Practical Bounded Min Registers

Jordan

July 14, 2023

b-bounded min register

A b-bounded min register is a shared object with initial value b, supporting the *minRead*() and *minWrite*(v) operations:

- *minRead*():
  - Return the current value of the register.
- *minWrite*(v), $0 \leq v \leq b$
  - If v is less than the current value of the register, write v to the register.
  - Otherwise do nothing.

Implementation

Based on the Max Register implementation by Aspnes, Attiya and Censor in their paper *Max Registers, Counters, and Monotone Circuits*

0-bounded min register z:

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

Lemma 1.

z is wait-free and linearizable.

Lemma 2.

The minWrite function is a no-op and the minRead function returns after one instruction. For every execution an operation on z is linearized immediately upon the step it is invoked - since z already has value 0 in the initial configuration, the minWrite function, by definition, has no effect. The minRead function always returns 0 which is consistent with the state of the object when the minRead was invoked – z has value 0 throughout the execution.

Implementation

Suppose we have a $left.size$-bounded min register $left$, a $right.size$ bounded min register $right$, and a one bit multi-writer register sw with initial value 1, where $0 \leq right.size \leq left.size$.

```
($left.size + right.size + 1$)-bounded min register $r$:
$r.minRead()$ :
  • if $sw = 1$, return $right.minRead() + left.size + 1$
  • else return $left.minRead()$
$r.minWrite(v)$:
  • if $v > left.size$
      • if $sw = 1$, $right.minWrite(v - left.size - 1)$
  • else
      • $left.minWrite(v)$
      • $sw \leftarrow 0$
```

Lemma 3.

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 $right$.

Lemma 4.

Suppose left and right are linearizable bounded min registers. Then so is r.

Proof.

Consider an execution in which every instance of $r.minRead()$ and $r.minWrite(v)$ instance is allowed to complete. Given an incomplete execution e, since r is wait-free, we can append an extension to allow incomplete instances to finish.

- Let C_{right} be the set of $r.readMin()$ instances that read 1 from sw and $r.writeMin(v)$ ops that read 1 from sw where $v > left.size$.
- Let C_{left} be the set of $r.readMin()$ instances that read 0 from sw and $r.writeMinv$ ops with $v \leq left.size$, and thus write 0 to sw.
- Let C_{switch} be the set of $r.writeMin(v)$ that read 0 from sw where $v > left.size$.

Proof.

Observe that C_{left} is exactly the set of instances that access *left*. A $r.writeMin(v)$ instance in C_{left} is linearized at the first instant *sw* is 0 after is linearization point on its $left.writeMin(v)$, in the order these operations were linearized.

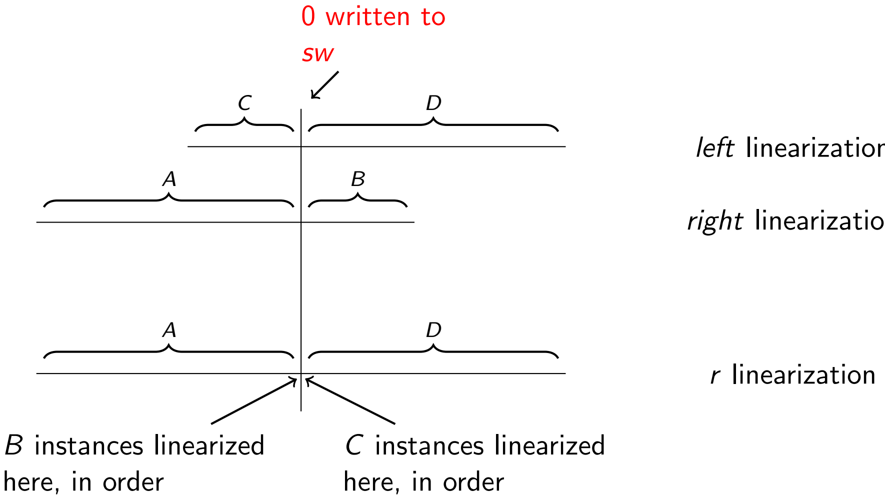
In particular, every instance in C_{switch} linearizes after the first such $r.writeMin(v)$ instance in C_{left} writes 0 to *sw*. And thus these writes in C_{switch} do nothing, since they linearize when r is already less than or equal to $left.size$.

A $r.readMin()$ instance in C_{left} is linearized at the same instant that its $left.readMin()$ instance is linearized, before which it reads 0 from *sw*. Thus it linearizes after the first instance of $r.writeMin(v)$ in C_{left} has linearized.

Proof.

Observe that C_{right} is exactly the set of instances that access *right*. Every instance in C_{right} is linearized in the same order as the order for their accesses to *right*, in either the configuration their access to right takes place or in the configuration before 0 is written to *sw*. Since every instance in C_{right} reads 1 from *sw*, their linearization point lies within their operation and before any operation in C_{left}.

Linearizability proof



Practical Implementations of a *b*-bounded min register

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

Pros:

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

Cons

- A lot of wasted space. On 64 bit systems, at least 24 bytes per node used, so total space used is $24b$ bytes.
  - Jeremy's trie, with height $= b$ requires a $b + 1$ bounded min register for every DelNode
  - A trie of height $= 31$ has 2^{30} delete nodes initially, thus this would be 500gb for the min registers alone!
- Slow to traverse to different bounded min register objects from *left*/*right*.

Practical Implementations of a b-bounded min register

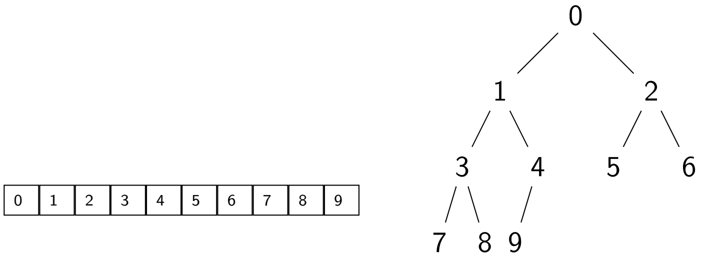
Idea 2: We can use an array-based tree, with where each array element (node) stores a switch register.

Array-based leveled binary tree

Definition 5.

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

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



The number of nodes in the left subtree in a leveled binary tree of size n:

$$a(n) = He(n) \cdot (n + 2^{\lfloor \log_2(n) \rfloor - 1} - 1) + (-1)^{He(n)} \cdot (2^{\lfloor \log_2(n) \rfloor} - 1)$$

where

$$He(n) = H(-n + 3 \cdot 2^{\lfloor \log_2(n) \rfloor - 1} - 1)$$

and

$$H(x) = [x \geq 0]$$

Proof by Bozinovski available here. https://oeis.org/A279521

Pros:

- More cache friendly - on x86-64 machines, accessing a single byte results in loading the next 63 bytes. If the cache is not invalidated after accessing first element of array, successive accesses do no incur a cache miss!

Cons:

- Now need *b* bytes for a size *b* register, which is better, but a Trie of height 32 would require 32 extra bytes per node for the min register alone, which is not great.
- If contention is high, cache may be invalidated by other threads that modify even different parts of the array, causing cache misses on successive reads/writes. (False sharing)

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... But what about writing 0 to particular bit? There's an atomic operation for that, we can use *fetch&and*!
To write 0 to bit i of a particular byte a atomically, we can use $fetch\&and(address(a), 11111111_2 - (1 << (7 - i)))$
For example to write 0 to bit 4 of a byte $x = 01001100_2$, we can use $fetch\&and(\&x, 11110111_2)$, which will turn off bit 4 and leave the other bits unchanged.

Implementation for *Jeremy*'s trie

If we restrict the maximum size of the register to $2^k - 1$ for some k, then the size of the left subtree = size of right subtree = $\lfloor \frac{2^k - 1}{2} \rfloor$.
I have reimplemented the *minWrite* and *minRead* functions to be iterative, and I use a 63-bounded min register using an array of 8 bytes. Initially the min register has value 63 but can be initialized to b with the use of *minWrite*(b).

Consequences

Pros:

- Can use use 8 bytes for a 63 bit bounded min register, rather than 63 bytes.
- No more risk of false sharing than with the array implementation.

Cons

- Contention costs may be higher since more registers are stored within the same byte.
- Doing a lot of reads from/writes to shared memory since we only read or write one bit at a time

One more thing

Notice that any reads occur before any writes in both minRead and minWrite operations.
Reads will read switches before going down the tree. Writes will read switches before going down the tree, then write on their way back up the tree.
There's no need for a gap between any two successive reads – or any two successive *fetch&and*s! Why not do all reads at the same time, and all *fetch&ands* at the same time! Then reads only need to read one 8 byte integer from memory, and writes only need to read one 8 byte integer from memory, then *fetch&and* one.