Practical Bounded Min Registers

Jordan

December 13, 2023

Introduction

In this presentation, we will be discussing wait-free implementations of k-bounded min registers.

- I'll introduce the model, what bounded min registers are, what wait-free means
- We'll discuss existing an existing implementation in the literature
- Then we'll discuss improvements to existing techniques that Faith and I have come up with
- And some open problems

Computational model

We are considering a system in which many processes communicate by performing operations on shared memory. A process can perform the following operations on a word w in memory, which holds some integer value:

- $w.read()$: Returns the current value of w.
- $w.write(v)$: Updates the value of w to v.
- $w.CAS(v, v')$: If w's value is v, w's value is updated to v'.
- $w.FAA(v)$: Computes the bitwise-and b of w's value and v, and updates w to hold b.

k-bounded min register

A k-bounded min register r is a shared object that stores an integer value in $\{0, \ldots, k-1\}$, initially holding $k-1$. It supports the following operations:

- $r.minRead()$:
  - Return the current value of r.
- $r.minWrite(v)$, where $0 \leq v < k$
  - If v is less than the value of r, set r's value to v.
  - Otherwise do nothing.

What does it mean for an implementation to be correct?

The correctness condition typically used for shared data structures is linearizability.

- For every operation, there should be a point during the operation in which we could say it 'took effect'
- e.g. If an instance of $r.minRead()$ returns v, then at some point during the instance r was equal to v.
- If an instance of $r.minWrite(v)$ terminates, then at some point during the instance r was less than v or r was updated to v.

Non-blocking progress guarantees

Here are the possible progress guarantees for non-blocking shared data structures:

- Obstruction-free: If a process p performs a sufficient number of steps without another process taking steps, it completes its operation.
- Lock-free: If a process p performs a sufficient number of steps, some process (which may or may not be p) has finished an operation.
- Wait-free: If a process p performs a sufficient number of steps, p completes its operation.

Related work

Aspnes, Attiya and Censor give a recursive wait-free implementation of a k-bounded min register[1].
A 1-bounded min-register z can be trivially implemented without memory, as so:

- $z.minRead()$:
  - Return 0
- $z.minWrite(v)$:
  - Do nothing.

This is clearly wait-free and linearizable.

[1]Their paper was actually about max registers, but we're giving an equivalent implementation for min registers.

Aspnes, Attiya, Censor Implementation

Suppose we have access to:

- a wait-free L-bounded min register *left*,
- a wait-free R-bounded min register *right*,
- a single bit *sw* stored in memory, initially $sw = 1$ (ON).

We can implement an $L + R$ bounded min register r.

Aspnes, Attiya, Censor Implementation

Initially, the min register looks like this:

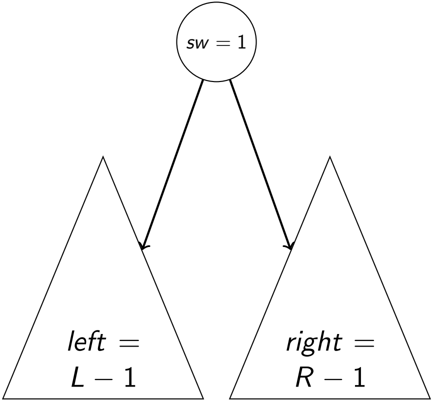


Figure: Their implementation takes the form of a binary tree. If the *sw* bit is 1, then the value of r is $L + right$'s value. Otherwise the value of r is *left*'s value.

Aspnes, Attiya, Censor Implementation

```
minRead()
    if switch = 0 then
        return left.minRead()
    else
        return right.minRead() + L
minWrite(v)
    if v < L then
        left.minWrite(v)
        switch ← 0
    else
        if switch = 1 then
            right.minWrite(v − L)
```

Suppose $L = 10$, $R = 10$, and we're implementing a 16-bounded min register.

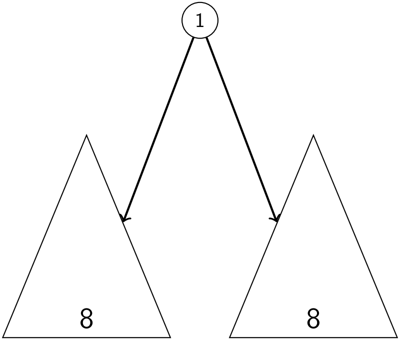
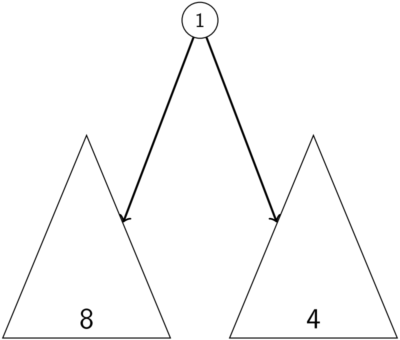


Figure: The min register initially looks like this.

After performing $r.minWrite(12)$, r looks like this:



After performing $r.minWrite(3)$, r looks like this:

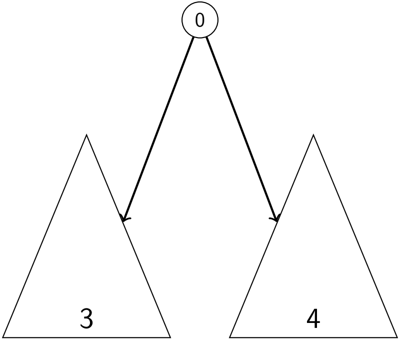


Figure: Essentially sw $= 0$ indicates that r's value $< L$, and that *left* holds the value of r.

Lemma 1.

Suppose left and right are wait-free bounded min registers. Then so is r.

Proof.

The $r.minRead()$ and $r.minWrite()$ functions use only use a single step to either read or write to sw, in addition to using the $minRead()$ and $minWrite()$ functions of $left$ and $right$.

How can we implement this practically?

How can we implement this on a real machine?

Practical Implementations of a b-bounded min register

Idea 1: A b-bounded min register object stores the bit *sw*, and *left*, *right* which are pointers to $\lceil \frac{b}{2} \rceil$ and $\lfloor \frac{b}{2} \rfloor$-bounded min register objects respectively.

Pros:

- Very simple to implement.
- Might reduce memory contention to store bits in different objects at different places in memory

Cons

- A lot of wasted space. On real machines, objects use at least 24 bytes per node used, so total space used is $24(b-1)$ bytes.
- Slow to traverse to different bounded min register objects from *left*/*right*.

Practical Implementations of a b-bounded min register

Idea 2: We can use a byte array-based tree, where each byte stores a switch register, and is the root of a subtree.

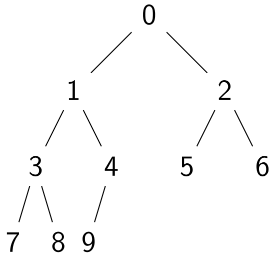
Array-based leveled binary tree

Definition 2.

A binary tree is leveled if every row, except possibly the last is filled.

We can use a b-byte array of atomic registers as a b node leveled binary tree. In a leveled binary tree, the left child of a node at index i, if one exists, is at index $2i + 1$. The right child, if it exists, is at index $2i + 2$.

0	1	2	3	4	5	6	7	8	9



Pros:

- More space efficient

Cons:

- Now need $b - 1$ bytes for a b-bounded register, which is still not great

But wait, there's more!

We are only using 1 bit for every byte in the array... What if instead we could use every bit in the array rather than one per byte? 8 times the capacity!

- Before:

0	X	X	X	X	X	X	X	1	X	X	X	X	X	X	X

- After:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Reads and Writes to individual bits

Reading a particular bit atomically is trivial. Can a byte and then use bitshifting.

But what about writing 0 to particular bit? We can use *fetch&and*!

To write 0 to bit i of a particular byte a atomically, we can use $a.FAA(11111111_2 - (1 << (7 - i)))$

Optimizing number of steps

Notice that any reads occur before any writes in both minRead and minWrite operations.

- minReads will read switches before going down the tree
- minWrites will read switches before going down the tree and then write to switches on the way back up

We could read and write multiple switches at a time if they're in the same word in memory. So we could implement word size min registers in this way. But is there a better way?

Word sized min registers

When I mentioned that most computers have FAA, Faith came up with a better idea:
Let w be a word of W bits such that the system supports FAA on w.

- We can implement a $(W + 1)$-bounded min register from w.
- Initially, w is all 1s.
- The value of w at any time is the position of the rightmost off bit, or W if all bits are on.
- A minRead will read w and return the position of the rightmost off bit.
- A minWrite(v) will use FAA to turn off the v-th bit, leaving the other bits unchanged.

This is a better implementation; minWrites now don't need to read, and we don't need to do any recursion or iteration to read/write different bits.

Larger min registers that are slightly more efficient

What about for larger min registers? Notice that in the earlier implementation, *sw* acts as a 2-bounded min register. It initially holds the value 1, and processes only read it or write 0 to it.

- What if we instead used our new word-size min register as the switch?
- Then we could use it to 'pick' between $(W+1)$ subtrees which are each min registers, rather than 2.
- Reduces the step complexity of a k-bounded register from $\log_2(k)$ to $\log_{W+1}(k)$.

Larger min registers that are slightly more efficient

Given we have words of size W and S-bounded min registers,

- We use a $(W+1)$-bounded min register sw, as the switch,
- and $(W+1)$ S-bounded min registers, $tree_0 \ldots tree_W$,
- to obtain a $(W+1)S$-bounded min register r.

$minRead()$

  $i \leftarrow sw.minRead()$

  return $tree_i.minRead() + is$

$minWrite(v)$

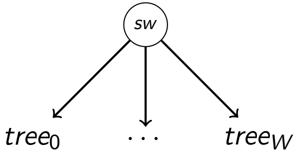
  $d \leftarrow \lfloor \frac{v}{S} \rfloor$

  $i \leftarrow sw.minRead()$

  if $i \geq d$ **then**

    $tree_d.minWrite(v - d \cdot S)$

    if $i > d$ **then** $sw.minWrite(d)$


sw
tree0
. . .
treeW

Open question

In both implementations, a wait-free b-bounded min register requires $b - 1$ bits and has $O(\log b)$ step complexity. Is there an implementation that uses fewer bits that has $O(\log b)$ or better step complexity?

Open questions

While technically wait-free, brute force implementations from CAS are slow. Given a word w of W bits, we could implement a 2^W bounded min register as follows:

- A minRead would return the current value of w.
- A minWrite(v) would read the current value c of w, and if c is greater than v, perform $w.CAS(c, v)$, repeatedly trying until the value of r less than v.

The step complexity of minWrites in this implementation would be $O(b - v)$, since a *minWrite* might have to perform a linear number of CAS operations on w before w holds a value that is less than or equal to v.