Concurrent Data Structures

FAITH ELLEN AND TREVOR BROWN
UNIVERSITY OF TORONTO
Outline

• survey of interesting techniques
• evaluating concurrent data structures
Singly-Linked List

Delete node B
Singly-Linked List

Delete node C
Singly-Linked List

Delete node B

Delete node C

Node C is still in the list!
Global Lock

Lock the data structure. Only the process holding the lock can access the data structure.
Fine Grained Locking

Multiple locks protect different parts of the data structure
Hand-over-hand Locking

Multiple locks protect different parts of the data structure

\[
\text{START} \quad \longrightarrow \quad A \quad \longrightarrow \quad B \quad \longrightarrow \quad C \quad \longrightarrow \quad D \quad -
\]
Hand-over-hand Locking

Multiple locks protect different parts of the data structure
Hand-over-hand Locking

Multiple locks protect different parts of the data structure
Hand-over-hand Locking

Multiple locks protect different parts of the data structure
Fine Grained Locking

Multiple locks protect different parts of the data structure

• more concurrency
• harder to prove correctness and progress
• no fault tolerance
Lock-free Data Structures

- wait-free
- non-blocking
- obstruction-free
Lock-free Data Structures

• wait-free
• non-blocking
• obstruction-free

Every operation completes within a finite number of steps by the process that invoked it.
Lock-free Data Structures

- wait-free
- non-blocking
- obstruction-free

From every reachable configuration with a pending operation, some operation completes within a finite number of steps.
Lock-free Data Structures

• wait-free
• non-blocking
• obstruction-free

From every reachable configuration, every operation completes within a finite number of consecutive steps by the process that invoked it.
Have an auxiliary node between each consecutive pair of real nodes.
Non-blocking Singly-Linked List [Valois 1995]

Have an auxiliary node between each consecutive pair of real nodes.
Non-blocking Singly-Linked List [Valois 1995]
Non-blocking Singly-Linked List [Valois 1995]
Non-blocking Singly-Linked List [Harris 2001]

Each node has a bit, which is marked to logically delete the node. Once a node has been marked, it can’t be changed.
Non-blocking Singly-Linked List [Harris 2001]

Each node has a bit, which is marked to logically delete the node.
Once a node has been marked, it can’t be changed.
Non-blocking Singly-Linked List [Harris 2001]

Each node has a bit, which is marked to logically delete the node.
Once a node has been marked, it can’t be changed.
Non-blocking Singly-Linked List [Harris 2001]

Each node has a bit, which is marked to logically delete the node.
Once a node has been marked, it can’t be changed.
Each node has a bit, which is marked to logically delete the node.
Once a node has been marked, it can’t be changed.
Non-blocking Singly-Linked List [Harris 2001]

To insert a node:
• search list to find where to insert the new node
• set pointer of new node to point to its successor
• change pointer of its predecessor to point to new node
Non-blocking Singly-Linked List [Harris 2001]

To insert a node:
- search list to find where to insert the new node
- set pointer of new node to point to its successor
- change pointer of its predecessor to point to new node
Non-blocking Singly-Linked List [Harris 2001]

To insert a node:

• search list to find where to insert the new node
• set pointer of new node to point to its successor
• change pointer of its predecessor to point to new node
Non-blocking Singly-Linked List [Harris 2001]

To insert a node:
- search list to find where to insert the new node
- set pointer of new node to point to its successor
- change pointer of its predecessor to point to new node
Non-blocking Singly-Linked List [Harris 2001]

If it cannot complete the operation, the process must restart from the beginning of the list to find a new place to insert.

Amortized cost of operations: $\Theta(nc)$

$\begin{align*}
\text{n} &= \text{length of list} \\
\text{c} &= \text{contention}
\end{align*}$
Each node has a backlink. Just before the node is deleted, its backlink is set to point to its predecessor.
Non-blocking Singly-Linked List
[Fomitchev and Ruppert 2004]

flag predecessor
set backlink to point to its predecessor
mark node
change pointer of its predecessor to point to its successor and remove flag
Non-blocking Singly-Linked List
[Fomitchev and Ruppert 2004]

flag predecessor
set backlink to point to its predecessor
mark node
change pointer of its predecessor to point to its successor and remove flag
Non-blocking Singly-Linked List
[Fomitchev and Ruppert 2004]

flag predecessor
set backlink to point to its predecessor
mark node
change pointer of its predecessor to point to its successor and remove flag
Non-blocking Singly-Linked List
[Fomitchev and Ruppert 2004]

flag predecessor
set backlink to point to its predecessor
mark node
change pointer of its predecessor to point to its successor and remove flag
If a process wants to change a node that is already flagged by another operation, it first helps complete that operation and tries again.
Non-blocking Singly-Linked List
[Fomitchev and Ruppert 2004]

Amortized cost of operations: $\Theta(n + c)$

$n = \text{length of list}$
$c = \text{contention}$
Non-blocking Singly-Linked List using double-compare-and-swap

DCAS\( (P_1, P_2, \text{old}_1, \text{old}_2, \text{new}_1, \text{new}_2) \)
if \( P_1 = \text{old}_1 \) and \( P_2 = \text{old}_2 \)
then \( P_1 \leftarrow \text{new}_1 \)
\( P_2 \leftarrow \text{new}_2 \)
Non-blocking Singly-Linked List using double-compare-and-swap

Delete node B

Delete node C

Only one of these deletions will occur.
Non-blocking Singly-Linked List using double-compare-and-and-swap

DCAS(P₁, P₂, old₁, old₂, new₁, new₂)
if P₁ = old₁ and P₂ = old₂
then P₁ ← new₁
    P₂ ← new₂

Advantages:
• easy to use
• easy to prove resulting implementation is correct
Non-blocking Singly-Linked List using double-compare-and-swap

DCAS($P_1$, $P_2$, old$_1$, old$_2$, new$_1$, new$_2$)
if $P_1 = \text{old}_1$ and $P_2 = \text{old}_2$
then $P_1 \leftarrow \text{new}_1$
$P_2 \leftarrow \text{new}_2$

Problem: not available on existing machines
Non-blocking Singly-Linked List using double-compare-and-swap

DCAS(P₁, P₂, old₁, old₂, new₁, new₂)
if P₁ = old₁ and P₂ = old₂
then P₁ ← new₁
   P₂ ← new₂

Problem: not available on existing machines
Solution: implement it
Non-blocking Singly-Linked List using SCX [Brown, Ellen, Ruppert 2013]

- SCX($N_1, N_2, \text{new}_1$)
  - if $N_1$ and $N_2$ have not changed
  - then $P_1 \leftarrow \text{new}_1$
  - mark $N_2$

Problem: not available on existing machines
Solution: implement it
Non-blocking Binary Search Tree
[Fraser 2003]

Gave an implementation that used a powerful primitive involving 8 words spread over 5 nodes.
Non-blocking Balanced Binary Search Tree
[Brown, Ellen, Ruppert 2014]

Gave implementations using SCX.
Implementing Data Structures using more powerful primitives

- should be well-suited for implementing data structures
- should be efficient to implement
Transactional Memory

begin transaction
  if predecessor.pointer = node
  and node.pointer = successor
  then predecessor.pointer ← successor
end transaction

Each transaction eventually commits (and appears to take place atomically) or aborts (and appears to have no effect).
Transactional Memory

• Easy to use
• Software transactional memory has high overhead
• Hardware transactional memory does not provide progress guarantees
Transactional Memory

- Easy to use
- Software transactional memory has high overhead
- Hardware transactional memory does not provide progress guarantees

Use hardware transactional memory with a software fallback path.
Transactional Memory

- Easy to use
- Software transactional memory has high overhead
- Hardware transactional memory does not provide progress guarantees

Use hardware transactional memory with a non-transactional fallback path.
- imposes overhead on the transactions
Transactional Memory using 3 paths [Brown, 2015]

- hardware transactional fast and middle paths
- non-transactional fallback path does not impose overhead on the fast path
Transformations
[Timnat and Petrank, 2014]

• converts a class of non-blocking data structures into efficient wait-free data structures

• combines a non-blocking fast path and a wait-free slow path

• a process whose operation has failed F times on the fast path announces the operation in a wait-free queue and tries to help the operation at the front of the queue

• processes whose operations are on the fast path periodically help the operation at the front of the queue complete
Transformations

[Ellen, Luchangco, Moir and Shavit, 2005]

- converts any obstruction-free data structure into a wait-free data structure, provided there exists some (unknown) upper bound on the relative execution rates of any pair of processes
- if a process does not complete its operation within B steps, it sets a panic flag, which causes processes to wait for increasingly many steps, and eventually serializes attempts of operations.
Transformations [Giakkoupis, Helmi, Higham, and Woelfel, 2013]

- converts any obstruction-free data structure into a randomized wait-free data structure against an oblivious adversary
- if each operation completes when given $B$ consecutive steps, the expected step complexity is polynomial in $B$ and the number of processes
Relaxed Data Structures

Relax structural properties of a sequential data structure

- lazy deletion
- decouple rotations from insertions and deletions in balanced binary search trees
- break up operations into smaller pieces
Relaxed Data Structures

Relax structural properties of a sequential data structure

• lazy deletion
• decouple rotations from insertions and deletions in balanced binary trees
• break up operations into smaller pieces

Define new data structures with relaxed properties

• pool (bag) instead of a queue
• SNZI instead of a counter
Memory Reclamation

Easy for sequential and lock-based data structures: When a process *retires* a node (removes it from the data structure), it can *free* the node (reuse it or return it to the operating system).
Memory Reclamation: Garbage collection

Easy to use
Often implemented using locks
Can be inefficient
Memory Reclamation: Object pools

- retired nodes are put into a pool and are never freed
- nodes in a pool can be reused immediately
- nodes have version numbers

```
Delete(c)
```

![Diagram of object pool with nodes a, c, and d]
Memory Reclamation: Object pools

- retired nodes are put into a pool and are never freed
- nodes in a pool can be reused immediately
- nodes have version numbers

Insert(f)

Object pool
Memory Reclamation: Object pools

• to avoid contention, have separate object pools for each process

• when a pool is empty, some nodes can be allocated and added to the pool or taken from a shared pool

• when a pool belonging to a process has too many nodes, some can be given to the shared pool
Memory Reclamation: Hazard pointers (HPs)

- each process has $k$ hazard pointers
- must acquire a hazard pointer to a node before accessing it
- cannot acquire a hazard pointer to a node after it is retired
- can free a retired node once no hazard pointers points to it

• Example: Insert(e) by process 2
hazard pointers can be problematic when a retired node can point to another retired node

Example: Delete(i), Delete(o), Free(o) by process p
   Insert(x) by process q

If operations can follow pointers in retired nodes, then Free(o) is unsafe when a retired node points to o

Invalid pointer, so the system crashes!

Process p’s private memory

Process q’s private memory
Memory Reclamation: Hazard pointers (HPs)

• must acquire a hazard pointer to a node n before accessing it
• cannot acquire a hazard pointer to a node after it is retired
• cannot acquire a hazard pointer to a node reached by following a backlink

⇒ must maintain hazard pointers to all nodes it has encountered in the search
**Epoch-based Memory Reclamation** [Fraser 2004]

- The execution is divided into *epochs*, and the current epoch number is stored in shared memory.

- At the start of each operation, a process:
  - reads the current epoch and announces the value it read
  - checks whether all other processes have announced the current epoch and, if so, increments the current epoch

- When the current epoch is incremented, any objects retired two epochs ago can safely be freed

---

**Example execution**

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Process p</th>
<th>Process q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insert(a)</td>
<td>Insert(g)</td>
</tr>
<tr>
<td>1</td>
<td>Insert(b)</td>
<td>Insert(e)</td>
</tr>
<tr>
<td>2</td>
<td>Delete(b)</td>
<td>Insert(f)</td>
</tr>
<tr>
<td>2</td>
<td>Insert(f)</td>
<td>Delete(a)</td>
</tr>
<tr>
<td>3</td>
<td>Delete(f)</td>
<td>Insert(c)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Insert(b)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Insert(d)</td>
</tr>
</tbody>
</table>

- Retired Objects: b,1, a,2, f,3
- Current epoch: 4
- Announced epochs: 3, 3
Epoch-based memory reclamation

• Contention can be reduced by having a separate pool of retired nodes for each process.

• n reads, 1 write, and 1 compare-and-swap per operation, which can be amortized over multiple operations

• Restriction: a process cannot locally save pointers read during one operation and then access the nodes they point to in subsequent operations

• Not fault tolerant: no memory can be reclaimed after a process crash, a slow process can prevent an unbounded number of retired records from being freed.

• Using signaling, these problems can be avoided [Brown 2015].
HTM-based memory reclamation [AEHMS 2014]

- Run each operation inside a transaction
- The HTM system will abort a transaction if an object it accesses is freed during the transaction
- Can split transactions into small segments for efficiency
- Subtlety: must safely handle the case where an object is freed between two segments that access it

As with hazard pointers, problems can arise if operations can follow pointers in retired nodes.
Evaluation of Concurrent Data Structures

• new algorithmic techniques
• design of new data structures
• analysis of data structures
• experimental comparison of different data structures
• algorithmic engineering
Design and Analysis of Data Structures

Rigorous proofs are needed.
Proof sketches are insufficient!
Design and Analysis of Data Structures

Rigorous proofs are needed.
Proof sketches are insufficient!
Proof checking and verification tools are not yet sufficient.
Design and Analysis of Data Structures

Rigorous proofs are needed.
Proof sketches are insufficient!
Proof checking and verification tools are not yet sufficient.
Complexity analyses
Design and Analysis of Data Structures

- undefined notation and terminology
- same notation used for different things
- bad definitions: incomplete, imprecise, formal but incomprehensible
- uninitialized variables
- missing or imprecise specifications and invariants
- missing cases, missing steps in proofs
- incorrect linearizations or linearization points
- formulas applied when inapplicable
Design and Analysis of Data Structures

All of the ideas necessary for an expert to verify fully the central claims in the paper should be included, some of which may be placed in a clearly marked appendix that will be read at the discretion of the program committee. If desired, the authors can simply attach a copy of the full paper as the appendix.
Design and Analysis of Data Structures

Understandability is important
A high level description of the data structure is essential
-what are the different cases?
-what are the key ideas?
Commented pseudocode can help a reader resolve ambiguities, but does not replace clear descriptions of the operations.
Polish the data structure and its description.
Experimental Evaluation of Concurrent Data Structures

Experiments should be informative

• comprehensive experiments that fairly evaluate the relative performance of concurrent data structures under various conditions
Experimental Evaluation of Concurrent Data Structures

Experiments should be informative

• comprehensive experiments that fairly evaluate the relative performance of concurrent data structures under various conditions

• provide some understanding about why certain data structures perform well or poorly
Experimental Evaluation of Concurrent Data Structures

Experiments should be informative

• comprehensive experiments that fairly evaluate the relative performance of concurrent data structures under various conditions

• provide some understanding about why certain data structures perform well or poorly

• understand how changes to a data structure affect its performance
Experimental Evaluation of Concurrent Data Structures

Experiments should be informative

• comprehensive experiments that fairly evaluate the relative performance of concurrent data structures under various conditions

• provide some understanding about why certain data structures perform well or poorly

• understand how changes to a data structure affect its performance

• compare concurrent data structures on real-world data to determine which is best for a particular application
Experimental Evaluation of Concurrent Data Structures

Experiments should be reproducible

• clear description of the computational platform and testing framework

• code should be made available, if possible

• artifact evaluation/replicated computational results
Use the same setup to compare different data structures

Perform the same experiments using the same:

• hardware
• operating system
• testing setup
• memory reclamation scheme
Overhead in experimental setup can mask performance differences

Operation A takes 1 microsecond
Operation B takes 500 microseconds
Overhead is 500 microseconds per operation

\[
\frac{\text{operation B + overhead}}{\text{operation A + overhead}} = \frac{1000}{501}
\]

Experiments show operation B seems to be only 2x slower than operation A rather than 500x slower.
Experimental Set-up Matters

Experiment

• 2 socket Intel machine with 48 hardware processors running Linux,
• Fixed implementation of a Binary Search Tree
• Prefill the tree so it contains 5000 different keys in [1..10,000]
• For each process, for 1 second, repeatedly:
  - pick a key uniformly at random in the range [1..10,000]
  - decide whether to insert it or delete it, each with 50% probability
• Count the number of operations completed (using 1 to 48 processes)
Parameters

• pinning: yes or no
• memory allocator: jemalloc or malloc
• memory reclamation: yes or no
• object pools: yes or no
Memory allocation

![Graph showing memory allocation over the number of processes. The graph compares 'jemalloc' and 'malloc' with 'jemalloc' performing better at higher numbers of processes.]
pinning
Effect of memory reclamation

![Bar chart showing operations per microsecond for AVL and Chromatic across 1GB, 2GB, 3GB, and 4GB]
Overhead of Timing
Uninteresting Paper

Slightly new concurrent data structure,
experiments on which it performs better than some other existing data structures
“Impractical” Paper

concurrent data structure based on a new idea
experiments on which it performs slightly worse than or comparable to existing data structures
Experimental Evaluation of Concurrent Data Structures

Don’t just present the experiments in which your concurrent data structure performs best and suppress the experiments where it performs worse than other data structures.

If experiments show that a new concurrent data structure doesn’t always significantly outperform all existing data structures, don’t necessarily reject the paper.
Experimental Evaluation of Concurrent Data Structures

All of the ideas necessary for an expert to verify fully the central claims in the paper should be included, some of which may be placed in a clearly marked appendix that will read at the discretion of the program committee. If desired, the authors can simply attach a copy of the full paper as the appendix.
Future Directions for Research in Concurrent Data Structures

• Understanding the effects of different design decisions on performance
• Better memory reclamation techniques for use with lock-free and wait-free data structures
• Better tools for proof checking and verifying concurrent data structures
• Identify useful relaxed data structures
• NUMA aware concurrent data structures
• Concurrent data structures for weak memory consistency models
• Amortized analyses of non-blocking data structures
• Lower bounds on complexity