A Pragmatic Implementation of Non-Blocking Linked-Lists

By Arveen Emdad, Hamit Efe Çınar, Iason Kaxiras

Motivations

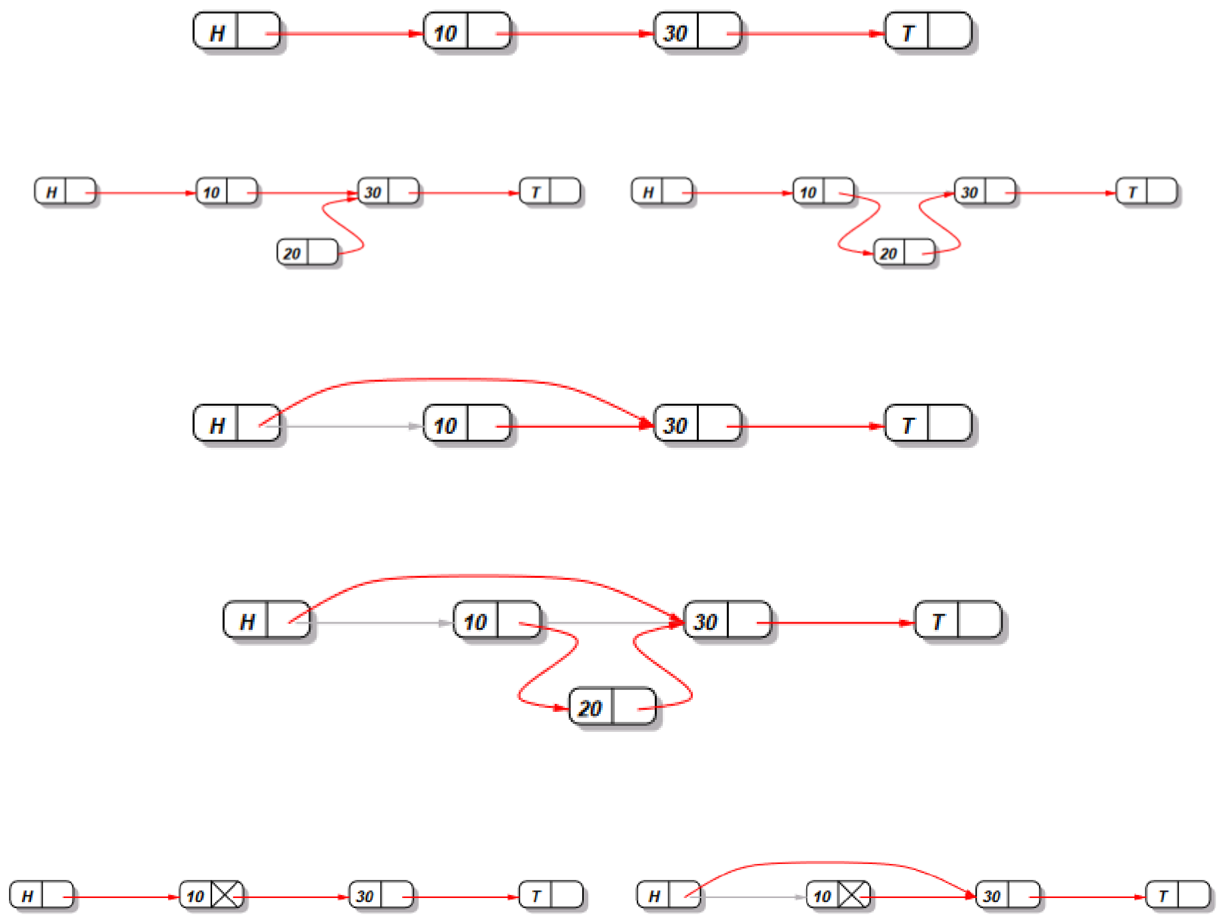
- Efficiently parallelize linked lists
- Linked lists are a basic data structure that if implemented with a non-blocking method would be very efficient.
- Other data structures can be based of linked lists

1. Create a non-blocking implementation of concurrent linked-lists supporting linearizable insertion and deletion operations.
2. Design an algorithm that is conceptually simpler and substantially faster than previous schemes.
3. Utilize non-blocking algorithms for parallel systems to deliver significant advantages using low-level atomic primitives and avoid the use of locks.

Related Work

- Herlihy - generalized non-blocking CAS-based implementation
    - Highly centralized
    - Suffers from poor performance (write to shared global pointer)
- Valois - more complex CAS-based implementation
    - Highly distributed
    - Harder to implement (much low-level implementation comparingly)
- Greenwald - proposed alternative with DCAS (Double Compare-And-Swap)
    - Not available on multi-processor architectures
    - Suggested the insertion algorithm's logic used in the paper

Implementation Overview



- Single CAS is not sufficient.
- Issues of concurrent operations (Insertion while deletion)
- Marking Nodes (logical and physical deletions)

Algorithm: Type Definitions

```
class List<KeyType> {
    Node<KeyType> *head;
    Node<KeyType> *tail;

    List() {
        head = new Node<KeyType>();
        tail = new Node<KeyType>();
        head.next = tail;
    }
}
```

```
class Node<KeyType> {
    KeyType key;
    Node *next;

    Node(KeyType key) {
        this.key = key;
    }
}
```

Algorithm: Insert

```
public boolean List::insert(KeyType key) {
    Node *new_node = new Node(key);
    Node *right_node, *left_node;
    do {
        right_node = search(key, &left_node);
        if ((right_node != tail) && (right_node.key == key)) /*T1*/
            return false;
        new_node.next = right_node;
        if (CAS(&(left_node.next), right_node, new_node)) /*C2*/
            return true;
    } while (true); /*B3*/
}
```

Algorithm: Delete

```
public boolean List::delete(KeyType search_key) {
    Node *right_node, *right_node_next, *left_node;
    do {
        right_node = search(search_key, &left_node);
        if ((right_node == tail) || (right_node.key != search_key)) /*T1*/
            return false;
        right_node_next = right_node.next;
        if (!is_marked_reference(right_node_next))
            if (CAS(&(right_node.next), /*C3*/
              right_node_next, get_marked_reference(right_node_next)))
                  break;
    } while (true); /*B4*/

    if (!CAS(&(left_node.next), right_node, right_node_next)) /*C4*/
        right_node = search(right_node.key, &left_node);
    return true;
}
```

Algorithm: Find

```
public boolean List::find(KeyType search_key) {
    Node *right_node, *left_node;
    right_node = search(search_key, &left_node);
    if ((right_node == tail) || (right_node.key != search_key))
        return false;
    else
        return true;
}
```

Algorithm: Search (pt.1)

```
private Node *List::search(KeyType search_key, Node **left_node) {
    Node *left_node_next, *right_node;
search_again:
    do {
        Node *t = head;
        Node *t_next = head.next;

        /* 1: Find left_node and right_node */
        do {
            if (!is_marked_reference(t_next)) {
                (*left_node) = t;
                left_node_next = t_next;
            }
            t = get_unmarked_reference(t_next);
            if (t == tail) break;
            t_next = t.next;
        } while (is_marked_reference(t_next) || (t.key<search_key)); /*B1*/
        right_node = t;
        ...
}
```

Algorithm: Search (pt.2)

```
private Node *List::search(KeyType search_key, Node **left_node) {

        ...

        /* 2: Check nodes are adjacent */
        if (left_node_next == right_node)
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again; /*G1*/
            else
                return right_node; /*R1*/

        ...

}
```

Algorithm: Search (pt.3)

```
private Node *List::search(KeyType search_key, Node **left_node) {
        ...

        /* 3: Remove one or more marked nodes */
        if (CAS(&(left_node.next), left_node_next, right_node)) /*C1*/
           if ((right_node != tail) && is_marked_reference(right_node.next))
               goto search_again; /*G2*/
           else
               return right_node; /*R2*/

     } while (true); /*B2*/
}
```

Correctness

- Proof Sketch:
  - Linearizability
  - Progress
- Model Checking
- Practical Testing

Proof Sketch: Linearizability

- **Linearization Points Identification**:
  - The instant at which an operation appears to occur atomically is identified.
  - This is crucial for proving linearizability, ensuring operations appear instantaneously and in order.
- **Conditions Maintained by Search**:
  - Relies on conditions identified in Section 4.1, especially during the invocation of **List::search**.
  - **Ordering Constraints**:
    - When initializing the right node, the loop ensures **search_key <= right_node.key**.
    - For the left node, **left_node.key < search_key**, otherwise, the loop would have terminated earlier.
  - **Adjacency Condition and Mark State**:
    - Different return paths from **List::search** (like R1 and R2) are considered to confirm the adjacency of nodes and their mark states.
- **Mark State of Nodes**:
  - At both return paths, the right node is confirmed to be unmarked.
  - Nodes never become unmarked once marked, implying the right node was unmarked at earlier points.
- **Defining Linearized Order**:
  - Based on the real-time **d_{i,m}** when **List::search** post-conditions are satisfied.
  - This defines the order of operations like Find, Insert, and Delete.
  - Criteria for successful and unsuccessful operations are specified.

Proof Sketch: Progress

- **Non-blocking Implementation**:
  - Shows that the implementation is non-blocking, meaning operations always make progress without waiting indefinitely.
- **Updates Caused by Operations**:
  - Successful insertions cause exactly one update.
  - Successful deletions cause at most two updates.
  - Unsuccessful operations cause no updates.
- **Behavior of CAS Instructions**:
  - CAS (Compare-And-Swap) instructions like C1 and C4 unlink marked nodes and are bounded by the number of marked nodes.
  - Each successful deletion marks exactly one node, impacting the number of updates.
- **Consideration of Backwards Branches**:
  - Analyzes different loop conditions and CAS instruction failures, ensuring list integrity and operation correctness.
  - Addresses how nodes in the list are advanced, modified, and marked during different operations.

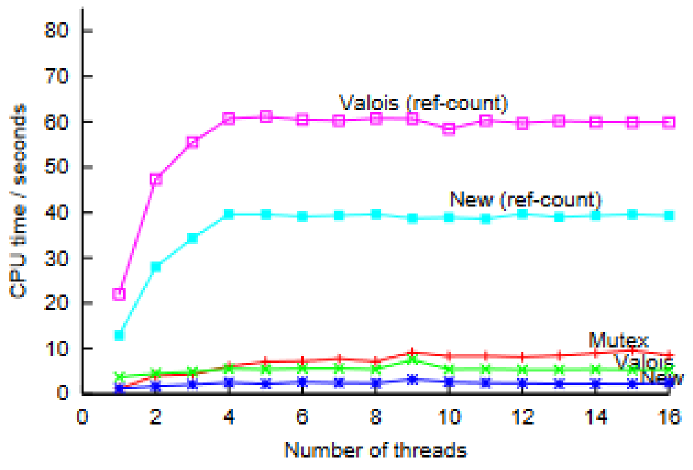
Model Checking

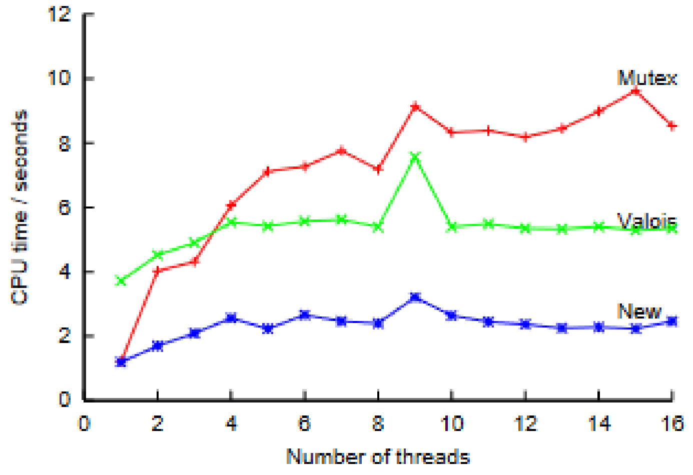
- **Model Checker Used**: dSPIN, an extension of SPIN model checker.
  - Adds support for pointers and storage scopes.
  - More suitable for specific problem domains requiring these features.
- **State Representation**:
  - **Linked List of Cells**: Represents one version of the set.
    - Updated using atomic **d step** instructions.
    - Implements Compare-And-Swap (CAS) operations.
  - **Bit Vector**: Another representation of the set.
    - Checked and updated at linearization points using **d step** instructions.
- **Model Parameterization**:
  - Based on the number of concurrent threads.
  - Determined by the number of operations each thread attempts.
  - Range of key values that can be used in the model.
- **Testing Configurations**:
  - Various setups including:
    - Four threads, each performing one operation.
    - Two threads, each performing two operations.
  - Tested with up to four potential keys.
- **Purpose of Model Checking**:
  - To exhaustively verify the extended capabilities of SPIN (via dSPIN) in handling memory management and function operations.
  - To ensure the correctness of the algorithm under different thread and operation configurations.
  - To verify properties like safety (preventing errors) and liveness (ensuring progress) in the context of concurrent executions.

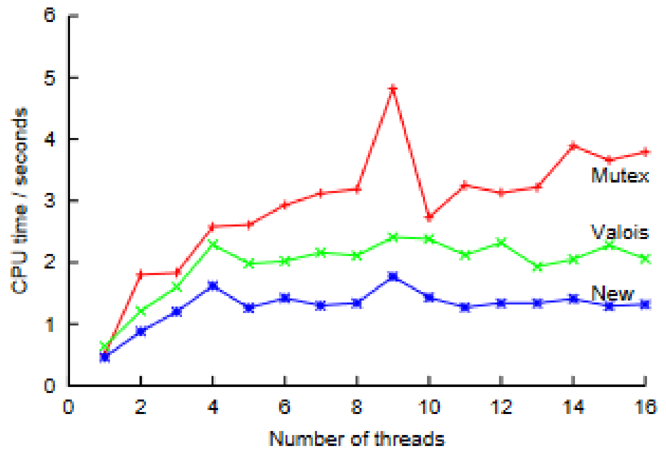
Practical Testing

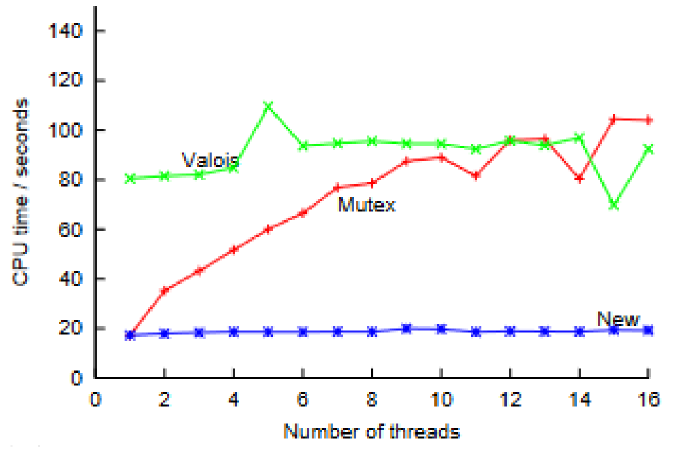
- **Testing Focus**: Evaluating the linearizability of operations in the presence of relaxed memory models.
  - Relaxed memory models present unique challenges compared to conventional shared memory machines.
  - Direct barrier instructions are used, reducing the need for simplifying assumptions.
- **Testing Linearizability**:
  - Ensuring operations are linearizable (i.e., appear to occur instantaneously at some point between their start and end times).
  - Acknowledges that while testing can't provide the same assurances as formal methods, it remains important.
- **Tractability Issues**:
  - Practical limitations in recording actual timestamp values for operations in a running process.
  - Operations are surrounded by code at each linearization point to record coherent per-processor cycle counts.
- **Data Collection and Analysis**:
  - Intervals of operations are recorded in an in-memory log.
  - This log is then replayed sequentially in timestamp order for analysis.
  - Comparison is made between logged results and concurrent execution outcomes.
- **Dealing with Overlapping Intervals**:
  - The replay program uses simple heuristics to handle overlapping intervals.
  - If a consistent linearized order is not evident, the program reports unresolved inconsistencies.
  - These inconsistencies require manual inspection and re-ordering for resolution.

Results


CPU time / seconds
Number of threads
Valois (ref-count)
New (ref-count)
Mutex
Valois
New


CPU time / seconds
Number of threads
Mutex
Valois
New


CPU time / seconds
Number of threads
Mutex
Valois
New


CPU time / seconds
Number of threads
Valois
Mutex
New

Delete: greater or equal than

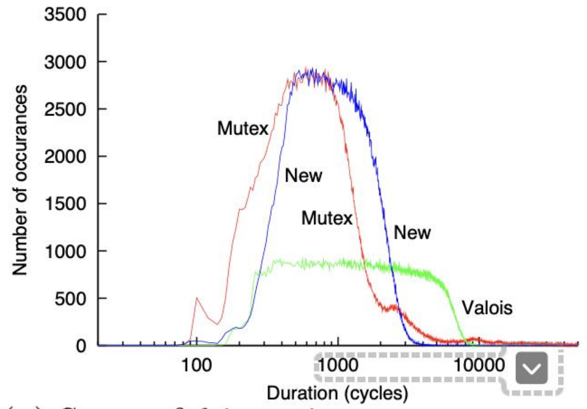
- Three insertion operations in sequence:
  - Insert(20)
  - Insert(15)
  - Insert(20)

  Concurent deleteGE(10)

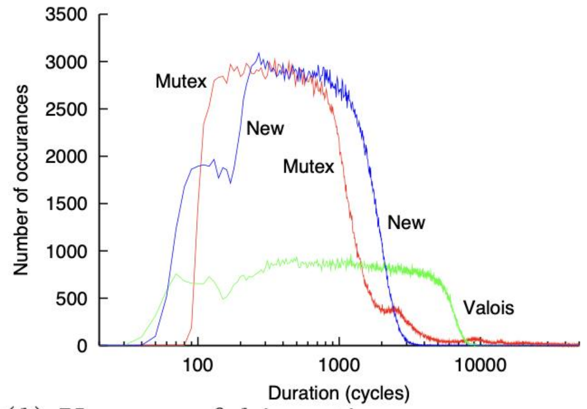
- List::deleteGE invokes List:: search
- The succesfull insertion of 20 occurs
- The unsuccessfull insertion of 20 occurs
- List::deleteGE(10) completes after deleting node containing 20

Delete: greater or equal than

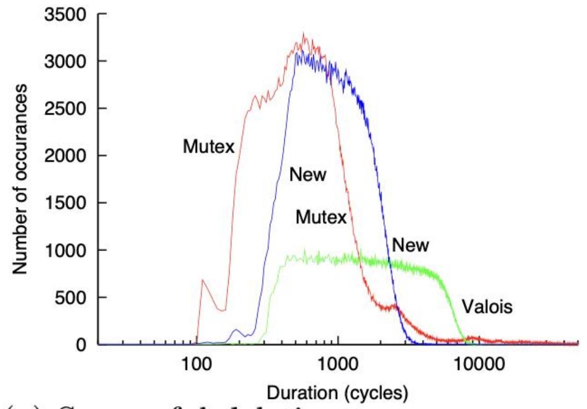
- •Sequential execution constraint: Deletion of 10 must be placed before insertion of 15 to ensure that 20 is returned instead of 15.
- •Linearization constraint: Deletion after failed insertion of 20 must be ensured to prevent the insertion from succeeding.
- •Irreconcilable constraints: The constraints of sequential execution and linearization conflict, as satisfying one leads to violation of the other.
- •Retain List::deleteGE implementation but modify List::insert to ensure failure of C3 whenever a new node might be inserted between left and right nodes.
- •Key consideration: At execution of C3, the right node should not necessarily be the immediate successor of the left node, especially considering concurrent updates.
- •So we update nodes whose keys are greater than or equal to the search key during List::deleteGE execution.
- •Use a single CAS operation to introduce a pair of new nodes, one containing the value being inserted and another duplicating the right node, while marking the original right node.



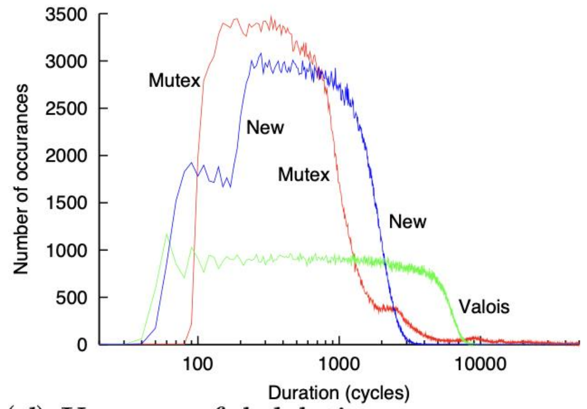
(a) Successful insertions



(b) Unsuccessful insertions



(c) Successful deletions



(d) Unsuccessful deletions

Conclusion

- The algorithms are linearizable
- Better performance than blocking algorithms
- Better performance than non-blocking alternatives

References

- Harris, Timothy. (2001). A Pragmatic Implementation of Non-Blocking Linked-Lists. 2180. 10.1007/3-540-45414-4_21.