Multi-Version Concurrent Data Structures

PANAGIOTA FATOUROU
University of Crete, Department of Computer Science
Foundation for Research and Technology - Hellas

School on the Practice and Theory of Distributed Computing, November 2023
A multiversion object maintains its previous versions, so threads can have access to the history of the object (i.e., to its previous values).
Multiversioning

- Multiversioning is widely used: Database systems

- Software Transactional Memory
  [Fernandes et al. PPoPP’11] [Lu et al. DISC’13]

- Concurrent data structures
  [Fatourou et al. SPAA’19] [Wei et al. PPoPP’21]
  [Kobus et al. PPoPP’22] [Sheffi et al. OPODIS’22]
Why Multiversioning?

Many applications require querying large portions or multiple parts of the data structure.

Big-data applications use shared in-memory tree-based data indices
- Fast data retrieval
- Useful data analytics
Why multivesion Concurrent Data Structures?

Concurrent Data Structure built out of CAS objects

Take a snapshot of the data structure

Answer multi-point Queries using snapshot

Snapshot: Saves a read-only version of the state of the data structure at a single point in time. [An atomic view of the state of the data structure.]

The vCAS technique

- Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun: Constant-Time Snapshots with Applications to Concurrent Data Structures, PPoPP 2021.
Background Knowledge
Model

- The system is asynchronous.
- Threads communicate by accessing shared variables.
- In addition to Read and Write, a thread may execute an atomic CAS instruction on a shared variable.
- Threads may fail by crashing.

```java
ATOMIC boolean Compare&Swap(
    Variable V, Value v_old, Value v_new) {
    if (V == v_old) { V = v_new; return TRUE; }
    return FALSE;
}
```
Correctness [Herlihy & Wing]

Linearizability

In every execution $\alpha$, each operation should have the same response as if it has executed serially (or atomically) at some point in its execution interval. This point is called linearization point of the operation.
Linearizability: Queue supporting ReadAll()

- Enq(1) ok
- Enq(2)
- Deq()
- res = 1
- Enq(3) ok
- ReadAll()
- ok() -> {1,2}

Queue: {1} {1, 2} {2} {2, 3}
Linearizability: Queue supporting `ReadAll()`

Example of non-linearizable execution

```
Enq(1) ok  Enq(2) ok  Deq()  res= 1  Enq(3) ok
{1}       {1, 2}      {2}          {2, 3}

Queue: ReadAll()
{1}  ?*X  ?*X  ?*X  {1, 2, 3}
```
Linearizability: Queue supporting ReadAll()

Example of non-linearizable execution

Set ReadAll()
{
    // sequential alg
    Node *q = Head;
    Set res;
    while (q != NULL) {
        res = res ∪ {q->data};
        q = q->next;
    }
    return res;
}
Linearizability: Queue supporting **ReadAll()**

Example of non-linearizable execution

```
Set ReadAll() {
    Node *q = Head;
    while (q != NULL) {
        res = res \cup \{q->data\};
        q = q->next;
    }
    return res;
}
```

```
Enq(1)  ok  Enq(2)  ok  Deq()  res = 1  Enq(4)  ok

ReadAll()  X  X  X  \{1,2,3\}
```

Set ReadAll() {
    Node *q = Head;
    while (q != NULL) {
        res = res \cup \{q->data\};
        q = q->next;
    }
    return res;
}
Progress

Non-blocking Algorithms

Wait-Freedom
Every thread finishes the execution of its operation within a finite number of steps.

Lock-Freedom
Some thread finishes the execution of its operation within a finite number of steps.
An Example of a Concurrent Queue Implementation
Michael & Scott Queue as an Example

```c
struct node {
    T value;        // immutable
    CAS Object next : struct node *
};

CAS objects Head, Tail: struct node *;  // initially, both point to a dummy node
```
Michael & Scott Queue as an Example

```c
struct node {
    T value;  // mutable
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *;  // initially, both point to a dummy node
```

Thread 1
Enqueue(1)
Read
Read

Tail

Head

Dummy
Michael & Scott Queue as an Example

Thread 1
Enqueue(1)

struct node {
    T value;       // immutable
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *;  // initially, both point to a dummy node

CAS

Thread 1
Enqueue(1)

Dummy

1

Head

Tail
Michael & Scott Queue as an Example

Thread 1
Enqueue(1)

struct node {
    T value;  // immutable
    CAS Object next: struct node *;
}

CAS objects Head, Tail: struct node *;  // initially, both point to a dummy node
Michael & Scott Queue as an Example

struct node {
    T value; // mutable
    CAS Object next: struct node *;
};

CAS objects Head, Tail: struct node *;  // initially, both point to a dummy node
Michael & Scott Queue as an Example

```c
struct node {
    T value; // immutable
    CAS Object next : struct node *;
};

CAS objects Head, Tail: struct node *; // initially, both point to a dummy node
```
Michael & Scott Queue as an Example

Thread 1
Enqueue(1)

Thread 2
Enqueue(2)

CAS ✓

CAS x
Michael & Scott Queue as an Example

Thread 1

Thread 2

CAS

Dummy

Head

1

2
Michael & Scott Queue as an Example
Michael & Scott Queue as an Example

[Diagram of a queue with a dummy node, head, tail, and a thread labeled 'Thread 2']
Michael & Scott Queue as an Example

Thread 1

CAS

Dummy

1

2

Head

Tail
Michael & Scott Queue as an Example

![Queue Diagram]

- **Head**: Dummy
- **Tail**: 2
- Elements: 1

Diagram shows the structure of a queue with elements connected from Dummy to 1 and then to 2, indicating the sequence in which elements are processed.
Michael & Scott Queue as an Example

Thread 1 is slow
Michael & Scott Queue as an Example

Thread 1 is slow
Michael & Scott Queue as an Example

Thread 1 is slow

CAS

Thread 2

Dummy

Head

1

2
Michael & Scott Queue as an Example

Thread 1 is slow

Thread 2

CAS
Michael & Scott Queue as an Example

![Queue Diagram](image)
Michael & Scott Queue as an Example

stores 1 as its return value
Michael & Scott Queue as an Example
Michael & Scott Queue as an Example

![Diagram of a queue with a head and tail, representing the concept of Michael & Scott Queue as an example.](image-url)
vCAS Technique

Simple, General, Efficient!

Lock-free Concurrent Data Structure

Take a snapshot of the Data Structure

Answer multi-point Queries using snapshot

Lock-free Queue [Michael & Scott’96]
Enqueue, Dequeue

Lock-free Snapshottable Queue
Enqueue, Dequeue

Preserves parallelism and time bounds

Snapshot
- Range Query
- i-th element
- All elements

O(1) time, a single CAS
Wait-free, Linearizable

Works with many lock-free data structures, including:
- BST [Ellen,Fatourou, Ruppert, Breugel’10]
- Linked List [Harris’01]
- Chromatic Tree [BrownEllenRuppert’14]
- …
Overview of the VCAS Approach

- **CAS Object**
  - Supports:
    - Read
    - CAS

- **Versioned CAS (vCAS) Object**
  - Supports:
    - vRead
    - vCAS
    - readVersion

**Time Complexity:**
- vRead(X)
- vCAS(X, old, new)
- takeSnapshot()
- readVersion(X, S)

O(1) time, small constant
wait-free

**Camera Object**
- Supports:
  - TakeSnapshot

Makes it possible for a thread to later read only the memory locations it needs from shared memory, knowing that all such reads will be atomic.
Supporting Multi-Point Queries

• Each query calls TakeSnapshot to get a timestamp.
• More than one queries may have the same timestamp.
• Each query attempts to atomically increment ts using CAS.
• Each version of a vCAS object has a timestamp, which has been read from ts.
VCAS objects are represented internally using version lists. The fields of a Vnode (i.e., a node of a version list) are:

- `val`
- `ts`
- `vnext`
Versioned CAS Implementation

VCAS Object

previous values of next field of node
Versioned CAS Implementation

- **vCAS(X, old, new)**
  - Link in a new node with timestamp TBD
  - Update its timestamp

- **vRead(X)**
  - Help update timestamp of most recent version
  - Return its value

- **readVersion(X, t)**
  - Help update timestamp
  - Find newest version with timestamp $\leq t$

- **takeSnapshot()**
  - Attempt to increment ts using CAS
  - Return its previous value

- VCAS Object
Overview of the VCAS Approach: Michael & Scott Queue as an Example

```
struct node {
  T value;
  CAS Object next: struct node *;
};

struct node {
  T value;
  vCAS Object next: struct node *;
};

vRead(X)
- Help update timestamp of most recent version of X
- Return current value of X
```
Overview of the VCAS Approach: Michael & Scott Queue as an Example

```
struct node {
    T value;

    struct node *next;
} CAS objects Head, Tail;
```

```
vRead(X)
- Help update timestamp of most recent version of X
- Return current value of X
```

```
struct node {
    T value;

    vCAS Object next : struct node *;
} vCAS objects Head, Tail;
```
Overview of the VCAS Approach: Michael & Scott Queue as an Example

struct node {
    T value;
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *;

struct node {
    T value;
    vCAS Object next : struct node *;
}

vCAS objects Head, Tail: struct node *;
Overview of the VCAS Approach: Michael & Scott Queue as an Example

struct node {
    T value;
    // CAS Object
    next : struct node *;
}

CAS objects Head, Tail: struct node *;

struct node {
    T value;
    // vCAS Object
    vCAS next : struct node *;
}
vCAS objects Head, Tail: struct node *;

vCAS(X, old, new)
- Malloc() a new vNode with timestamp TBD (-1)
- Link it in the version list of the vCAS object
- Update its timestamp

Thread 1
Enqueue(1)

vCAS

Head

nextv 0 val

Tail

nextv 0 val

Dummy

nextv 0 val

nextv -1 val

nextv 0 val

Thread 1
Enqueue(1)
Overview of the VCAS Approach: Michael & Scott Queue as an Example

```
struct node {
    T value;
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *;
```

```
struct node {
    T value;
    vCAS Object next : struct node *;
}

vCAS objects Head, Tail: struct node *;
```

```
vCAS(X, old, new)
    • Malloc() and link in a new vNode with timestamp TBD (-1)
    • Make it the first node in the vlist of vCAS object
    • Update its timestamp
```
Overview of the VCAS Approach: Michael & Scott Queue as an Example

**Dummy**

**Tail**

Thread 1

vCAS

```
nextv 0 val
```
Overview of the VCAS Approach: Michael & Scott Queue as an Example
Overview of the VCAS Approach: Michael & Scott Queue as an Example
Overview of the VCAS Approach: Michael & Scott Queue as an Example
Multi-Point Queries: Michael & Scott Queue as an Example

Thread 3
Enqueue(3)

ReadAll()
Takesnapshot → 0

Thread 4
deq()
Multi-Point Queries: Michael & Scott Queue as an Example
Multi-Point Queries: Michael & Scott Queue as an Example

- Executes Sequential Code, but...

```c
Set ReadAll() {
    Node *q = Head;
    while (q != NULL) {
        res = res ∪ \{q->data\};
        q = q->next;
    }
    return res;
}
```
Linearizability: Queue supporting ReadAll()

Example of non-linearizable execution

Set ReadAll() {
    Node *q = Head;
    while (q != NULL) {
        res = res ∪ {q->data};
        q = q->next;
    }
    return res;
}
Multi-Point Queries: Michael & Scott Queue as an Example

**ReadAll()**

- Executes Sequential Code for ReadAll() but...
- Uses `readVersion(0)` to read the values of vCAS objects as it goes

**readVersion(X, t)**

- Help update timestamp
- Find newest version with time ≤ t

It returns {1,2}
void enq(T value) {
    NODE *next, *last;
    NODE *p = newcell(NODE);
    // p->value = value; p->next = NULL;
    while (TRUE) {
        last = Tail;
        next = last->next;
        if (last != Tail) continue;
        if (next != NULL) {
            CAS(Tail, last, next);
            continue;
        }
        if (CAS(last->next, NULL, p)) break;
    }
    CAS(Tail, last, p);
}

void enq(T value) {
    NODE *next, *last;
    NODE *p = new(NODE, value, NULL);
    while (TRUE) {
        last = vRead(Tail);
        next = vRead(last->next);
        if (last != vRead(Tail)) continue;
        if (next != NULL) {
            vCAS(Tail, last, next);
            continue;
        }
        if (vCAS(last->next, NULL, p)) break;
    }
    vCAS(Tail, last, p);
}
Versioned CAS on BSTs

Examples of queries
- Range queries
- Tree height
- Smallest key that matches a condition
- K-successors
- Multi-lookup

Snapshottable BST

Expanding out VCAS objects
Comparison with Existing Techniques

Efficiency

- LFCA [Winblad et al., SPAA’18]
- KiWi [Basil et al., PPoPP’17]
- PNB-BST [Fatourou et al., SPPA’19]
- SnapTree [Bronson et al., PPoPP’10]

Generality

- Epoch RQs [Arbel, Raviv et al., PPoPP’18]
- SnapCollector [Petrak et al., PPoPP’13]

The VCAS Approach, Wei et al.

STM [Fernandez et al., PPoPP’11]
Practical Optimizations

- Avoiding Indirection
- Using exponential backoff to reduce contention when accessing the global timestamp
- Removing redundant versions from the version list
- Garbage collecting old versions
Avoiding Indirection

Snapshottable BST

Expanding out VCAS objects

Without indirection

Merge version list and data structure nodes
Experimental Evaluation

- Adding support for multi-point queries on top of existing concurrent lock-free data structures was very easy and required adding fewer than 150 lines of code (in C++).
- The vCAS approach adds very little overhead to the original data structure.
- The vCAS approach (which is general-purpose) is often as fast as, or faster than, state-of-the-art lock-free data structures supporting range queries.
Summary of vCAS Technique

- vCAS is an approach for adding snapshotting and multi-point queries to existing concurrent data structures
  - **Easy-to-use**: simply replace CAS with Versioned CAS
  - **Efficient**: both theoretically and practically
  - **General**: supports a wide range of data structures and multi-point queries

- Code is available on GitHub: [https://github.com/yuanhaow/vcaslib](https://github.com/yuanhaow/vcaslib)

Multi-Version Garbage Collection

ANY SYSTEM THAT MAINTAINS MULTIPLE VERSIONS OF EACH OBJECT NEEDS A WAY OF EFFICIENTLY RECLAIMING THEM!
Research Question

How do we garbage collect, efficiently, for multiversion data structures?
A general Multiversion Garbage Collection (GC) scheme with the following properties:

- **Progress:** wait-free
- **Time:** $O(1)$ per reclaimed version, on average
- **Space:** constant factor more versions than needed, plus an additive term

Previous solutions either use:

- **unbounded space** [Fernandes et al., PPoPP’11], or
- **$O(P)$** time per reclaimed version [Lu et al. DISC’13] [Böttcher et al., VLDB’19]
- $P$: number of processes
Multiversion Garbage Collection (MVGC)

Maintaining all old versions ⇒ high memory usage

How do we identify which versions are not needed?
How do we safely reclaim them?
Which Versions are Needed?

Versions

- Versions needed by read-only operations
- Most recent versions needed

Timestamps of multipoint queries

Objects

- X
- Y

Time

- Which Versions are Needed?
- Versions
- Versions needed by read-only operations
- Most recent versions needed
- Timestamps of multipoint queries
- Objects
- X
- Y
- Time
Related Work – Epoch-Based Solutions

- Reclaim versions overwritten before the start of the oldest read-only operation

- Safe to collect

- Operations started before this point have completed

X
Related Work – Epoch-Based Solutions

Cons: High space usage
- Unable to collect newer obsolete versions
- Particularly bad with long read-only operations
  - E.g. database scans, large range queries
- Paused process can lead to unbounded space usage

Pros: Fast, easy to implement
Related Work – Other Solutions

Techniques have been developed to address shortcomings of epoch-based solutions.

- GMV [Lu et al. DISC’13], Hana [Lee et al. SIGMOD’16], Steam [Böttcher et al. VLDB’19]
- Require $\Omega(P)$ time, on average, to collect each version in worst case executions.
  - $P$: number of processes
- Keep up to $P$ times more versions than necessary
What is the problem to solve?

Step 1: Identify obsolete versions

Step 2: Unlink from version list

Step 3: Reclaim memory of unlinked versions
Step 1: Identify obsolete versions

Unneeded Versions

Active multi-point queries

Range Tracker

O(1) amortized time
O(needed) space

Obsolete Versions

Range Tracker

X

Y

Unneeded Versions

Active multi-point queries

Obsoleted Versions

[Diagram showing the process of identifying obsolete versions]
Step 2: Unlink from version list

We present a wait-free, amortized $O(1)$ algorithm for remove()
Step 3: Reclaim memory of unlinked versions

- n is not safe to reclaim right away because a thread (P1) could be paused to access it
- Using Hazard Pointers (HP) or Concurrent Reference Counting (CRC) would solve this problem, but
  - HP sacrifices wait-freedom
  - CRC sacrifices space bounds
- Ben-David et al. presents a new safe reclamation scheme specifically for the doubly-linked version list implementation it provides
Overall Results

Time bounds:
- $O(1)$ time, on average, to identify, remove, and reclaim a version
- Wait-free

Space bounds:
- Number of unreclaimed versions $\in O(\# \text{ required versions}) + \text{additive term}$

Full version (with proof of correctness) available on arxiv:
https://arxiv.org/abs/2108.02775
New MVGC Schemes
[Wei, Blelloch, Fatourou, Ruppert, PPoPP 2023]

Use range tracker to get good space efficiency

Time efficiency: BBF+ is over optimized for worst-case

Concurrent remove()s

- **DL-RT**: Range tracker + new doubly-linked version list
- **SL-RT**: Range tracker + new singly-linked version list
Results

Two new MVGC schemes:

◦ Fast and space efficient in practice
◦ Strong space bounds in theory

Full paper (with proofs of correctness) is available on arxiv:
https://arxiv.org/abs/2212.13557

Code is available on GitHub:
https://github.com/cmuparlay/ppopp23-mvgc
Conclusions

The vCAS Approach

- Simple, constant-time approach to take a snapshot of a collection of CAS objects.
- Technique to use snapshots to implement linearizable multi-point queries in many lock-free data structures.
- Adding snapshots to a CAS-based data structure preserves the data structures’ asymptotic time bounds.
- Every read is completed within a finite number of instructions (i.e. it is wait-free).
Conclusions

- We present theoretically efficient solutions to the MVGC problem
- Developed new techniques for all 3 steps:
  1. Identify obsolete versions
  2. Unlink from version list
  3. Reclaim memory of unlinked versions

- The MVGC schemes:
  - Provide strong space and time bounds in theory.
  - Space and time efficient in practice.
This work was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the “Second Call for HFRI Research Projects to support Faculty Members and Researchers” (project number: 3684, project acronym: PERSIST).

Thank You!

QUESTIONS?

faturu@csd.uoc.gr

www.ics.forth.gr/~faturu/
We are recruiting!