A LAZY CONCURRENT LIST-BASED SET ALGORITHM

30 MARCH 2023

Authors: Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer III, and Nir Shavit

Presenter: Jun-Qing Lim

UNIVERSITY OF WATERLOO | DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE

Problem

- List-based sets: basic building blocks for concurrent algorithms
- Current lock-free algorithms (mostly) require the operations with list traversals:
  - To use a mark bit as reference of nodes being logically removed
  - Then perform cleanup – which is expensive (due to overheads on “helping to cleanup”)

Ways to Synchronize Lists

1. Coarse-grained
   - single lock
   - simple but no concurrency
2. "hand-over-hand"
   - lock for each successive node before releasing predecessor's lock
   - higher concurrency than (1) but acquire many locks
   - in high contention situations, thread 1 waits for thread 2, thread 2 waits for thread 3…
   - which slows down the application (expensive)

New Idea Proposed – "lazy" synchronization

- Based on "optimistic" locking scheme
- 3 main operations:
  - `add(x)`
    - adds x into the set, returns true if x is not already in the set
  - `remove(x)`
    - removes x from the set, return true if x was in the set
  - `contains(x)`
    - returns true if the set contains x

Algorithm - Overview

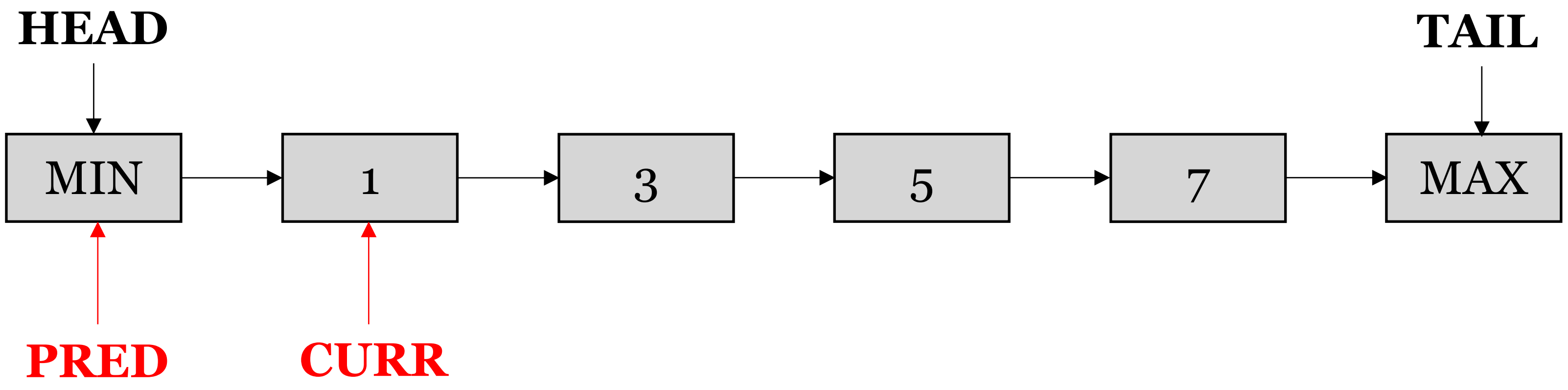
- Each operation searches without acquiring locks / interfering others
- Only locks when it locates the position to add/remove → synchronization method!

- In removing, nodes are marked as removed (logical) → then physically unlinked
- Other threads do not “help” in unlinking while traversing (hence “lazy”!)
- Avoids unnecessary overheads

Algorithm - Design

- Maintain list in increasing order
- Locks when position is located
- Traverse to search for 2…. (for update ops)

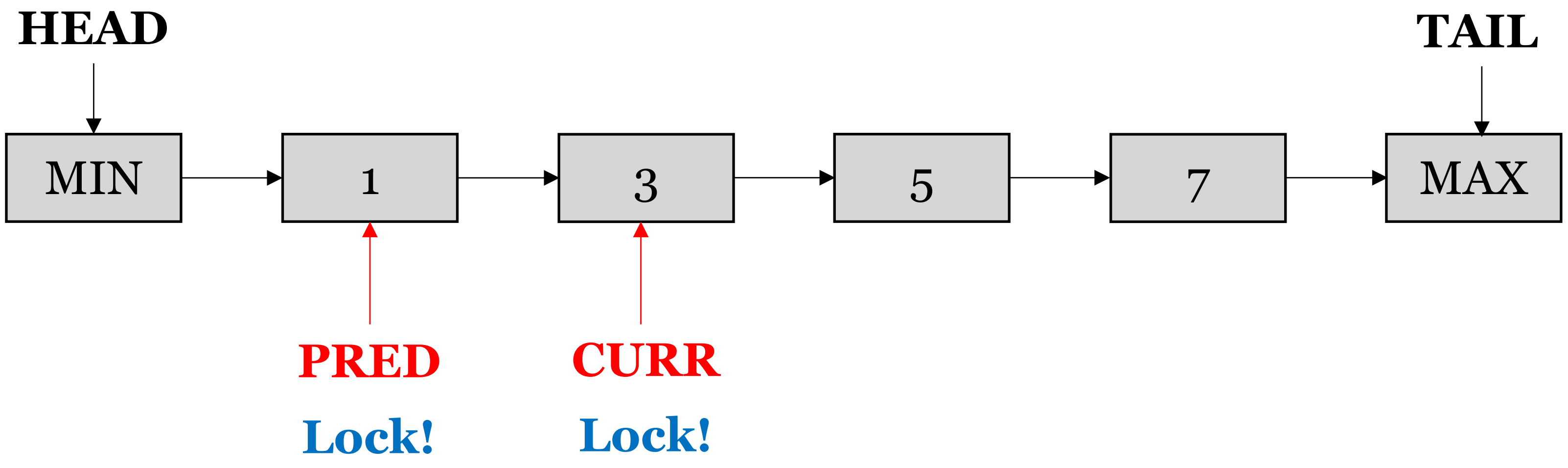
```
struct Entry {
  int key; // value of element
  Entry next; // reference to the next object
  bool marked; // logically removed?
  mutex lock; // used for synchronization
}
```



Algorithm - Design

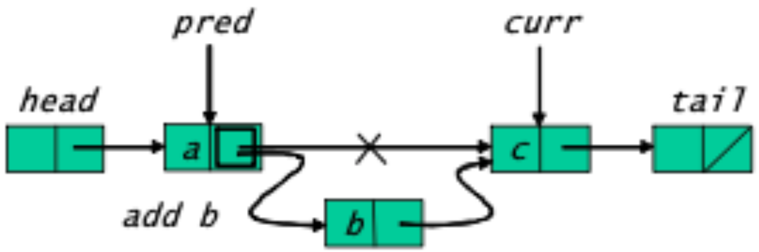
- Maintain list in increasing order
- Locks when position is located
- Traverse to search for 2…. (for update ops)

```
struct Entry {
  int key; // value of element
  Entry next; // reference to the next object
  bool marked; // logically removed?
  mutex lock; // used for synchronization
}
```



Algorithm – add()

1. Traverse to find entry of `curr.key >= key`
2. Locks the predecessor and current
3. Validate to see if the values are changed
4. Perform insert OR restart
5. Always unlocks the entries




```
bool add(int key) {
  while (true) {
    Entry pred = head;
    Entry curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            return false;
          } else {
            Entry entry = Entry(key);
            entry.next = curr;
            pred.next = entry;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

Find position

Locks

Insert

UNIVERSITY OF WATERLOO | DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE

Algorithm – add()

- Why validate?
  - Gap between unsynchronized traversal and locks
  - pred/curr could be removed
  - Some entry inserted between pred and curr

```
bool validate(Entry pred, Entry curr) {
  return !pred.marked && !curr.marked && pred.next = curr;
}
```

```
bool add(int key) {
  while (true) {
    Entry pred = head;
    Entry curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            return false;
          } else {
            Entry entry = Entry(key);
            entry.next = curr;
            pred.next = entry;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

Find position

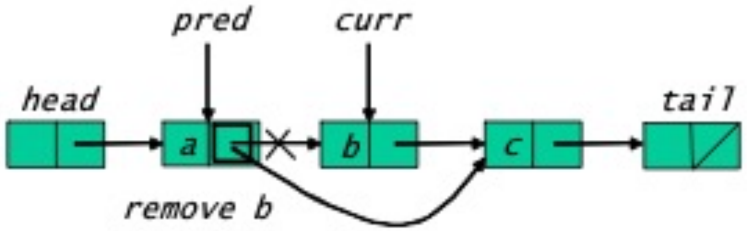
Locks

Insert

UNIVERSITY OF WATERLOO | DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE

Algorithm – remove()

1. Traverse to find entry of `curr.key >= key`
2. Locks the predecessor and current
3. Validate to see if the values are changed
4. Perform remove OR restart
   - Logical removal
   - Physical removal
5. Always unlocks the entries




```
bool remove(int key) {
  while (true) {
    Entry pred = head;
    Entry curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key != key) {
            return false;
          } else {
            curr.marked = true;
            pred.next = curr.next;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
}
```

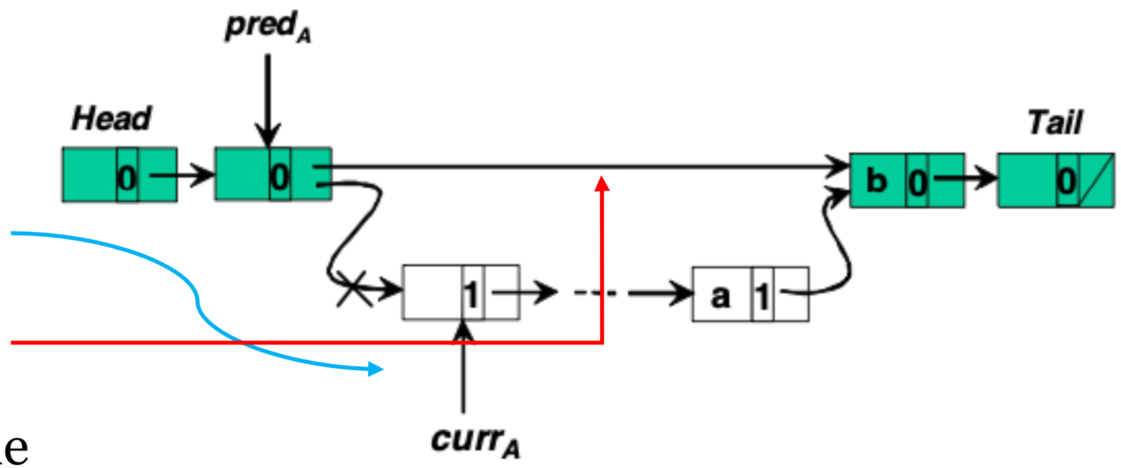
Find position

Locks

Removes

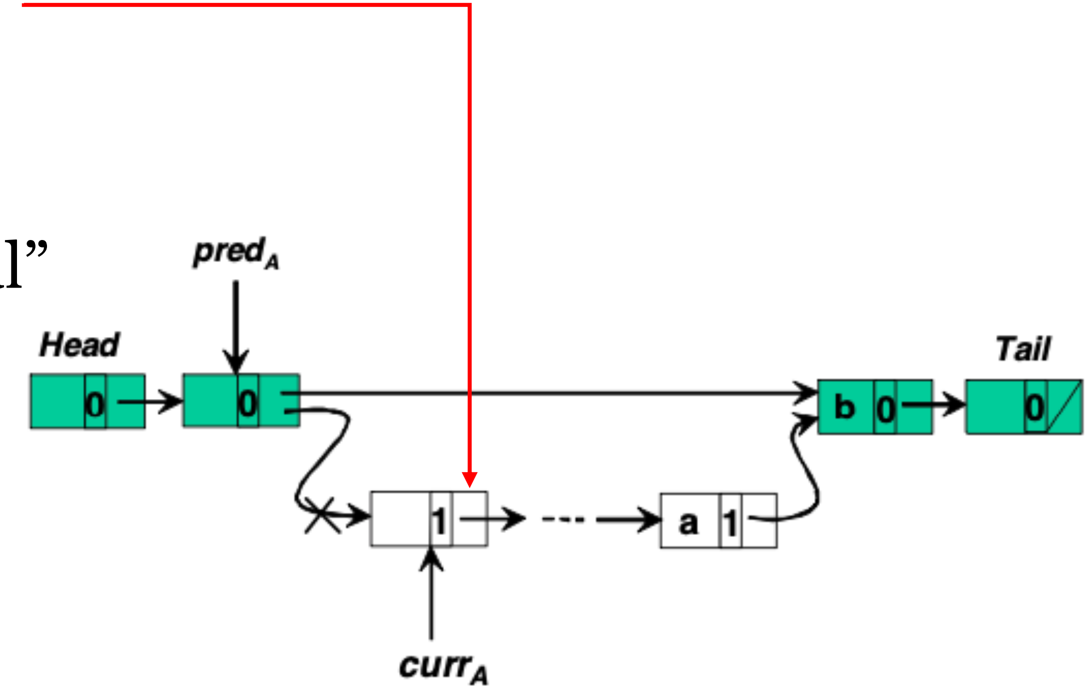
What happens to concurrent threads traversal?

- Can traverse logically/physically removed nodes
  - Suppose thread A traverses with pred and curr
  - Then another thread B removes the node currA
  - currA can still move to the next node because the pointer is still there



What happens to concurrent threads traversal?

- Say if some thread stops its search at the removed node, and performs some operation
- Operation can still return correct result
- Because nodes are marked as “logical removal”
- Validate checks for `!marked`



Algorithm – contains()

- Since node is marked for logical removal
- Also used to determine if it's still present
- Usual traversal + return true if == key and unmarked
- Correct because removed nodes must be marked or not present at all

```
bool contains(int key) {
  Entry curr = head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && ! curr.marked;
}
```

Advantage?

1. `contains()` is wait-free - does not interfere with any concurrent operations
2. Traversals are not delayed by physical removals in remove()
   - Eventually a thread will be able to decide if the node really exists through the marking

- Compared to Michael's lock-free list algorithm
  - When it encounters a logically removed node, the thread "helps in physical removal" with CAS
  - Traversals to be restarted and drop in performance

Performance

- Compared 6 different algorithms with different locking techniques

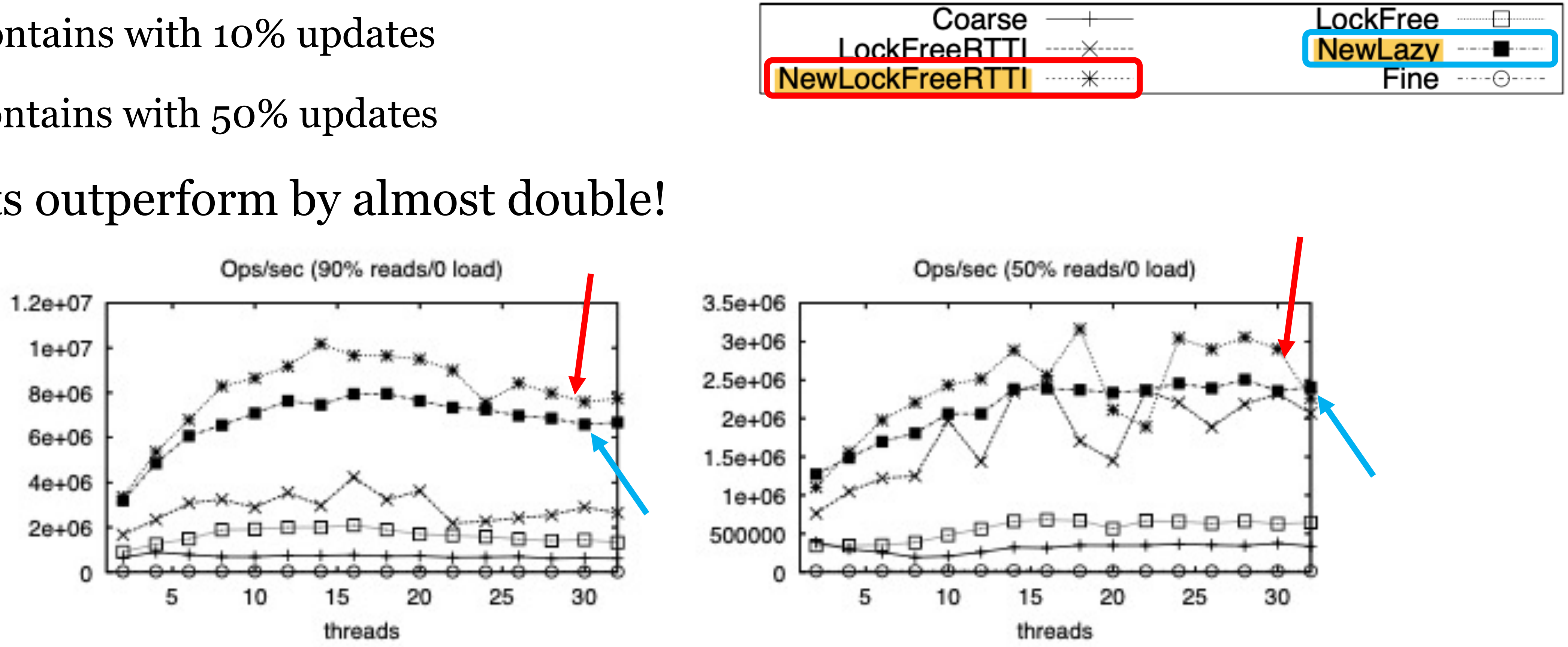
1. **Coarse**: single lock
2. **Fine**: “hand-over-hand” locking (1 lock per item)
3. **LockFree**: lock-free list using marking but not wait-free for contains() [Michael’s]
4. **LockFreeRTTI**: improved version of LockFree + Java’s RTTI mechanism

   RTTI: Run-Time Type Identification – exposes type info during runtime

5. **NewLockFreeRTTI**: algo (4) but wait-free using the new `contains()`
6. **NewLazy**: the new algorithm!

Results?

- Two tests:
  - 90% contains with 10% updates
  - 50% contains with 50% updates
- Both tests outperform by almost double!



Conclusion

- Paper introduced lazy list – based on lazy marking and deletion of nodes
- Main improvement performance comes from the wait-free searching

THANK YOU!