Jagiellonian University
Department of Theoretical Computer Science

Robert Obryk
Student Number: 1040504

Write-and-f-array: implementation and an application

Master's Thesis
Computer Science - IT Analyst

Supervisor:
dr Grzegorz Herman

September 2013

Abstract

We introduce a new shared memory object: the write-and-f-array, provide its wait-free implementation and use it to construct an improved wait-free implementation of the fetch-and-add object. The write-and-f-array generalizes single-writer write-and-snapshot[5] object in a similar way that the f-array[7] generalizes the multi-writer snapshot object. More specifically, a write-and-f-array is parameterized by an associative operator f and is conceptually an array with two atomic operations:

- write-and-f modifies a single array's element and returns the result of applying f to all the elements,
- read returns the result of applying f to all the array's elements.

We provide a wait-free implementation of an N-element write-and-f-array with $O(N \log N)$ memory complexity, $O(\log^3 N)$ step complexity of the write-and-f operation and $O(1)$ step complexity of the read operation. The implementation uses CAS objects and requires their size to be $\Omega(\log M)$, where M is the total number of write-and-f operations executed. We also show, how it can be modified to achieve $O(\log^2 N)$ step complexity of write-and-f, while increasing the memory complexity to $O(N \log^2 N)$.

The write-and-f-array can be applied to create a fetch-and-add object for P processes with $O(P \log P)$ memory complexity and $O(\log^3 P)$ step complexity of the fetch-and-add operation. This is the first implementation of fetch-and-add with polylogarithmic step complexity and subquadratic memory complexity that can be implemented without CAS or LL/SC objects of unrealistic size[8].

Contents

1. Introduction

The advent of popular manycore systems has made concurrent programming for shared memory systems an important topic. With increasing parallelism available in a single shared-memory system, performance of techniques used to communicate between concurrently executing threads, or to access memory common to many such threads, is becoming one of the most significant components of the performance of an application running on such a system. In many cases of communication-heavy tasks, simple coarse-grained locking is not enough to yield good performance, so programmers need to resort to lock-less communication and concurrent data structures. This work provides a new data structure with an implementation that can be used concurrently and doesn't use locks, and uses it to create an implementation of the fetch-and-add object (a kind of counter) with improved memory usage.

1.1. Concurrent Objects

A useful abstraction in a single-threaded system is an object: an entity with a set of methods one can invoke, and a specification of semantics of these methods. Naturally, a program can use multiple objects and invoke their methods in any order.

We want to use a similar abstraction to model interaction between threads in a multi-threaded system. Consider a set of independent, concurrently running threads (running possibly distinct code) that can only communicate by calling methods on some objects. We place no assumptions on the speed of execution and delays of the threads – they may wait arbitrarily long before a method call. We also assume that method calls on the objects complete instanteously, and that no two of them happen at the same time. This allows us to define semantics of the objects in the same fashion we define them in the single-threaded case: methods of every object are invoked in a known order, so single-threaded semantics suffice to determine the object's behaviour[1]. We will call such objects *concurrent objects* to emphasize that they may be accessed concurrently.

An object we will predominantly use in this work is a CAS (Compare-And-Swap) register. We will specify its semantics by providing pseudocode which correctly implements this object in a single-threaded program (this will be our method of choice of providing object's semantics):

```
1: var x : T
2: function RD
3:     return x
4: function WR(y)
5:     x ← y
```

[1]The whole program may still be nondeterministic, due to nondeterministic scheduling.

```
6: function CAS(o, n)
7:     if x = o then
8:         x ← n
9:         return true
10:    else
11:        return false
```

Intuitively, such a register holds a value that can be read (by RD), modified unconditionally (by WR) and modified conditionally (by CAS). It's interesting due to its universality properties[2] and because it is commonly seen in real-world hardware.

Example 1. *Let's consider two threads, executing following pseudocodes (X and Y are two CAS objects, with initial value 0):*

1. *1: X.WR(1)*
   2: Y.RD()
2. *1: Y.WR(1)*
   2: X.RD()

Figure 1.1 presents a possible execution of such two threads. Note that in any correct execution, at least one of the RD() calls returns 1.

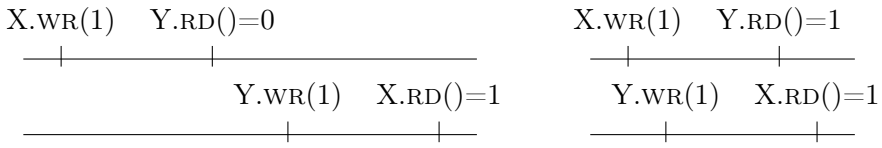

Figure 1.1: Two possible interleaved executions of threads from Example 1.

1.2. Serialization

In many situations it would be convenient to have more complicated concurrent objects: counters that can be incremented, queues that can be pushed to and popped from, etc. Alas, real-world hardware usually doesn't implement such objects directly. We will thus implement them using CAS registers or other simpler objects: we will substitute a set of simpler objects for every such object and for every method call on this object, execute a procedure that implements it instead. However, these prodecures might run concurrently, so we can't blindly use a single-threaded implementation of the object. Obviously, we will want such substitution to preserve semantics of the program in which it was substituted. The following condition immediately implies this[2]:

Definition 1. *An implementation of a concurrent object* serializes *iff for every execution of a program with the implementation substituted for the object: For every call to object's method that completes, we can define a* serialization point *in its execution interval (between its first and last constituent operation), such that all these calls would return the same values if they happened instaneously at their serialization points.*

[2]For a detailed discussion of subtleties related to this definition, refer to chapter 3.3-3.6 of [1]

An observant reader might note that if all procedures in an implementation never terminate, the definition is vacously true. Indeed, we will present some terminations guarantees separately, but first let us consider an example implementation of a simple structure.

Example 2. *Let's consider an increase-only counter: a shared object with the following semantics:*

```
1: var x : int
2: function INC
3:     x ← x + 1
4: function READ
5:     return x
```

We will see that the following implementation serializes to this shared object:

```
1: var V : int                                   ▷ A CAS object holding an int
2: function INC
3:     repeat
4:         o ← V.RD()
5:     until V.CAS(o, o + 1)
6: function READ
7:     return V.RD()
```

Note that every call to INC *executes exactly one successful call to* V.CAS. *Let us claim that the serialization point of* INC *is that successful call to* CAS, *and that the serialization point of* READ *is the call to* V.RD. *We can easily see that the number returned by* RD *is exactly the number of* INC*s that serialized before that* READ.

We can also see that this implementation satisfies some global progress property – when the CAS in INC fails, another INC must have suceeded very recently. In fact, it must have succeeded between the most recent execution of line 4 and the failed CAS. Still, a single INC operation can fail to terminate.

1.3. Wait-freedom

Consider an alternative implementation of an increment-only counter. In the following pseudocode, P denotes the number of threads in the system.

```
1: var T : array[P] of int                   ▷ T is an array of P CAS objects of type int
2: function INC_p
3:     x ← T[p].RD()
4:     T[p].WR(x + 1)
5: function READ
6:     R ← 0
7:     for i ← 1 to P do
8:         R ← R + T[i].RD
9:     return R
```

Here we differentiate between threads: the index p in INC corresponds to the current thread's ID from range $\{0, \ldots, P-1\}$. Let us first see that this implementation actually serializes to the increment-only counter. Obviously, we need to choose the call to WRITE as the serialization point

of INC. From the monotonicity of entries of T we can infer that READ returns a value between (inclusive) the number of calls to INC that have serialized before READ has started and the number of calls serialized before it has finished. Thus if a call to READ returns x, there is a point during its execution when exactly x INCs had been serialized. We can choose this point to be the serialization point of READ. Note that it need not correspond to an action executed by the implementation of READ.

We can see that this implementation satisfies a very strong termination condition – the number of steps every procedure takes is bounded and the bound depends only on the number of threads (and not on the behaviour of the scheduler). We will call such implementations *wait-free*. For a discussion of weaker termination guarantees, see chapter 3.7 of [1].

Every object that has a single-threaded specification has a wait-free implementation[2] using only CAS registers. However, all currently-known such constructions yield objects with step complexities inflated at least by a factor of the number of threads. Thus, creating faster wait-free implementations of data structures is interesting, also from a practical point of view.

1.4. Snapshots

The object implemented in this work is a generalization of a popular building block for multithreaded objects – an atomic snapshot object[4]. Its semantics can be defined by the following single-threaded pseudocode:

```
1: var T : array[N] of T
2: function UPDATE_i(x)
3:     T[i] ← x
4: function SCAN
5:     for i ← 0 to N do
6:         R[i] ← T[i]
7:     return R
```

Intuitively, this object represents an array that can be modified piecewise and read all at once. There are known wait-free implementations of an N-element atomic snapshot with $O(1)$ UPDATE and $O(N)$ SCAN step complexities[3]. These complexities are obviously optimal.

One can try to extend this structure in many ways. Two of them are particularly interesting for us. First, we can merge the update and scan operations into an operation that does an update and a scan[3]. This object, which we will call write-and-snapshot[5], can be defined by the following single-threaded specification:

```
1: var T : array[N] of T
2: function UPDATE-AND-SCAN_i(x)
3:     T[i] ← x
4:     for i ← 0 to N do
5:         R[i] ← T[i]
6:     return R
```

[3]Obviously, one can call these operations sequentially on a normal snapshot object. However, another update operation could then happen between the update and the scan. This cannot happen if the update and scan calls are merged into a single atomic operation.

For the second extension, let us note that in many cases one doesn't need the SCAN operation to return the whole array, but rather some sort of its digest. The f-arrays[7] allow one to do precisely that for digests expressible as a result of applying an associative operation to all the array's elements (for example, if we need to retrieve the sum of them). If we call the operator f, the following is a specification of an f-array:

```
1: var T : array[N] of 𝒯
2: function UPDATE_i(x)
3:     T[i] ← x
4: function SCAN
5:     return f(T[0], T[1], ..., T[N − 1])
```

There is an implementation of f-array with step complexities of UPDATE and SCAN being respectively $O(\log N)$ and $O(1)$ and the memory complexity being linear in N.

This work introduces a write-and-f-array: an object that combines these two modifications. Its semantics are defined by the single-threaded specification below:

```
1: var T : array[N] of 𝒯
2: function UPDATE-AND-f_i(x)
3:     T[i] ← x
4:     return f(T[0], T[1], ..., T[N − 1])
```

The main result of this work is an implementation of a single-writer write-and-f-array: that is, one that disallows concurrent modifications to the same array element. The implementation uses $O(N \log N)$ memory and the step complexity of the operation is $O(\log^3 N)$.

We then use this object to implement a fetch-and-add object. Fetch-and-add is a generalization of the increment-only counter from previous chapters. Its semantics are specified by the listing below:

```
1: var V : int
2: function FETCH-AND-ADD(x)
3:     r ← V
4:     V ← V + x
5:     return r
```

Fetch-and-add object can be used to produce unique identifiers, implement mutual exclusion, barrier synchronization[10], or work queues[11]. Its known wait-free shared memory implementations for P processes have $\Omega(P^2)$ memory complexity and $O(\log^2 P)$ step complexity [8][6]. They also need to employ complicated memory management techniques to bound their memory use. Our implementation reduces the memory complexity to $O(P \log P)$ while maintaining polylogarithmic step complexity.

2. The History Object

We will first implement a helper object – the history object. Intuitively, it contains a versioned memory cell and allows retrieval of past N values of the cell. The cell holds objects of type $\mathcal{T}$. The semantics of this object are precisely specified by the following single-threaded implementation:

```
1:  var H : array[...] of T                          ▷ An unbounded array of values
2:  var V : int                                      ▷ Current version
3:  function GET-CURRENT
4:      return V, H[V]
5:  function GET(v)
6:      if v ≤ V − N or v > V then
7:          return none
8:      else
9:          return H[v]
10: function PUBLISH(v, T)
11:     if v = V + 1 then
12:         H[v] ← T
13:         V ← v
14:         return true
15:     else
16:         return false
```

Our wait-free implementation will impose an additional constraint on its use: every call to PUBLISH is parameterized by an integer in range $[0, P)$ and executions of PUBLISH_p for the same p can not run simultaneously[1]. Both N and P must be known when the object is created.

Our implementation is presented below:

```
1:  var S : pair ⟨v : int, p : int⟩        ▷ Version number and p of the most recently published value
2:  var H : array[N] of pair ⟨v : int, T : T ⟩        ▷ A circular buffer of recent N values
3:  var L : array[P] of pair ⟨v : int, T : T ⟩        ▷ Temporary storage for values being published
4:  function GET-CURRENT
5:      s ← S.RD()                                    ▷ Possible serialization point
6:      HELP()
7:      return H[s.v mod N].RD()
8:  function HELP                                     ▷ Updates H[] with value from L[], if required
9:      s ← S.RD()
10:     l ← L[s.p].RD()
11:     h ← H[s.v mod N].RD()
12:     if l.v = s.v and h.v < s.v then
13:         H[s.v mod N].CAS(h, l)
```

[1] An obviously correct choice for the parameter is a unique thread ID.

```
14: function GET(v)
15:     s ← S.RD()                                   ▷ Possible serialization point
16:     if s.v < v then
17:         return none                              ▷ The requested version hasn't been published yet
18:     HELP()
19:     h ← H[v mod N].RD()
20:     if h.v = v then
21:         return h.T
22:     else
23:         return none
24: function PUBLISH_p(v, T)
25:     s ← S.RD()                                   ▷ Possible serialization point
26:     if v ≠ s.v + 1 then
27:         return false
28:     HELP()
29:     L[p].WR((v, T))
30:     if S.CAS(s, (v, p)) failed then              ▷ Possible serialization point
31:         return false
32:     return true
```

As the comments indicate, $L[p]$ is used to temporarily hold the value being published by PUBLISH$_p$. The array H is used to hold, roughly, the N most recently published values (the most recently published value might be absent; exact semantics will be given by the lemmas below) together with their version numbers. The field S holds the particulars of the most recently published value; the successful publish calls will serialize at the moment S changes. Our implementation contains an additional function HELP. It is used internally to write the most recently published value to the array H, if it isn't stored there already.

We will now prove some properties related to exact semantics of H and L and then use them to prove that our implementation serializes to the specification.

We posit that the serialization point of PUBLISH is in line 30, unless that line isn't reached. In that case (ie. when the call returns false in line 27) the serialization point is in line 25.

Lemma 1. *If $S = (v, p)$ at time t, and at some later point in time $L[p] = (v, T)$, then there was a successful call to* PUBLISH$_p(v, T)$ *with serialization point before time t.*

Proof. S must have been modified by a successful CAS in line 30. Obviously, a successful invocation of PUBLISH$_p(v, T)$ must have executed that CAS. Let's denote it by π. This invocation sets $L[p]$ to (v, T). What remains to be proven is that no later invocation of PUBLISH. will set $L[p]$ to (v, x) for any x. Only invocations of PUBLISH$_p$ modify $L[p]$ and they can start only after π has finished. Together with a simple observation that $S.v$ is nondecreasing in time this implies that any later invocation of PUBLISH that sets $L[p]$ must have been called with a strictly greater v. □

Observation 1. *If $H[i] = (v, T)$ at some point then a successful call to* PUBLISH.(v, T) *has had its serialization point earlier.*

Lemma 2. *Let π be a call to* PUBLISH$_p$. *Let π_q be a successful call to* PUBLISH$_q$ *and π_p a call to* PUBLISH$_p$, *both with serialization points after serialization point of π. Then there is a call to* HELP

that starts after the serialization point of π and ends before the one of π_q and before the execution of line 29 in π_p.

Proof. The serialization point of π must happen before π_q executes line 25 – if S has changed between line 25 and line 30 of π_q, π_q would fail. Thus the call to HELP from π_q will start after the serialization point of π and will finish before the serialization point of π_q. The call to HELP from π_p will start after serialization point of π (π_p may only start after π has finished) and will finish before line 29 of π_p. Of these two calls to HELP, the one that finishes earlier satisfies both conditions. □

Lemma 3. *If a call to* HELP*() has executed fully (ie. started and finished) after the serialization point of a successful* PUBLISH.$(v, \cdot)$*, then $H[v \mod N].v \geq v$.*

Proof. Let us first note that $H[v \mod N].v$ is nondecreasing. It thus suffices to prove that the condition is met at some point before the end of the call to HELP. We will do so by induction on the time at which the invocation of PUBLISH serializes. Consider a call to PUBLISH$_p(v, \cdot)$ that serializes at time t, and a call to HELP that starts after t. By Lemma 2 there is a call to HELP that finishes before the next serialization point of a successful PUBLISH. (thus before S is modified) and before $L[p]$ is modified. Without loss of generality we can assume that the call to HELP we are considering satisfies these conditions. By inductive hypothesis and Lemma 2, since a successful call to PUBLISH.$(v - N + 1, \cdot)$ has occured before time t, $H[v \mod N].first \geq v - N$ at time t. Taking into account that $H[i].v \mod N = i$, at any later point in time one of $H[v \mod N].v = v - N$ and $H[v \mod N] \geq v$ will hold. Thus the CAS in line 13 either succeeds, which causes the lemma's conclusion to start being satisfied, or fails, which means that the conclusion was already satisfied. □

Theorem 1. *The operations in the implementation of the history object serialize to the single-process object with serialization points of* PUBLISH. *as posited earlier.*

Proof. Note that $S.v$ is at all times equal to the first argument of the latest successful PUBLISH.

Let us first consider a call to GET-CURRENT that returns r. From Lemma 3 we know that $r.v \geq v.v$. By Observation 1, the successful PUBLISH.$(r.v, \cdot)$ happened before the read from $H[v.v \mod N]$. If $r.v = v.v$, then this call to PUBLISH was the most recent successful PUBLISH at the time when S was read, so we can serialize the operation there. Otherwise, it has happened (had its serialization point) after that read, so we can serialize the operation just after it.

Let us consider a calls to PUBLISH.:

- If a call to PUBLISH.$(v, \cdot)$ fails in line 27, then we can see that the most recently serialized (from the point of view of line 25) successful PUBLISH published a version different than $v - 1$.
- If a call to PUBLISH.$(v, \cdot)$ fails by failing the CAS in line 30, then a successful PUBLISH. has occured after line 25 had been executed, so the most recent successful PUBLISH. at the time of CAS has published a version greater than $v - 1$.
- If a call to PUBLISH.$(v, \cdot)$ succeeds, then at the time of CAS in line 30 the most recently published version is $v - 1$.

This suffices to prove that the value returned by PUBLISH is always correct with respect to the posited serialization order. What remains to consider are the calls to GET:

- If a call to GET(v) fails in line 17, then the most recent successful PUBLISH. when line 15 executed had a smaller version number than requested, so we can serialize the call to GET at line 15.
- If a call to GET(v) fails in line 23 because $h.v > v$, then (by Observation 1) a successful call to PUBLISH.$(h.v, \cdot)$ has occured by that time and $h.v \geq v + N$, so if we serialize GET at that point, it should fail.
- If a call to GET(v) fails in line 23 because $h.v < v$, then (by Lemma 3) a call to PUBLISH.$(v, \cdot)$ has not occured before the call to HELP in line X. We can thus serialize this GET just before the call to HELP has started.
- If a call to GET(v) succeeds, then by Observation 1 it returns a value that was published by a successful call to PUBLISH.$(v, \cdot)$. By Lemma 3, version $v + N$ was not published before the call to HELP has started. Thus we can serialize this GET just after the successfull call to PUBLISH.$(v, \cdot)$ has occured, or, if it has occured before GET started, just before the call to HELP started.

□

Obviously all operations run in $O(1)$ time. Memory complexity is $O(N + P)$, where N and P are the parameters defined at the beginning of this section. We will now use this object in the main result of this work, an implementation of a write-and-f-array.

3. The Write-and-f-array

We will now present our main result – an implementation of a write-and-f-array. We will actually present a wait-free implementation of a slightly richer object – the additional operations and return values are required for the recursive construction of the concurrent implementation.

A single-threaded specification of the structure is shown below. It uses a VERSION function, which is a black-box integer-valued function, subject to following conditions:

1. Subsequent return values of VERSION are nondecreasing.
2. If a call to VERSION(**false**) is followed by a call to VERSION(**true**), the second call must return a strictly greater integer than the first one.

```
1: var v[N] : array[N] of T
2: var last_update : array[N] of int
3: var last_version : array[N] of int
4: var last_value : array[N] of T
5: function WRITE-AND-F_i(T)
6:     v[i] ← T
7:     r ← f(v[0], v[1], ..., v[N − 1])
8:     last_update[i] ← last_update[i] + 1
9:     last_version[i] ← VERSION(true)
10:    last_value[i] ← r
11:    return last_update[i], last_version[i], last_value[i]
12: function GET-LAST(i)
13:    return last_update[i], last_version[i], last_value[i]
14: function READ
15:    return VERSION(false), f(v[0], v[1], ..., v[N − 1])
```

One can observe that the version numbers group the calls to WRITE-AND-F into groups of consecutive calls with no intervening calls to READ.

The concurrent implementation will be restricted by disallowing concurrent calls to WRITE-AND-F_i for the same i. It is conceptually very similar to Jayanti's tree-based f-array implementation. It uses a binary tree structure, with each array element assigned to a leaf and intervals of array elements assigned to internal nodes. We will construct it recursively. The implementation for $N = 1$ is presented below. If we note that no concurrent calls to WRITE-AND-F may be made in it, we can easily see that it is indeed correct and that all operations take constant time.

```
1: var S : pair < v : int, T : T >
2: function WRITE-AND-F_0(T')
3:     v, T ← S.RD()
4:     S.WR((v + 1, T'))
5:     return v + 1, v + 1, T'
6: function GET-LAST(i)
7:     v, T ← S.RD()
8:     return v, v, T
```

```
9: function READ
10:     return S.RD()
```

The implementation for $N > 1$ is presented below (interspersed with comments):

```
1: struct NODE-VALUE
2:     var T : array[2] of 𝒯                                  ▷ Child values
3:     var v : array[2] of int                                ▷ Child versions
4: struct LAST-VALUE
5:     var n : int
6:     var v : int
7:     var T : 𝒯
8: var C : array[2] of WRITE-AND-F-ARRAY                      ▷ of sizes ⌈N/2⌉ and ⌊N/2⌋, resp.
9: var H : history object<NODE-VALUE>                         ▷ of size N + 1, for N concurrent updates
10: var L : array[N] of LAST-VALUE
```

C[0] and C[1] are the subobjects from the recursive construction – they are of size $\lceil N/2 \rceil$ and $\lfloor N/2 \rfloor$, respectively. Array elements of the enclosing structure are mapped bijectively to consecutive array elements of these two subobjects. The mapping is defined by following functions (the element i is mapped to element $child\_id(i)$ in C[$side(i)$]):

$$side(i) = \begin{cases} 0 & i \in \{0, \ldots, \lceil N/2 \rceil - 1\} \\ 1 & i \in \{\lceil N/2 \rceil, \ldots, N-1\} \end{cases}$$

$$child\_id(i) = \begin{cases} i & i \in \{0, \ldots, \lceil N/2 \rceil - 1\} \\ i - \lceil N/2 \rceil & i \in \{\lceil N/2 \rceil, \ldots, N-1\} \end{cases}$$

History object H is used to store the object's current value (the value that would be returned by READ) and past values. The version numbers exported by the history object correspond to the values returned by VERSION in the specification. The elements of H aren't just values; they instead contain the values of both children together with their versions. The array L is used to store the values GET-LAST should return, but the values there might be stale (we will prove bounds on their staleness later on).

```
11: function READ
12:     v, h ← H.GET-CURRENT()
13:     return v, f(h.T[0], h.T[1])
14: function UPDATE_i
15:     v, h ← H.GET-CURRENT()
16:     h'.v[0], h'.T[0] ← C[0].READ()
17:     h'.v[1], h'.T[1] ← C[1].READ()
18:     HELP(v mod N)
19:     return H.PUBLISH_i(v + 1, h')
20: function WRITE-AND-F_i(T)
21:     C[side(i)].WRITE-AND-F_{child_id(i)}(T)
22:     if not UPDATE_i() then
23:         UPDATE_i()
24:     return GET-LAST(i)
```

The implementation of READ and WRITE-AND-F strongly resemble the f-array. WRITE-AND-F uses a helper function UPDATE. The intuition behind UPDATE is that it "pushes" new values from $C[0]$ and $C[1]$ to H. We will show that it suffices to attempt to call UPDATE twice to accomplish that.

```
25: function HELP(x)
26:     l_c ← C[side(x)].GET-LAST(child_id(x))
27:     v_current ← H.GET-CURRENT().v
28:     binary search {v_current − (N + 1), ..., v_current} for first v such that:
            H.GET(v) ≠ none and H.GET(v).v[side(x)] ≥ l_c.v
29:     if no such v exists then
30:         return
31:     h_old ← H.GET(v − 1)
32:     h_new ← H.GET(v)
33:     if h_old = none or h_new = none then
34:         return
35:     if side(x) = 0 then
36:         T ← f(l_c.T, h_old.T[1])
37:     else
38:         T ← f(h_new.T[0], l_c.T)
39:     l ← L[x].RD
40:     if l.n < l_C.n then
41:         L[x].CAS(l, (l_C.n, v, T))
42: function GET-LAST(x)
43:     HELP(x)
44:     return L[x].RD()
```

The use of binary search in line 28 warrants explanation. The predicate employed can obviously change its value in time. Thus, the binary search will return a v such that the predicate held at one point in time for v and didn't at another point for $v-1$. We will see that this is sufficient. Indeed, the result of the binary search will be important only in the cases when the value of the predicate doesn't change during the search.

We will first prove two simple facts about UPDATE:

Lemma 4. *For any two subsequent successful calls to H.*PUBLISH *with values h_1 and h_2, $h_1.v[i] \leq h_2.v[i]$ for both i.*

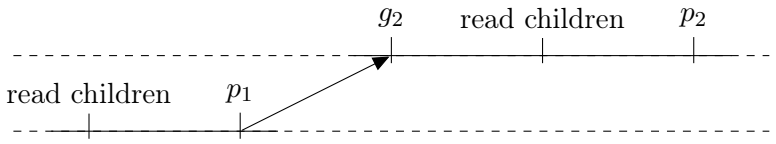

Figure 3.1: Diagram for proof of Lemma 4 (p_i – call to PUBLISH; g_i – call to GET-CURRENT in line 15 that corresponds to p_i).

Proof. Assume otherwise. Consider the first pair p_1, p_2 of consecutive calls to PUBLISH that contradicts the Lemma (see Figure 3.1). The only place PUBLISH is called is line 19 in UPDATE. p_2 had its

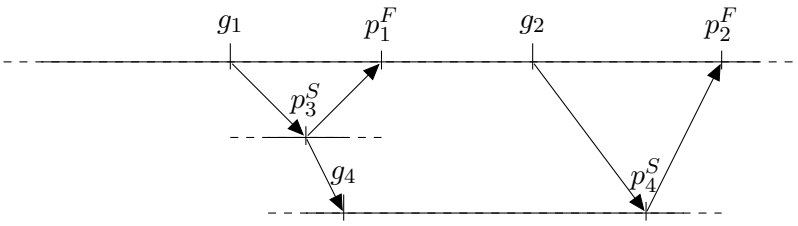

Figure 3.2: Consequences of two failed UPDATE calls (g_i – call to GET-CURRENT in line 15, p_i^S – successful call to PUBLISH in line 19, p_i^F – failed call to PUBLISH in line 19).

most recent call to GET-CURRENT (denoted by g_2) in line 15 sometime earlier. No successful publish could have occured between g_2 and p_2; otherwise p_2 would have failed. Thus, p_1 must have occured before g_2 and so its execution of lines 16 and 17 (denoted by "read children" on the diagram) must have occured strictly earlier than the same for the second call. This together with monotonicity of versions in children delivers the contradiction. □

Lemma 5. *For every execution of* WRITE-AND-F*: during execution of lines 22 and 23 at least one successful call to* UPDATE *occurs.*

Proof. If one of the calls to UPDATE from the lines in question suceeds, then the Lemma trivially holds. Thus we assume that they have both failed. The situation is depicted in Figure 3.2. g_1, p_1^F, g_2 and p_2^F correspond to these two failed calls to UPDATE. In order for p_1^F to fail, a PUBLISH must have succeeded between g_1 and p_1^F – let's call it p_3^S. Similarly, a PUBLISH must have suceeded between g_2 and p_2^F. Alas, this PUBLISH (denoted p_4^S) must have had its corresponding call to GET-CURRENT (denoted g_4) after p_3^S. The call to UPDATE that called g_4 and p_4^S satisfies the Lemma's conclusion. □

Let us consider an execution of WRITE-AND-F$_i$. Let (n_c, v_c, T_c) be the return value of the corresponding call to $C[side(i)]$.WRITE-AND-F$_{child_id(i)}$. Then by the time the call to WRITE-AND-F$_i$ finishes, the value of the most recently published node has $v[side(i)] \geq v_c$. We will consider the first call to PUBLISH that publishes such a node value as the serialization point of the WRITE-AND-F. Obviously, many calls to WRITE-AND-F can then serialize at the same instant in time. We will order them by taking first these with $side(i) = 0$ in the order in which their corresponding calls to WRITE-AND-F happened in $C[0]$ and then those with $side(i) = 1$ in the corresponding order. The choice of order on elements of C is arbitrary, albeit it is reflected in the computation of T in HELP.

Before we prove that the implementation is correct, we need the following two lemmas about the array L.

Lemma 6. *If* HELP *is called after the serialization point of the n^{th} call to* WRITE-AND-F$_i$*, then $L[i].n \geq n$ after the call to* HELP *completes.*

Proof. We will prove this Lemma by induction on n. The proof for the base case will be very similar to the induction step, so we present both of them simultaneously. Let $p(v)$ be the time the successfull call to PUBLISH$(v, \cdot)$ occured. Consider an UPDATE$_i$ that was serialized at $p(v)$. For every $u \equiv i \pmod N$, a call to HELP(i) starts and finishes between $p(u-1)$ and $p(u)$. Thus, there is such a call that starts and finishes between $p(v)$ and $p(v+N)$. Without loss of generality we can

assume that the call to HELP from the Lemma's hypothesis executes in this time interval. In that case, the binary search from line 28 will not see any GET(x) for $x \geq v-1$ returning **none** and will return version v. For the same reason, both GETs will succeed.

We want to prove that $L[i].n \geq n-1$ from $p(v)$ on. For the base case this holds, because the initial values satisfy this requirement. For the inductive step, we have to use the hypothesis: at $p(v)$, the $n-1^{\text{th}}$ call to WRITE-AND-F$_i$ has completed, so a call to HELP(i) has completed between the serialization point of that call to WRITE-AND-F$_i$ and $p(v)$. Thus $L[i].n \geq n-1$ from $p(v)$ on. If the condition in line 40 is not satisfied or the CAS in the next line fails, the Lemma hold. Otherwise, by the time the CAS succeeds the Lemma will obviously hold.

□

Lemma 7. *If a call to* HELP(i) *sets* $L[i]$ *to* (n, v, T)*, then by the time the call has finished:*

1. WRITE-AND-F$_i$ *has been called at least* n *times.*
2. *Let* (n_c, v_c, T_c) *be the return value of the* n^{th} *call to* $C[side(i)]$.WRITE-AND-F$_{child_id(p)}$*. Let* v' *and* h_{new} *be the version and node value published at the serialization point of the* n^{th} WRITE-AND-F$_i$ *and* h_{old} *be the node value published with version* $v'-1$*. Then* $v = v'$ *and:*
   - *if* $side(p) = 0$ *then* $T = f(T_c, h_{old}.T[1])$*,*
   - *if* $side(p) = 1$ *then* $T = f(h_{new}.T[0], T_c)$*.*

Proof. For the first part, we need to note that by the time line 26 executes, the n^{th} WRITE-AND-F$_{child_id(i)}$ has already executed in the appropriate child. If the binary search fails or returns version different from v', HELP will exit early. Indeed, if the binary search returns a different v, GET($v-1$) will fail. This immediately proves the first part of the Lemma and shows that h_{old} and h_{new} in the code have the same values as h_{old} and h_{new} in the Lemma, which proves the second part. □

Theorem 2. *The operations in multi-process write-and-f-array serialize to the single-process specification with serialization point of* WRITE-AND-F *as posited earlier.*

Proof. Let us choose the call to GET-CURRENT as the serialization point of READ. We will first prove that these are correct serialization points for WRITE-AND-F and READ. Consider the n^{th} call to WRITE-AND-F$_i$ and let (n', v', T') be its return value. As this triple was read from $L[i]$, we can use Lemmas 6 and 7 to show that $n' = n$ (note that executions of WRITE-AND-F$_i$ for the same i are disjoint in time). From Lemma 7 we get that v' is equal to the version published at the serialization point of the call to WRITE-AND-F. This obviously implies that VERSION is nondecreasing for subsequent calls to WRITE-AND-F and READ. As only the calls to WRITE-AND-F that serialize on a single PUBLISH return equal v, no READ call can serialize in between. This proves both properties required from VERSION.

We still need to prove that T' and the values returned by READ are consistent with the posited serialization order. Consider a set of WRITE-AND-F calls that are serialized together. Let h_{new} and h_{old} be the history elements published, respectively, at the serialization point and most recently before it. Let a_i be the sequence of return values of $C[0]$.WRITE-AND-F calls corresponding to WRITE-AND-F calls in question, in serialization order. Let b_i be a similar sequence for $C[1]$. By Lemma 4 these are exactly the calls to $C[j]$.WRITE-AND-F serialized with versions in $(h_{old}.v[j], h_{new}.v[j]]$ for $j \in \{0, 1\}$. By Lemma 7 the sequence returned by the WRITE-AND-F calls is

$f(a_0, b_0)$, $f(a_1, b_0)$, ..., $f(h_{new}.T[0], b_0)$, $f(h_{new}.T[0], b_1)$, ... in the order of serialization. Additionally, $f(h_{new}.T[0], h_{new}.T[1])$ is the value that would be returned by any READ calls until the next serialization point of WRITE-AND-F. This suffices to show that the return values of consecutive calls to WRITE-AND-F are actually the results of applying the updates. This proves that values returned by GET and UPDATE are correct wrt. posited serialization order.

We still need to show that we can correctly serialize the calls to GET-LAST. Lemma 7 implies that two calls to GET-LAST(i) will never return different triples with equal n. As WRITE-AND-F returns the result of a call to GET-LAST, we just need to show that GET-LAST(i) can be serialized so that the n in its return value is the number of previously serialized calls to WRITE-AND-F$_i$. From Lemma 6 we get that n is at least the number of calls to WRITE-AND-F$_i$ that serialized before GET-LAST started. From Lemma 7 we know that n was at most the number of such calls that serialized before GET-LAST finished. Thus, there is a point in the execution interval of GET-LAST when n is equal to the number of such calls that have already serialized. We choose any such point in time to be the serialization point of the call to GET-LAST. □

The structure uses $O(N \log N)$ memory. A GET-LAST operation takes $O(\log^2 N)$ time, a READ operation takes $O(1)$ time and a WRITE-AND-F operation takes $O(\log^3 N)$ time.

We can construct a fetch-and-add object for P threads by using a write-and-f-array of size P with addition as the operation f. Every thread is assigned an element in the array, and modifies only that element. This gives us a fetch-and-add object for P threads with $O(\log^3 P)$ step complexity and $O(P \log P)$ memory complexity. Additionally, the object implements a method that retrieves the current value in $O(1)$ time.

4. Implementation

A fetch-and-add object was implemented using the above-mentioned construction from the write-and-f object. The implementation is written in C++ and uses the C++11 standard library support for atomic operations. It is published on Github [1].

Direct implementation of previously described algorithm would require CAS objects of size larger than the 64-bit Intel processors support. However, the way most of these objects are accessed in the implementation makes it possible to split them into multiple CAS objects. The only CAS objects that don't afford this transformation are the version number holders from the history object. We use 64-bit versions and thus even these objects are no larger than 128 bits. Thus, our implementation can run on 64-bit Intel architecture processors, but not on 32-bit ones.

Unfortunately, the C++ standard library support for atomic operations doesn't allow atomic reads of a part of a larger atomic variable (specifically, reads of 64-bit halves of a 128-bit variable). As 128-bit atomic reads on amd64 are very costly (they use a 128-bit wide CAS), limiting their number was very important for the efficiency's sake. Thus, the implementation makes an unwarranted assumption that an atomic variable has the same memory layout as a normal, non-atomic variable of the same size. This assumption holds for gcc 4.6 on amd64.

The correctness of the implementation was tested experimentally both on real hardware and by using the Relacy Race Detector[12]. Relacy Race Detector is a framework for testing multi-threaded programs in C++11. It substitues its own implementation of synchronization primitives and atomic variables and runs user-supplied tests multiple times, with various interleavings of threads. It can detect data races on non-atomic variables, deadlock conditions, and failed user-supplied invariant and assertion checks. One notable feature is the support for atomic operations with reduced consistency – Relacy can simulate acquire/release semantics, as specified in the C++11 atomic variables library. Our fetch-and-add implementation can be compiled both to run on bare hardware and to run in Relacy. The use of Relacy not only provided confidence about correctness of the implementation, but also allowed us to downgrade consistency guarantees of some writes.

The implementation was benchmarked on a machine with 4 12-core AMD Opteron 6174 processors. The benchmark created a fetch-and-add object and started a preset number of threads. Each thread incremented the counter in a loop, counting its operations locally. After 20 seconds elapsed, all threads were signalled to stop and their operation counters were summed. For comparison, the same benchmark was run with the fetch-and-add object implemented by a simple read-modify-write loop. All benchmarks were run when the machine was minimally loaded (had 5 minute load average smaller than 0.5 at the start of the benchmark).

The results of the benchmark are shown in Figure 4.1. The figure contains a plot of average time it takes to complete one operation as a function of the number of concurrently executing threads. The horizontal scale of the figure is proportional to the square of the logarithm of the number of threads. The reason for this is that the optimistic time complexity of our fetch-and-add object is $\Theta(\log^2 P)$. We can see that the graph for our implementation approximates a straight line up to about 30 threads, where it starts to diverge upward. This peculiar divergence disappears if we assign one core to each thread and force it to run only there (by setting CPU affinity). Other affinity

[1] The version current as of the time this work is written is available at: `https://github.com/robryk/parsum/tree/d3433f7f7b137b52a73cfb244d46081528c696c3`

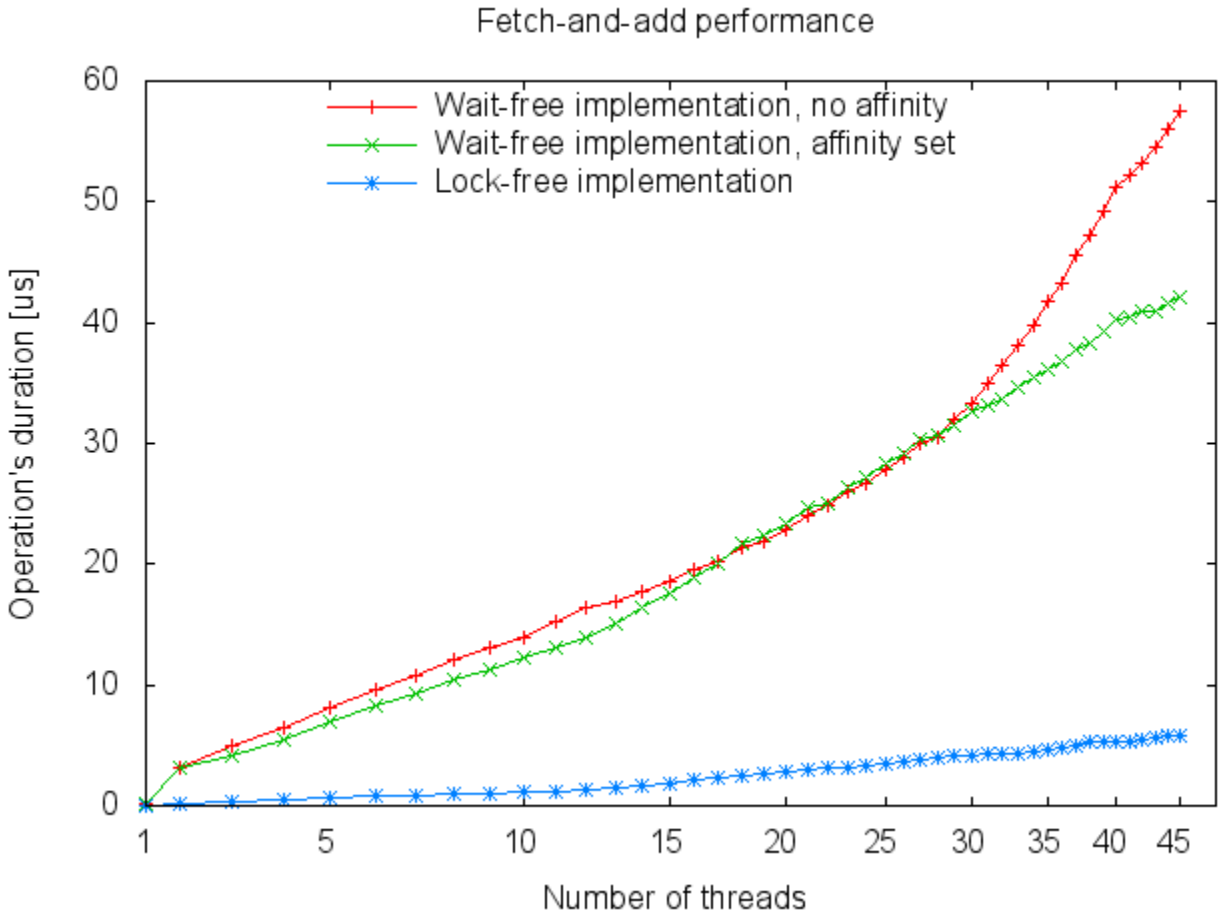

Figure 4.1: Results of the benchmark

settings yielded graphs similar to one of these two. We were unable to explain this behaviour. It might be worth noting that the anomaly in the no-affinity case happens when we start using all 4 physical CPUs.

It can be easily seen from the graph that our fetch-and-add object is far from practical. It is about 100 times slower than the naive lock-free implementation for small numbers of threads and about 10 times slower for larger numbers of threads. We believe that this implementation is suboptimal, although achieving similar performance as the naive implementation seems impossible. One change that could improve the performance of this implementation is changing the arity of the tree. The construction can easily be adapted to trees with larger arity and this might decrease the number of expensive CAS operations at the expense of the number of atomic reads, which are comparatively cheap on Intel processors.

5. Extensions and Open Problems

Our implementation of write-and-f-array can be modified by splitting the work done by HELP into $O(\log N)$ separate pieces (updates on consecutive levels of the tree) and running only a single such piece in UPDATE. If we then enlarge the history size to $O(N \log N)$, Lemma 6 still holds. This modification increases the memory complexity to $O(N \log^2 N)$ and decreases the step complexity of WRITE-AND-F to $O(\log^2 P)$.

The structure can be straightforwardly modified to use LL/CS instead of CAS, albeit it then requires the ability to have two outstanding LLs at the same time. The modified version can be further modified to use bounded version numbers, from a range of size $O(N)$. Unfortunately, popular architectures that implement LL/SC (such as PowerPC) allow only one outstanding LL at a time.

The effect of the arity of the tree on the performance seems to be worth investigating.

We pose also two purely theoretical problems:

- Our implementation of write-and-f-array is single-writer, so it is natural to consider the multi-writer write-and-f-array. Can we construct a multi-writer write-and-f-array in subquadratic memory with similar step complexities of the operations?
- The memory consumption of write-and-f-array implemented with atomic registers and CAS or LL/CS objects is bounded from below by a linear function of the number of processes that can execute WRITE-AND-F concurrently[9], so it must be $\Omega(N)$. Can we achieve lower memory and/or step complexities?

6. Acknowledgements

The author would like to thank his advisor, dr. Grzegorz Herman for many fruitful discussions and help while preparing this work. He would also like to thank Szymon Acedański for his help in improving the presentation of this work.

This work was supported by the Polish Ministry of Science and Higher Education program "Diamentowy Grant".

Bibliography

[1] Herlihy M., Shavit N. "The Art of Multiprocessor Programming", Morgan Kaufmann Publishers Inc., 2008

[2] Herlihy M. "Wait-free synchronization", ACM Trans. Program. Lang. Syst. 13, 1, pp. 124–149 (1991)

[3] Riany Y., Shavit N., Touitou D. "Towards a practical snapshot algorithm", Theoretical Computer Science 269, pp. 163–201 (2001)

[4] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., Shavit N. "Atomic snapshots of shared memory", Journal of the ACM 40, pp. 873–890 (1993)

[5] Afek Y., Weisberger E. "The instancy of snapshots and commuting objects", Journal of Algorithms 30.1, pp. 68–105 (1999)

[6] Chandra T. D., Jayanti P., Tan K. "A polylog time wait-free construction for closed objects", Proceedings the 17th PODC, pp. 287–296 (1998)

[7] Jayanti P. "f-arrays: Implementation and applications", Proceedings of the 21st PODC, pp. 270–279 (2002)

[8] Ellen F., Ramachandran V., Woelfel P. "Efficient Fetch-And-Increment", Lecture Notes in Computer Science 7611, pp. 16–30 (2012)

[9] Fich F. E., Hendler D., Shavit N. "Linear lower bounds on real-world implementations of concurrent objects", Foundations of Computer Science, pp. 165–173 (2005)

[10] Freudenthal, E., Gottlieb, A. "Process coordination with fetch-and-increment", Proceedings of ASPLOS-IV, pp. 260—268 (1991)

[11] Goodman, J., Vernon, M., Woest, P. "Efficent synchronization primitives for large-scale cache-coherent multiprocessors", Proceedings of ASPLOS-III, pp. 64-–75 (1989)

[12] Vyukov D., Relacy Race Detector, `http://www.1024cores.net/home/relacy-race-detector`