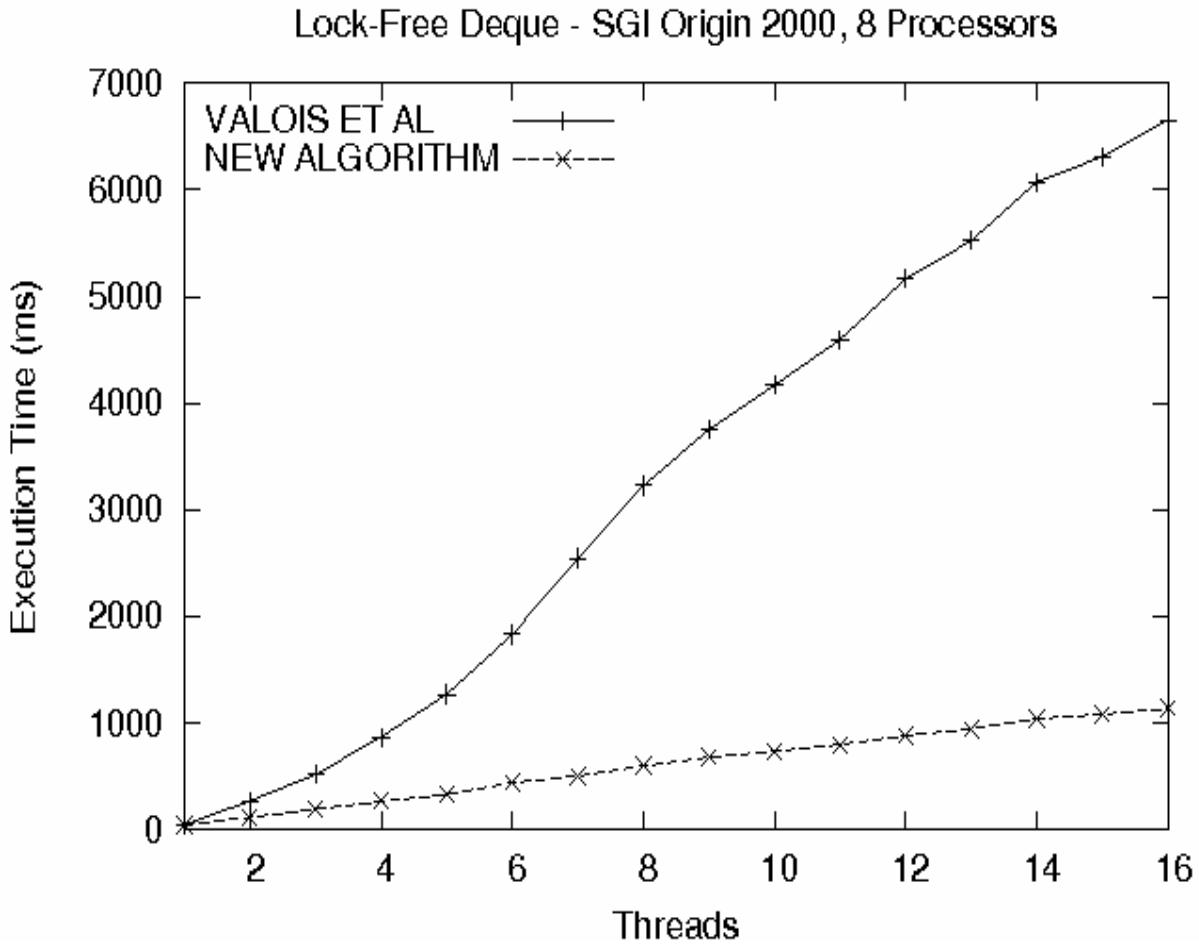
  (deque – double-ended queue)

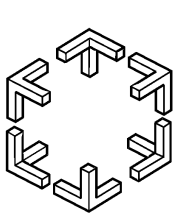
  - The algorithm needs traversal of deleted nodes
  - Time for 10000 random operations/thread
- Tested memory reclamation schemes
  - Reference counting, Valois et al.
  - The new algorithm
- Systems
  - 4 processor Xeon PC / Linux (UMA)
  - 8 processor SGI Origin 2000 / IRIX (NUMA)

Experimental evaluation



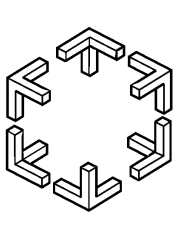
Experimental evaluation





Conclusions

- First lock-free memory reclamation scheme that
  - Only uses atomic primitives available in contemporary architectures
  - Guarantees safety of
    - Local and
    - Global references
  - Has an upper bound on the amount of deleted but unreclaimed nodes
  - Allows arbitrary reuse of reclaimed memory



Questions?

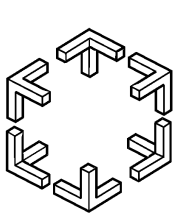
- Contact Information:
  - Address:

    Anders Gidenstam,
    Computer Science & Engineering,
    Chalmers University of Technology,
    SE-412 96 Göteborg, Sweden
  - Email:

    andersg @ cs.chalmers.se
  - Web:

    http://www.cs.chalmers.se/~dcs
    http://www.cs.chalmers.se/~andersg
  - Implementation

    http://www.noble-library.org/



Conclusions

- First lock-free memory reclamation scheme that
  - Only uses atomic primitives available in contemporary architectures
  - Guarantees safety of
    - Local and
    - Global references
  - Has an upper bound on the amount of deleted but unreclaimed nodes
    (Bound: N * N * (k + L_max + a + 1))
  - Allows arbitrary reuse of reclaimed memory