Efficient Multi-word Compare and Swap

CS6868:Concurrent Programming – Course Project

Albin James Maliakal, Kunal Sanjiv Umaji, Sanjeev Reddy
CS25S027, CS25S020, CS25S013

Indian Institute of Technology Madras

April 30, 2026

Instructor: Prof. KC Sivaramakrishnan

Overview

1. What is MCAS?
   - Why Should We Care?
   - MCAS – The Formal Bit
2. The Impossibility Result
3. How the k+1 CAS Algorithm Works
   - Two Phases, One CAS to Rule Them All
   - But Wait – How Do Reads Work?
   - The Scoreboard
   - Recap
4. Detailed Algorithm & Persistent Memory
   - Listing 1: Data Structures
   - Listing 2: `readInternal`
   - Listing 3: Main Algorithm
5. Experimental Evaluation & Conclusion
   - Performance Overview
   - Doubly Linked List Benchmark

Overview

1. What is MCAS?
   - Why Should We Care?
   - MCAS – The Formal Bit
2. The Impossibility Result
3. How the k+1 CAS Algorithm Works
4. Detailed Algorithm & Persistent Memory
5. Experimental Evaluation & Conclusion
6. Research Question & Conclusion

What is MCAS? – From One Word to Many

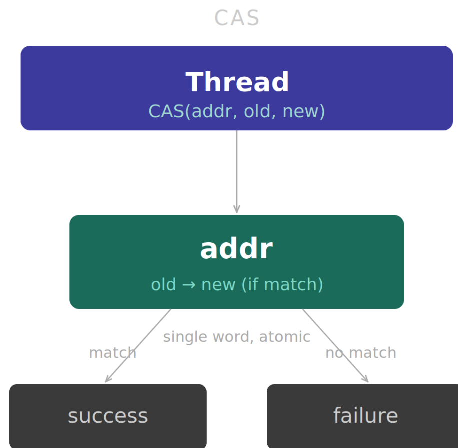
Compare-and-Swap

CAS atomically swaps **one** memory word.

`CAS(addr, old, new)`

Single word, atomic

That's all hardware gives us!



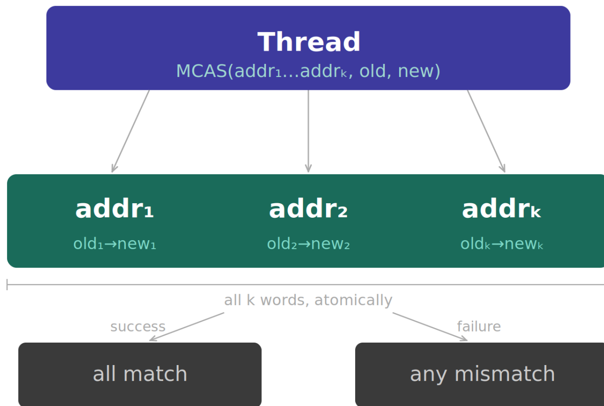
What is MCAS? – From One Word to Many

Multi-word CAS (Guerraoui et al., 2020)

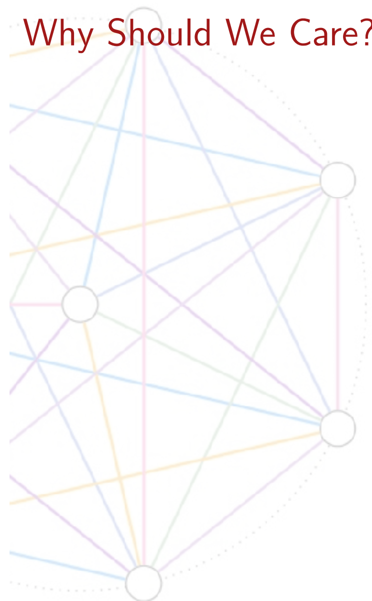
MCAS does the same for k words at once!

`MCAS(addr`$_1$`..addr`$_k$`, old`$_1$`..old`$_k$`, new`$_1$`..new`$_k$`)`

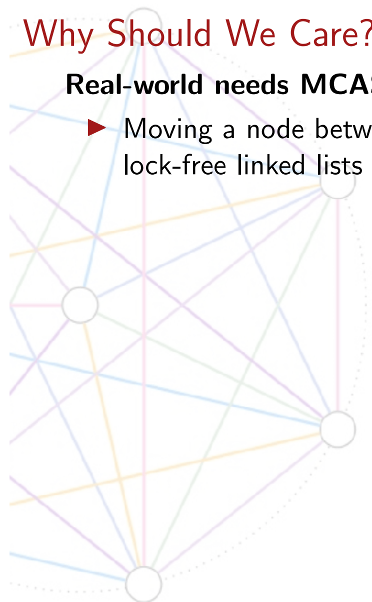
k **words, still atomic!**



Why Should We Care?



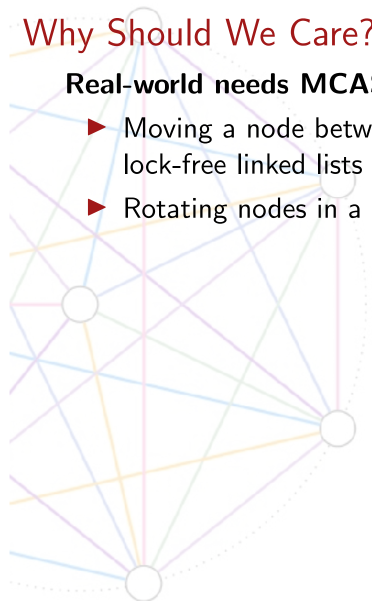
Why Should We Care?



Real-world needs MCAS:

- Moving a node between two lock-free linked lists

Why Should We Care?



Real-world needs MCAS:

- Moving a node between two lock-free linked lists
- Rotating nodes in a concurrent BST

Why Should We Care?

Real-world needs MCAS:

- Moving a node between two lock-free linked lists
- Rotating nodes in a concurrent BST
- Updating multiple fields of a persistent record atomically (Zuriel et al., 2019)

Why Should We Care?

Real-world needs MCAS:

- Moving a node between two lock-free linked lists
- Rotating nodes in a concurrent BST
- Updating multiple fields of a persistent record atomically (Zuriel et al., 2019)

Locks? Deadlocks!

STM? Too slow!

MCAS! Just right.

Why Should We Care?

Real-world needs MCAS:

- Moving a node between two lock-free linked lists
- Rotating nodes in a concurrent BST
- Updating multiple fields of a persistent record atomically (Zuriel et al., 2019)

Locks? Deadlocks!

STM? Too slow!

MCAS! Just right.

Fun fact #1

Hardware only gives us single-word CAS. Every MCAS is built *in software* on top of it!

Why Should We Care?

Real-world needs MCAS:

- Moving a node between two lock-free linked lists
- Rotating nodes in a concurrent BST
- Updating multiple fields of a persistent record atomically (Zuriel et al., 2019)

Locks? Deadlocks!

STM? Too slow!

MCAS! Just right.

Fun fact #1

Hardware only gives us single-word CAS. Every MCAS is built *in software* on top of it!

Fun fact #2

The best prior algorithm (Harris et al., 2002) needed $\geq 2k+1$ CAS operations.

Why Should We Care?

Real-world needs MCAS:

- Moving a node between two lock-free linked lists
- Rotating nodes in a concurrent BST
- Updating multiple fields of a persistent record atomically (Zuriel et al., 2019)

Locks? Deadlocks!

STM? Too slow!

MCAS! Just right.

Fun fact #1

Hardware only gives us single-word CAS. Every MCAS is built *in software* on top of it!

Fun fact #2

The best prior algorithm (Harris et al., 2002) needed $\geq 2k+1$ CAS operations.

This paper? **Just k+1.**

Nearly half the work. Mic drop.

MCAS – The Formal Bit

Theorem 1.1: Multi-word Compare-and-Swap (MCAS) (Guerraoui et al., 2020)

- **Input:** Array of k tuples $\langle \text{addr}_i, \text{old}_i, \text{new}_i \rangle$
- **Atomically:**
  - If $\forall i : *\texttt{addr}_i = \text{old}_i \Rightarrow$ write all new values, return $\textsc{Success}$
  - Otherwise $\Rightarrow$ change nothing, return $\textsc{Failure}$

MCAS – The Formal Bit

Theorem 1.2: Multi-word Compare-and-Swap (MCAS) (Guerraoui et al., 2020)

- **Input:** Array of k tuples $\langle \text{addr}_i, \text{old}_i, \text{new}_i \rangle$
- **Atomically:**
  - If $\forall i : *\texttt{addr}_i = \text{old}_i \Rightarrow$ write all new values, return SUCCESS
  - Otherwise $\Rightarrow$ change nothing, return FAILURE

Linearizable (Herlihy and Wing, 1990)
Each op looks instantaneous

Lock-free (Herlihy, 1991)
Progress even if threads crash

MCAS – The Formal Bit

Theorem 1.3: Multi-word Compare-and-Swap (MCAS) (Guerraoui et al., 2020)

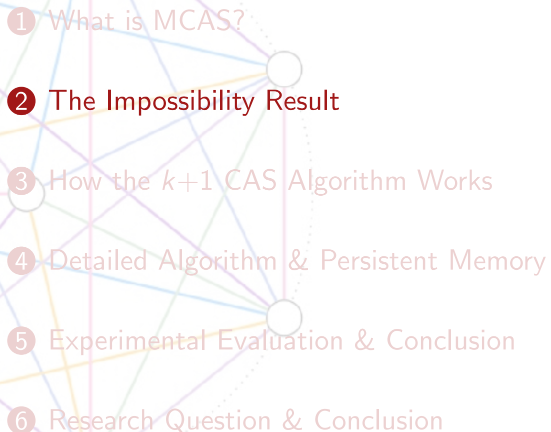
- **Input:** Array of k tuples $\langle \text{addr}_i, \text{old}_i, \text{new}_i \rangle$
- **Atomically:**
  - If $\forall i : *\texttt{addr}_i = \text{old}_i \Rightarrow$ write all new values, return SUCCESS
  - Otherwise $\Rightarrow$ change nothing, return FAILURE

Linearizable (Herlihy and Wing, 1990)
Each op looks instantaneous

Lock-free (Herlihy, 1991)
Progress even if threads crash

Addresses sorted in a total order (prevents deadlocks). Each operation uses a **descriptor** to track its state.

Overview



1 What is MCAS?

2 The Impossibility Result

3 How the k+1 CAS Algorithm Works

4 Detailed Algorithm & Persistent Memory

5 Experimental Evaluation & Conclusion

6 Research Question & Conclusion

The Impossibility Result – You Can't Cheat Physics

Theorem (Guerraoui et al.):

Any lock-free, DAP k-CAS needs $\geq k$ CAS operations.

The Impossibility Result – You Can't Cheat Physics

Theorem (Guerraoui et al.):

Any lock-free, DAP k-CAS needs $\geq k$ CAS operations.

Intuition: build a "star" of $k+1$ conflicting calls – c_0 shares a target with each c_i, but the c_i's are pairwise disjoint.

The Impossibility Result – You Can't Cheat Physics

Theorem (Guerraoui et al.):

Any lock-free, DAP k-CAS needs $\geq k$ CAS operations.

Intuition: build a "star" of $k+1$ conflicting calls – c_0 shares a target with each c_i, but the c_i's are pairwise disjoint.

By pigeonhole, c_0 *must* CAS at least k distinct locations.

The Impossibility Result – You Can't Cheat Physics

Theorem (Guerraoui et al.):
Any lock-free, DAP k-CAS needs $\geq k$ CAS operations.

Intuition: build a "star" of $k+1$ conflicting calls – c_0 shares a target with each c_i, but the c_i's are pairwise disjoint.

By pigeonhole, c_0 *must* CAS at least k distinct locations.

Punchline: k is the floor. This paper hits $k+1$. **Gap of just 1!**

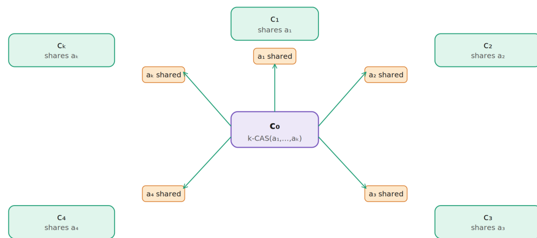
The Impossibility Result – You Can't Cheat Physics

Theorem (Guerraoui et al.):
Any lock-free, DAP k-CAS needs $\geq k$ CAS operations.

Intuition: build a "star" of k+1 conflicting calls – c_0 shares a target with each c_i, but the c_i's are pairwise disjoint.

By pigeonhole, c_0 *must* CAS at least k distinct locations.

Punchline: k is the floor. This paper hits k+1. **Gap of just 1!**

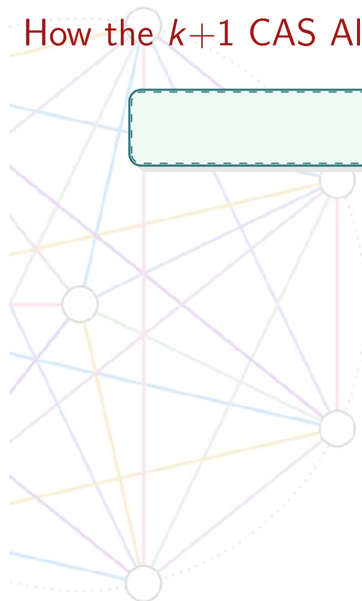


Star configuration: c_0 is the hub, $c_1 \ldots c_k$ are the spokes.

Overview

1. What is MCAS?
2. The Impossibility Result
3. How the $k+1$ CAS Algorithm Works
   - Two Phases, One CAS to Rule Them All
   - But Wait – How Do Reads Work?
   - The Scoreboard
   - Recap
4. Detailed Algorithm & Persistent Memory
5. Experimental Evaluation & Conclusion

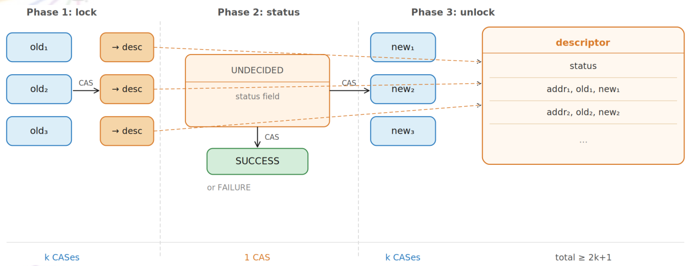
How the k+1 CAS Algorithm Works – The Big Idea



What if we just... didn't unlock?

How the $k+1$ CAS Algorithm Works – The Big Idea

What if we just... didn't unlock?



How the $k+1$ CAS Algorithm Works – The Big Idea

What if we just... didn't unlock?

Phase 1: lock
old$_1$ → desc
old$_2$ CAS → desc
old$_3$ → desc
k CASes

Phase 2: status
UNDECIDED
status field
CAS
SUCCESS
or FAILURE
1 CAS

Phase 3: unlock
CAS
new$_1$
new$_2$
new$_3$
k CASes

descriptor
status
addr$_1$, old$_1$, new$_1$
addr$_2$, old$_2$, new$_2$
...
total ≥ 2k+1

Prior work: $2k+1$ CAS

lock(k) + status(1) + unlock(k)

This paper: $k+1$ CAS

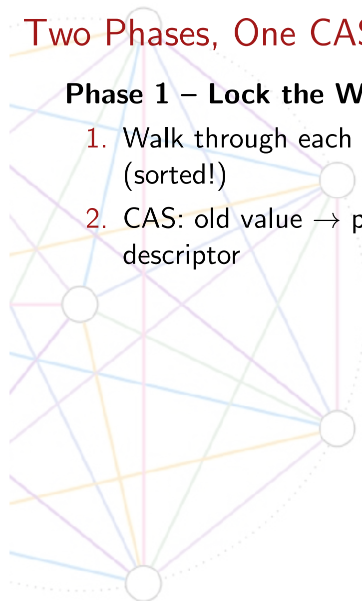
lock(k) + status(1) + ~~unlock~~

Descriptors cleaned up lazily!

Two Phases, One CAS to Rule Them All

Phase 1 – Lock the Words

Two Phases, One CAS to Rule Them All



Phase 1 – Lock the Words

1. Walk through each target address (sorted!)
2. CAS: old value $\rightarrow$ pointer to our descriptor

Two Phases, One CAS to Rule Them All

Phase 1 – Lock the Words

1. Walk through each target address (sorted!)
2. CAS: old value $\rightarrow$ pointer to our descriptor
3. All k locked? Move to Phase 2
4. Mismatch? Abort immediately

Two Phases, One CAS to Rule Them All

Phase 1 – Lock the Words

1. Walk through each target address (sorted!)
2. CAS: old value → pointer to our descriptor
3. All k locked? Move to Phase 2
4. Mismatch? Abort immediately

Phase 2 – Flip the Switch

1. One single CAS on the **status word**:
   UNDECIDED → SUCCESS *or* FAILURE

Two Phases, One CAS to Rule Them All

Phase 1 – Lock the Words

1. Walk through each target address (sorted!)
2. CAS: old value → pointer to our descriptor
3. All k locked? Move to Phase 2
4. Mismatch? Abort immediately

Phase 2 – Flip the Switch

1. One single CAS on the **status word**:
   UNDECIDED → SUCCESS *or* FAILURE
2. That's it. Operation is **done**.
   This CAS is the linearization point!

Two Phases, One CAS to Rule Them All

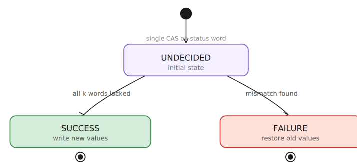
Phase 1 – Lock the Words

1. Walk through each target address (sorted!)
2. CAS: old value → pointer to our descriptor
3. All k locked? Move to Phase 2
4. Mismatch? Abort immediately

Phase 2 – Flip the Switch

1. One single CAS on the **status word**:
   UNDECIDED → SUCCESS *or* FAILURE
2. That's it. Operation is **done**.

   This CAS is the linearization point!



Two Phases, One CAS to Rule Them All

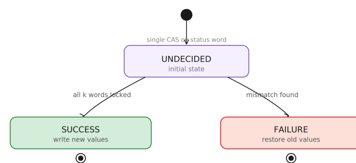
Phase 1 – Lock the Words

1. Walk through each target address (sorted!)
2. CAS: old value → pointer to our descriptor
3. All k locked? Move to Phase 2
4. Mismatch? Abort immediately

Phase 2 – Flip the Switch

1. One single CAS on the **status word**: UNDECIDED → SUCCESS *or* FAILURE
2. That's it. Operation is **done**.

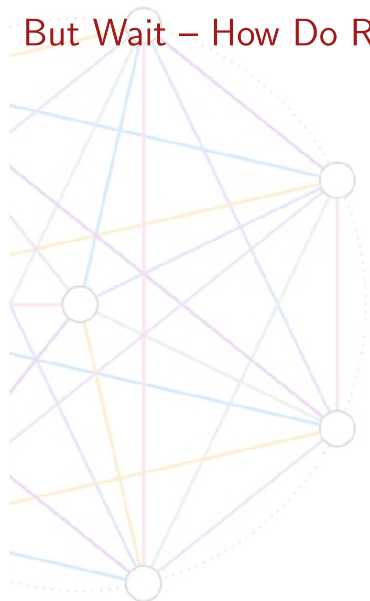
This CAS is the linearization point!



No Phase 3!

Old algorithms unlock each word (k more CAS ops). This paper leaves descriptors in place and cleans up later via epoch-based garbage collection.

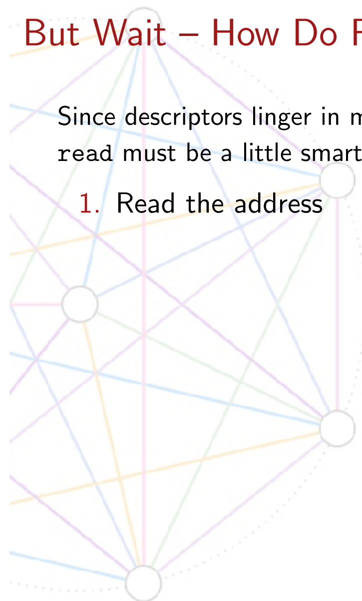
But Wait – How Do Reads Work?



But Wait – How Do Reads Work?

Since descriptors linger in memory cells, a `read` must be a little smarter:

But Wait – How Do Reads Work?



Since descriptors linger in memory cells, a `read` must be a little smarter:

1. Read the address

But Wait – How Do Reads Work?

Since descriptors linger in memory cells, a `read` must be a little smarter:

1. Read the address
2. **Plain value?**
   $\rightarrow$ Return it! (fast path)

But Wait – How Do Reads Work?

Since descriptors linger in memory cells, a `read` must be a little smarter:

1. Read the address
2. **Plain value?**
   $\rightarrow$ Return it! (fast path)
3. **Descriptor pointer?**
   $\rightarrow$ Follow the pointer (slow path)
   - Status = SUCCESS?
     Return `new_val`
   - Status = FAILURE?
     Return `old_val`

But Wait – How Do Reads Work?

Since descriptors linger in memory cells, a `read` must be a little smarter:

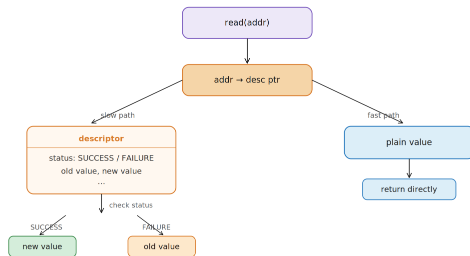
1. Read the address
2. **Plain value?**
   → Return it! (fast path)
3. **Descriptor pointer?**
   → Follow the pointer (slow path)
   - Status = SUCCESS?
     Return `new_val`
   - Status = FAILURE?
     Return `old_val`

In practice, the slow path is rare once GC kicks in.

But Wait – How Do Reads Work?

Since descriptors linger in memory cells, a `read` must be a little smarter:

1. Read the address
2. **Plain value?**
   → Return it! (fast path)
3. **Descriptor pointer?**
   → Follow the pointer (slow path)
   - Status = SUCCESS? Return `new_val`
   - Status = FAILURE? Return `old_val`



In practice, the slow path is rare once GC kicks in.

The Scoreboard – How Do We Compare?

The Scoreboard – How Do We Compare?

algorithm	CAS per k-CAS	read cost
Harris et al. (RDCSS)	≥ 3k+1	O(1), no indirection
PMCAS (persistent)	5k+1 + 2k+1 fences	O(1), no indirection
This work (volatile)	k+1	+1 level (desc present)
This work (persistent)	k+1 + 2 fences	+1 level (desc present)

The Scoreboard – How Do We Compare?

algorithm	CAS per k-CAS	read cost
Harris et al. (RDCSS)	≥ 3k+1	O(1), no indirection
PMCAS (persistent)	5k+1 + 2k+1 fences	O(1), no indirection
This work (volatile)	k+1	+1 level (desc present)
This work (persistent)	k+1 + 2 fences	+1 level (desc present)

Fewer CASes
$\Rightarrow$ less contention
$\Rightarrow$ higher throughput

Read indirection
Extra pointer follow
(only when desc present)

The Scoreboard – How Do We Compare?

algorithm	CAS per k-CAS	read cost
Harris et al. (RDCSS)	≥ 3k+1	O(1), no indirection
PMCAS (persistent)	5k+1 + 2k+1 fences	O(1), no indirection
This work (volatile)	k+1	+1 level (desc present)
This work (persistent)	k+1 + 2 fences	+1 level (desc present)

Fewer CASes

$\Rightarrow$ less contention

$\Rightarrow$ higher throughput

Read indirection

Extra pointer follow

(only when desc present)

Tradeoff: MCAS throughput improves significantly; reads pay a small cost only when a descriptor is still in place.

Recap – What We Learned

CAS → MCAS
Atomically swap k words at once

→

Lower Bound
$\geq k$ CAS needed (star config)

→

k+1 CAS
Skip the unlock! Lazy cleanup.

→

Tradeoff
Faster MCAS
Reads: +1 deref

Recap – What We Learned

CAS → MCAS
Atomically swap k words at once

Lower Bound
$\geq k$ CAS needed (star config)

k+1 CAS
Skip the unlock! Lazy cleanup.

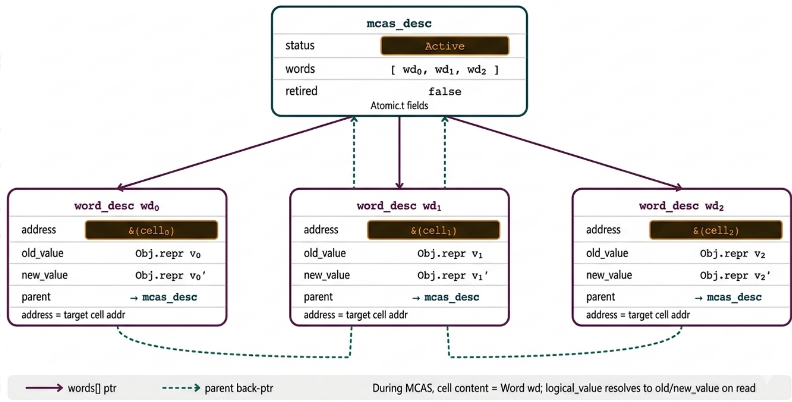
Tradeoff
Faster MCAS
Reads: +1 deref

TL;DR: By leaving descriptors in place instead of unlocking, we cut the critical path from $2k$+1 CAS down to just k+1 – provably near-optimal, lock-free, and linearizable. Not bad for a day's work!

Overview

1. What is MCAS?
2. The Impossibility Result
3. How the k+1 CAS Algorithm Works
4. Detailed Algorithm & Persistent Memory
   - Listing 1: Data Structures
   - Listing 2: `readInternal`
   - Listing 3: Main Algorithm
5. Experimental Evaluation & Conclusion
6. Research Question & Conclusion

Detailed Algorithm & Persistent Memory – Descriptor Object Graph



The descriptor links words to their parent MCAS operation to facilitate helping (Guerraoui et al., 2020).

Listing 1: Data Structures

Algorithm 1 Data structures used by our algorithm (Guerraoui et al., 2020)

```
1:  struct WordDescriptor {
2:     void * address;
3:     uintptr_t old_val;
4:     uintptr_t new_val;
5:     MCASDescriptor * parent;
6:  }
7:  enum StatusType { ACTIVE, SUCCESSFUL, FAILED }
8:  struct MCASDescriptor {
9:     StatusType status;
10:    size_t N;
11:    WordDescriptor words[N];
12: }
```

Listing 2: readInternal

Algorithm 2 The `readInternal` auxiliary function (Guerraoui et al., 2020)

```
1:  function READINTERNAL(addr, self)
2:      retry_read: val = *addr
3:      if not ISDESCRIPTOR(val) then return <val, val>
4:      else                                              ▷ found a descriptor
5:          parent = val->parent
6:          if parent != self and parent->status == ACTIVE then
7:              HELP(parent); goto retry_read
8:          end if
9:          if parent->status == SUCCESSFUL then return <val, val->new_val>
10:         else return <val, val->old_val>
11:     end if
12: end function
```

Listing 3: Main Algorithm

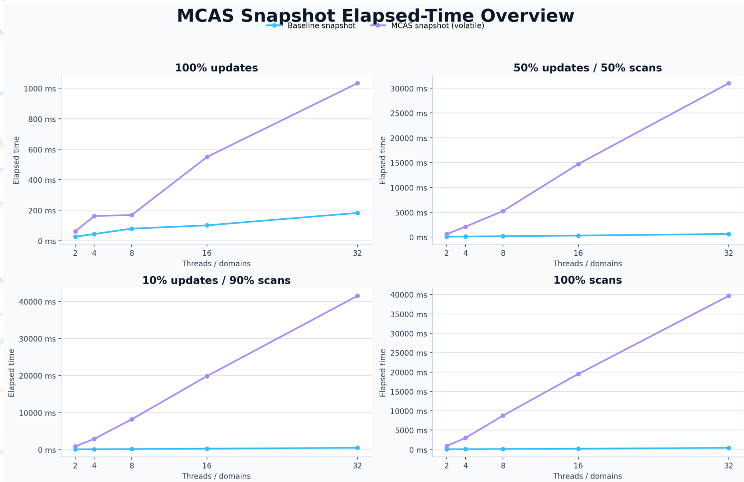
Algorithm 3 Main algorithm (Guerraoui et al., 2020)

```
1:  function READ(address)
2:      <content, value> = READINTERNAL(address, NULL); return value
3:  end function
4:  function MCAS(self)
5:      for i = 0 to self->N - 1 do
6:          w = &self->words[i]; retry_acquire:
7:          <content, value> = READINTERNAL(w->address, self)
8:          if content == w then continue
9:          end if
10:         if value != w->old_val or self->status != ACTIVE then break
11:         end if
12:         if not CAS(w->address, content, w) then goto retry_acquire
13:         end if
14:     end for
15:     if i == self->N then CAS(&self->status, ACTIVE, SUCCESSFUL)
16:     else CAS(&self->status, ACTIVE, FAILED)
17:     return self->status == SUCCESSFUL
18: end function
```

Overview

1. What is MCAS?
2. The Impossibility Result
3. How the $k+1$ CAS Algorithm Works
4. Detailed Algorithm & Persistent Memory
5. Experimental Evaluation & Conclusion
   - Performance Overview
   - Doubly Linked List Benchmark
   - Linked List Benchmark
6. Research Question & Conclusion

Performance Overview



General benchmark overview showing MCAS performance across different workloads.

Doubly Linked List Benchmark

MCAS Doubly-Linked-List Elapsed-Time Overview

Baseline Mutex DLL — MCAS Lock-Free DLL

90% updates / 10% search

67% updates / 33% search

100% search

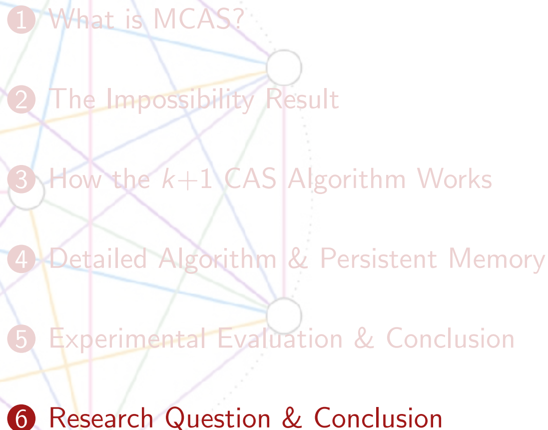
Elapsed time — Threads / domains

Performance results for the Doubly Linked List implementation.

Linked List Benchmark

Performance results for the Singly Linked List implementation.

Overview



1. What is MCAS?
2. The Impossibility Result
3. How the k+1 CAS Algorithm Works
4. Detailed Algorithm & Persistent Memory
5. Experimental Evaluation & Conclusion
6. Research Question & Conclusion

Research Question & Conclusion

Research Question:

Does MCAS provide enough expressive power to meaningfully simplify the implementation of atomic snapshot, and what is the performance cost of the software MCAS layer?

Research Question & Conclusion

Research Question:
Does MCAS provide enough expressive power to meaningfully simplify the implementation of atomic snapshot, and what is the performance cost of the software MCAS layer?

Answer: MCAS provides useful expressive power, but for atomic snapshot specifically, that expressive gain comes with a **very large performance cost**.

Research Question & Conclusion

Research Question:

Does MCAS provide enough expressive power to meaningfully simplify the implementation of atomic snapshot, and what is the performance cost of the software MCAS layer?

Answer: MCAS provides useful expressive power, but for atomic snapshot specifically, that expressive gain comes with a **very large performance cost**.

- **Expressiveness:** Gives a uniform way to express atomic multi-location changes, especially where several pointers or flags must move together.
- **Reusability:** The MCAS layer is a reusable abstraction (used in snapshot, linked lists, doubly linked lists).
- **Complexity vs Simplicity:** For atomic snapshot alone, the non-MCAS version is much simpler conceptually.

References I

Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. *International Symposium on Distributed Computing*, pages 265–279, 2002.

Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. volume 3, pages 1–26. ACM, 2019.