f-Arrays: Implementation and Applications*

Prasad Jayanti
6211 Sudikoff Lab for Computer Science
Dartmouth College
Hanover, NH 03755

ABSTRACT

We introduce *f-array*, a new type of shared object that generalizes the multiwriter snapshot object, and design efficient (linearizable and wait-free) algorithms for implementing it. f-arrays have made possible improved solutions to some important problems, as listed below:

- A wait-free implementation of multiwriter snapshot, where the time complexity of *scan* and *update* operations is independent of the number of processes accessing the implementation.
- A wait-free implementation of counter object whose time complexity has the dual advantage that it is adaptive and guarantees a small worst-case bound: the time complexity is $O(1)$ for *read* and $O(\min(k, \log n))$ for *increment*, where k is point contention and n is the maximum number of processes that the implementation is designed to handle.
- A wait-free implementation of a restricted version of a priority queue with similar time complexity as the counter implementation.
- A local spinning mutual exclusion algorithm that admits processes into Critical Section (CS) according to their priorities; processes with the same priority enter the CS in first-come-first-served order. In both cache coherent and NUMA multiprocessors, a process makes at most $O(\min(k, \log n))$ remote references to complete the entry and exit sections once. To the best of our knowledge, this is the first mutual exclusion algorithm that supports process priorities and has sublinear worst-case time complexity.

All algorithms in this paper require support for LL/SC instructions.

*This work is partially supported by NSF Grant CCR-9803678 and by Alfred P. Sloan Foundation Fellowship.

PODC 2002, July 21-24, 2002, Monterey, California, USA.

1. INTRODUCTION

In this paper, we define *f-array*, a new type of shared object that generalizes the multiwriter snapshot object [1, 3], and design efficient algorithms for implementing it. We also show how, using f-arrays, improved solutions are possible for some important problems.

To motivate f-arrays, consider the problem of atomically computing the sum of the elements of an array of m components, even as these components are concurrently updated by processes. One solution is to take a snapshot of the array and return the sum of the values in the snapshot. This solution is obviously correct, but is an overkill: intuitively, the sum of an array has less information content than the snapshot of the array and, hence, should be more efficiently computable. This hunch—that computing the sum of an array ought to be simpler than taking a snapshot—applies equally to other aggregation functions, such as max, min, number of negative values in the array, whether the array is sorted *etc.*

The above observations led us to define an f-array, for any function f, as essentially a multiwriter snapshot object with the following difference: the scan operation, instead of returning the values of all array components, returns the result of applying the function f to the values of components (in the rest of this paper, we call this operation *read*, not *scan*). There is only one other difference between a snapshot object and an f-array. In a snapshot object, *write* is the only operation by which a process can change the state of a component. We have removed this restriction when defining an f-array: the type of each array component is unrestricted; processes may update a component by applying any operation allowed by that component's type. The different components of the array can be of different types.

More precisely, if $T_1, \ldots, T_m$ are object types, then *f-array*$[T_1, \ldots, T_m]$ is a new type that consists of m components; the ith component has type T_i and, therefore, can be updated by any operation supported by type T_i. Additionally, the f-array supports a *read* operation which returns $f(v_1, \ldots, v_m)$, where $v_1, \ldots, v_m$ are the values of the m components.

To illustrate the above definition, let T_1 be the standard *register* type that supports read and write operations, T_2 be a type that supports read, write and fetch&add, f be the multiplication function, and $\mathcal{O}$ be an object of type f-

array$[T_1, T_2]$. Suppose that $\mathcal{O}$'s initial value is $[0, 0]$ and processes apply operations in the following order: p_1 applies *write*(5) to component 2, p_1 applies *write*(10) to component 1, p_2 applies *fetch&add*(15) to component 2, and p_3 applies *read*. Then, p_2's fetch&add returns 5 and p_3's read returns 200.

1.1 Contribution

This paper makes two contributions, described as follows. (Our results below include several implementations of shared objects. All of these implementations are linearizable[1] [10] and wait-free[2] [14, 17, 8].)

1. Efficient algorithms for implementing f-arrays.

We present an algorithm for implementing f-array$[T_1, \ldots, T_m]$, for any function f and any types $T_1, \ldots, T_m$. The algorithm requires a total of $m + 1$ shared objects: one object of each type T_i, $1 \leq i \leq m$, and an LL/SC object O that is large enough to store any value in the range of the function f. The time complexity is as follows. An update operation on any of the m components of the f-array performs $O(m)$ operations on primitive objects (*i.e.*, the objects used in the implementation). A read operation on the f-array performs just one operation on the primitive LL/SC object.

A nice feature of the algorithm is that its correctness, space and time complexities are all independent of the number of processes concurrently accessing the implementation. The number of processes can be even infinite.

We present a second algorithm, called the *Tree Algorithm*, for implementing f-arrays. This algorithm reduces the time complexity of an update operation to $O(\log m)$, while preserving all other good properties of the first algorithm. However, this algorithm works only if the function f has a "balanced tree structure" (explained later). Fortunately, as we will show, many common aggregation functions have this structure.

2. Using f-arrays to efficiently solve other problems.

The second contribution of this paper is to show that some important synchronization problems have efficient reductions to f-arrays. These reductions have given rise to improved algorithms for these problems. Our algorithms require support for LL/SC instructions. The specific results are:

(i). Since f-array is a generalization of a multiwriter snapshot object, our algorithm for implementing an f-array also implements a multiwriter snapshot. This implementation has two advantages over existing ones: (1) the time complexity of scan and of writing to a component are independent of the number of processes for which the algorithm is designed (the time complexity is $O(m)$ for both scan and write, where m is the number of components), and (2) a component can be updated with operations other than write (provided that the hardware supports these operations).

(ii). A *counter* supports the operations *read* and *inc*(d), which adds d to the object's value and returns an acknowledgement. We present a counter implementation whose time complexity has the dual advantage that it is adaptive and guarantees a small worst-case bound: the time complexity is $O(1)$ for *read* and $O(\min(k, \log n))$ for *inc*, where k is point contention during the operation and n is the maximum number of processes that the implementation is designed to handle. (Point contention at time t is the number of processes accessing the implementation at t.) Since counter is a commonly needed object in parallel programs, we believe that this result is significant.

(iii). We define a new type of object that we call *priority process-queue*, which supports *insert*, *findmin* and *delete* operations, with two restrictions: (i) an element can be deleted only by the process that inserted it, and (ii) after a process p inserts an element, p must delete that element before inserting a new one.

This object is useful in any application that needs to maintain a prioritized queue of processes. A process enqueues itself by inserting a pair consisting of its name and priority. Then, with *findmin*, it can determine the highest priority waiting process and help that process in a manner suitable to the application. Finally, when appropriate, the process can remove itself from the queue. We employ these ideas to design a novel mutual exclusion algorithm.

We implement a priority process-queue with an adaptive time complexity of $O(1)$ for *findmin* and $O(\min(k, \log n))$ for a matching pair of *insert* followed by *delete*, where k and n are as before.

(iv). Using the adaptive counter and the adaptive priority process-queue objects (implemented earlier), we design a starvation-free mutual exclusion algorithm with three main features, described as follows.

First, the algorithm allows processes to have priorities and ensures that waiting processes enter the Critical Section (CS) in the order of their priorities. Processes with the same priority enter the CS in "First-Come-First-Served" (FCFS) order [13]. Second, the algorithm is local spinning: while busy-waiting, a process accesses only the variables in its local memory module. Third, the algorithm is adaptive and guarantees small worst-case time complexity: to execute the Entry Section and the Exit Section once, a process performs at most $O(\min(k, \log n))$ operations on remote variables, where n is the maximum number of processes for which the algorithm is designed and k is the point contention (this time complexity bound is good for both cache-coherent and NUMA multiprocessors).

To the best of our knowledge, our algorithm has two advan-

[1] A *shared object is linearizable* if, even though operations applied on the object are not instantaneous, they appear to be so; that is, every operation appears to take effect at some instant between its invocation and completion. An *implementation is linearizable* if it ensures that the implemented object O is linearizable whenever the primitive objects from which O is implemented are linearizable.

[2] An implementation is *wait-free* if every process completes its operation on the implemented object in a bounded number of its steps, regardless of whether other processes are slow, fast or have crashed.

tages over existing ones. First, among the mutual exclusion algorithms that support processes priorities, this is the first one to have a sublinear worst-case time complexity. Existing algorithms [15, 7, 12] have a time complexity of $O(k)$ and so their worst-case time complexity is $O(n)$. Second, among the algorithms that are based on LL/SC or compare&swap or the standard read/write instructions, this is the first local spinning algorithm to satisfy FCFS property or support process priorities. Anderson and Kim's algorithm [4] is local spinning and uses only read and write instructions, but it does not satisfy FCFS and does not support process priorities.

1.2 Support for LL/SC

Since all algorithms in this paper require support for *load-link* (LL) and *store-conditional* (SC) operations, in the following we briefly review their semantics and state how they can be realized.

The operation *LL*, when applied on an object O by a process p, returns O's value. The operation $SC(v)$ by p "succeeds" if and only if no process performed a successful SC on O since p's latest *LL*. If *SC* succeeds, it changes O's value to v and returns *true*. Otherwise, O's value remains unchanged and *SC* returns *false*. The following two points clarify the kind of LL/SC objects required by this paper's algorithms:

1. The algorithms of this paper require support for standard LL/SC instructions, defined above. In particular, the weak version of LL/SC [5], known as WLL/WSC, is not adequate (because the WLL operation does not always return the object's value).

2. Our algorithm for implementing an f-array requires an LL/SC object that is large enough to hold any value in the range of the function f. For certain functions f, the value of f fits into a single word: for instance, if the components of the f-array are wordsized and f is the max function, the value of f is also wordsized. On the other hand, if $f(v_1, \ldots, v_m) = [v_1, \ldots, v_m]$ (*i.e.*, read operation of the f-array returns a snapshot of the array), the value of f does not fit in a single word; in this case, we need a large LL/SC object of the size of m words.

No real machine supports LL/SC objects in hardware. Fortunately, however, these objects can be efficiently implemented from instructions available on many modern machines. Specifically, MIPS, Alpha and PowerPC support a restricted form of LL/SC, which Moir calls RLL/RSC [16]. Small LL/SC objects (strictly smaller than the size of a word) can be implemented from either RLL/RSC objects [16] or from compare&swap objects [6, 16]. Wordsized or m-word LL/SC objects can then be implemented from the small LL/SC objects, using Anderson and Moir's algorithm [5]. Their algorithm has a space complexity of $O(mn^2)$, which we recently improved as follows:

THEOREM 1 ([11]). *It is possible to implement a m-word LL/SC object, shared by n processes, from $O(mn)$ wordsized compare&swap objects or from $O(mn)$ wordsized RLL/RSC objects. The time complexity of read, LL or SC is $O(m)$.*

By the above theorem, we can implement a wordsized LL/SC object with constant time complexity and constant space per process. Thus, the reliance of this paper's algorithms on LL/SC objects is not a concern.

2. F-ARRAY: DEFINITION AND EXAMPLES

2.1 Type

Every object has a type. A *type* T is a tuple (Q, OP, RES, δ), where Q is a set of states, OP is a set of operations, RES is a set of responses and $\delta : Q \times OP \rightarrow Q \times RES$ is the sequential specification. If $\delta(q, op) = (q', res)$, it means that applying operation op to a type T object in state q causes the object to move to state q' and return the response res.

A type is *readable* if its set of operations includes *read*, which does not modify the state. Specifically, if $\delta(q, read) = (q', v)$, we require $q' = q$, and say v is the *value* of T at state q. For most types, v is the same as q, but in general they are different. Define $vals(T)$ as $\{v \mid q \in Q, v \text{ is the value at } q\}$.

To illustrate the above notation, we specify below the 64-bit LL/SC type for a set $\mathcal{P}$ of processes. For this type, notice that the value and the state are not the same. In the following, let $\mathcal{A}$ be the set of all 64-bit values.

- $OP = (\{\text{LL}, \text{read}\} \cup \{\text{SC(u)} \mid u \in \mathcal{A}\}) \times \mathcal{P}$
- $Q = \{[v, S] \mid S \subseteq \mathcal{P}, v \in \mathcal{A}\}$
- $RES = \mathcal{A}$
- $\delta([v, S], [\text{read}, p]) = ([v, S], v)$
  $\delta([v, S], [\text{LL}, p]) = ([v, S \cup \{p\}], v)$
  $\delta([v, S], [\text{SC}(u), p]) = ([v, S], \mathit{false})$ if $p \notin S$
  $\delta([v, S], [\text{SC}(u), p]) = ([u, \emptyset], \mathit{true})$ if $p \in S$

Another type that we commonly use in this paper is *register*, which is the standard wordsized type that supports read and write operations.

2.2 f-array

Let $T_1, \ldots, T_m$ be any (not necessarily distinct) readable types, where $T_i = (Q_i, OP_i, RES_i, \delta_i)$. Let f be any function whose domain is $vals(T_1) \times \cdots \times vals(T_m)$. Then, $f\text{-}array[T_1, \ldots, T_m]$ is a new readable type (Q, OP, RES, δ), defined as follows:

- $Q = Q_1 \times Q_2 \times \cdots \times Q_m$
- $OP = \{read\} \cup \{(op, i) \mid op \in OP_i, i \in \{1, 2, \ldots, m\}\}$
- $RES = RES_1 \cup RES_2 \cup \cdots RES_m \cup \text{Range}(f)$
- $\delta([q_1, \ldots, q_m], (op, i)) = ([q'_1, \ldots, q'_m], res)$, where $(q'_i, res) = \delta_i(q_i, op)$ and, for all $j \neq i$, $q'_j = q_j$.
- $\delta([q_1, \ldots, q_m], read) = ([q_1, \ldots, q_m], f(v_1, \ldots, v_m))$, where v_i is the value of T_i at q_i.

We present below examples to show that some important objects reduce to f-arrays. Thus, if we design an efficient algorithm for implementing any f-array object, we can implement several other objects efficiently.

EXAMPLE 1. Consider the problem of atomically computing an aggregate of an array of m values, even as these values are dynamically updated by processes. Aggregate functions of common interest include min, max, mean, standard deviation, sum, the number of negative values, whether

the array is sorted, whether all array entries are prime *etc.* Each of these problems trivially reduces to an f-array of m registers, where f is the appropriate aggregation function.

EXAMPLE 2. An m-component multiwriter snapshot object [1, 3] supports $write_i(v)$, which writes v to ith component, and *scan*, which returns the values of all m components. This object is the same as an f-array of m registers, with $f(v_1, \ldots, v_m)$ defined as $[v_1, \ldots, v_m]$, and *scan* identified with the *read* operation on the f-array.

EXAMPLE 3. Consider a generalization of a snapshot object that allows its components to be updated by more complex operations, besides *write*. Specifically, we define $snapshot[T_1, \ldots, T_m]$ as an object that consists of m components, where the ith component can be updated with any operation of type T_i (if the operation is a read-modify-write operation, such as fetch&add, it returns a value to the invoking process besides changing the component's state). The *scan* operation returns the values of the m components, as usual.

It is obvious that $snapshot[T_1, \ldots, T_m]$ object is the same as f-array$[T_1, \ldots, T_m]$, for $f(v_1, \ldots, v_m) = [v_1, \ldots, v_m]$.

EXAMPLE 4. A *counter* supports *read* and $inc(d)$, which adds d to object's value and returns an acknowledgement. We can reduce a counter, shared by n processes, to an f-array $\mathcal{O}$ of n registers, where f is the *sum* function. The reduction is as follows: Process i increments the counter by first reading the value v of the ith component of $\mathcal{O}$ and then writing $v + d$ to it. To read the counter, a process simply reads $\mathcal{O}$. (This reduction works because process i is the only one to access component i.)

EXAMPLE 5. Define *priority process-queue*, a new type that supports *insert*, *findmin* and *delete* operations, with two restrictions: (i) an element can be deleted only by the process that inserted it, and (ii) after a process p inserts an element, p must delete that element before inserting a new one.

A priority process-queue, shared by n processes, reduces to an f-array $\mathcal{O}$ of n registers, where f is the *min* function. Process i inserts v in the queue by writing v to ith component of $\mathcal{O}$, and performs a delete by writing ∞ (a special value considered bigger than any user-supplied value) to the ith component. The *findmin* operation is implemented by reading and returning the value of $\mathcal{O}$.

3. IMPLEMENTATION OF f-ARRAY

Figure 1 describes a wait-free implementation of an object $\mathcal{O}$ of type f-array$[T_1, \ldots, T_m]$, for any function f and any types $T_1, \ldots, T_m$. The algorithm tracks the state of the m components of $\mathcal{O}$ through objects $A_1, \ldots, A_m$, of types $T_1, \ldots, T_m$, respectively. In addition, the algorithm uses an LL/SC object O and ensures that O always holds $\mathcal{O}$'s current value. This makes it trivial to implement the read operation on $\mathcal{O}$: simply return O's value (Lines 1-2).

Shared Variables
A_i: Object of type T_i, $1 \leq i \leq m$
O: LL/SC object, big enough to hold any value in the range of f

```
    procedure read()
1.  v = read(O)
2.  return v

    procedure apply(op, i)
3.  res = apply(A_i, op)
4.  if ¬ refresh()
5.      refresh()
6.  return res

    procedure refresh()
7.  LL(O)
8.  for j := 1 to m
9.      v_j = read(A_j)
10. return SC(O, f(v_1, ..., v_m))
```

Figure 1: Implementing f-array$[T_1, \ldots, T_m]$ object $\mathcal{O}$

To apply an operation op on the ith component of $\mathcal{O}$, a process p first performs op on A_i and receives A_i's response (Line 3). This operation won't however take effect until the change in A_i's state is propagated to O. To make this happen, p calls `refresh()` (Line 4), which requires p to perform LL on O (Line 7), read the values $v_1, \ldots, v_m$ of $A_1, \ldots, A_m$ (Lines 8-9), and then attempt to write in O the value $f(v_1, \ldots, v_m)$ that p believes to be the current value of $\mathcal{O}$ (Line 10). If the refresh succeeds (i.e., SC returns *true*), p is certain that its change to A_i on Line 2 is reflected in O's value. So, p terminates its operation, returning *res* (Line 6). On the other hand, if the refresh fails, p executes refresh once more (Line 5). This refresh may fail also, but its failure implies that some other process q performed a successful refresh and the reading of A_i during q's refresh occurred after p's update of A_i on Line 2. So, regardless of the success of its second refresh, p is certain that its change to A_i on Line 2 is reflected in O's value. So p terminates its operation on Line 6, returning *res*.

THEOREM 2. *Figure 1 presents a linearizable, wait-free implementation of an f-array$[T_1, \ldots, T_m]$ object that can be accessed by an arbitrary number of processes.*

Proof Sketch: We make two key observations that follow from the semantics of LL and SC operations: (1) no two successful refreshes are concurrent, (2) if both refreshes of a process p fail (on Lines 4 and 5), it must be that some process q performs a successful refresh that begins after p's first refresh begins and completes before p's second refresh completes.[3] The algorithm's correctness is based on these observations and the following three rules for linearizing operations on $\mathcal{O}$.

1. Each `read()` operation is linearized to the read operation on O (Line 1).

[3]This property, which follows from LL/SC semantics, was first exploited by Herlihy in the design of a universal construction [9].

2. Consider any apply(op,i) operation σ. If r is the earliest successful refresh to read A_i after σ performs op on A_i, then σ is linearized to r's (successful) SC operation.

3. The previous rule makes it possible for several operations to linearize to the same point. This rule specifies how to order such operations among themselves. Specifically, if σ and σ' are any two operations that are linearized to the same point by the previous rule, σ is ordered before σ' if and only if σ performed Line 3 before σ' performed Line 3. ■

3.1 Applications

The next two results state that snapshot and generalized snapshot, defined in Examples 2 and 3 of Section 2, have efficient implementations whose time complexity is independent of the number of processes. The two theorems have similar proofs, so we just prove the first one.

THEOREM 3. *There is a linearizable, wait-free implementation of a m-word multiwriter snapshot object, shared by n processes, from $O(mn)$ LL/SC words and $O(m)$ registers. A scan operation or a write to a component completes in $O(m)$ primitive operations, where each primitive operation is applied on either an LL/SC word or a register.*

Proof Use the algorithm in Figure 1 to implement the snapshot object as an f-array of m registers, where $f(v_1,\ldots,v_m) = [v_1,\ldots,v_m]$. Implement the m-word LL/SC object required in the algorithm from wordsized LL/SC objects with the help of Theorem 1. ■

THEOREM 4. *There is a linearizable, wait-free implementation of a snapshot$[T_1,\ldots,T_m]$ object, shared by n processes, from LL/SC words and one object each of types $T_1,\ldots,T_m$. If a value of each type T_i fits in a single word, the implementation requires $O(mn)$ LL/SC words; a scan operation or an updating of a component completes in $O(m)$ primitive operations, where each primitive operation is applied on either an LL/SC word or an object of type T_i.*

4. TREE ALGORITHM

4.1 Composing f-arrays

Let $A,B,\ldots,H$ be any types and ψ,ψ',ψ'' be any functions, related as follows: $A = \psi$-array$[B,C]$, $B = \psi'$-array$[D,E]$, and $C = \psi''$-array$[F,G,H]$. The first tree in Figure 2 depicts the tree structure of type A. Suppose we now define a new type A' as ϕ-array$[D,E,F,G,H]$, where $\phi(d,e,f,g,h) = \psi(\psi'(d,e),\psi''(f,g,h))$. What is the relationship between A and A'? As we explain below, they are essentially the same: A' is just a "flattened" version of A.

To precisely state the relationship between the types A and A', we need two definitions. In the following, let $T = (Q, OP, RES, \delta)$ and $T' = (Q', OP', RES', \delta')$. T is a *subtype* of T' if $Q = Q'$, $OP \subseteq OP'$ and, for all $(q,op) \in Q \times OP$, we have $\delta(q,op) = \delta'(q,op)$. Informally, T supports fewer operations, but has identical behavior as T'. Types T and T' are *isomorphic* if $Q = Q'$ and there is a 1-to-1 onto (renaming) function $\alpha : OP \to OP'$ such that for all $(q,op) \in Q \times OP$ we have $\delta(q,op) = \delta'(q,\alpha(op))$. Informally, T and T' are identical, modulo renaming of operations.

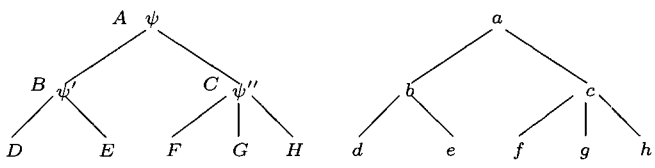

Figure 2: Type tree and object tree

OBSERVATION 1. *Let the types $A,B,\ldots,H$, A' and the functions ψ,ψ',ψ'',ϕ be as specified in the first paragraph of this section. Then, A' is isomorphic to a subtype of A.*

Proof The operations supported by A' are {read}$\cup$ $\{(op,1) \mid op \in OP(D)\} \cup \{(op,2) \mid op \in OP(E)\} \cup \ldots \cup \{(op,5) \mid op \in OP(E)\}$. These operations are renamed in A' as follows: $(op,1)$ of A' is renamed to $((op,1),1)$ of A, $(op,2)$ to $((op,2),1)$, $(op,3)$ to $((op,1),2)$, $(op,4)$ to $((op,2),2)$, and $(op,5)$ to $((op,3),2)$. The read of A' corresponds to read of A.

Type A supports two other operations—(read, 1) and (read, 2), which return the values of the "intermediate" functions ϕ and ϕ'—that have no analogs in A'. This is why A' is a subtype. ■

Thus, if we wish to implement A', we have two options. The first option is to implement ϕ-array$[D,E,F,G,H]$ directly using the algorithm in Figure 1. The second option is to implement A instead of A', which is legitimate by the above observation. Its advantage is that A can be implemented compositionally, as follows: (i) implement $B = \psi'$-array$[D,E]$, (ii) implement $C = \psi''$-array$[F,G,H]$, and then (iii) implement ψ-array$[B,C]$. We can use the algorithm in Figure 1 for each of these three steps. The resulting implementation is depicted by the second tree in Figure 2, where $d,\ldots,h$ are objects of types $D,\ldots,H$, respectively, and a,b,c are LL/SC objects. The subtree at b implements a type B object, the subtree at c implements a type C object, and these are used together with a to implement a type A object. For example, the following steps are performed to execute the operation $(op,2)$, which applies op to the second component of ϕ-array$[D,E,F,G,H]$: apply op to e (let res be the response), propagate the change at e to b (by performing one or two refreshes at b), then propagate the change at b to a (by performing one or two refreshes at a), and finally return res. In other words, an operation on a component of $\mathcal{O}$ is implemented by performing that operation at the appropriate leaf and then traversing up, towards the root, updating each internal node x along the way by performing one or two refreshes at x.

The following lemma generalizes Observation 1.

LEMMA 1. *Let A be the type ψ-array$[A_1,\ldots,A_m]$, where each A_i is the type ψ_i-array$[A_{i,1},\ldots,A_{i,j_i}]$. Let A' be ϕ-array$[A_{1,1},\ldots,A_{1,j_1},A_{2,1},\ldots,A_{2,j_2},\ldots,A_{m,1},\ldots,A_{m,j_m}]$, where ϕ is the function defined as follows:*
$\phi((x_{1,1},\ldots,x_{1,j_1},x_{2,1},\ldots,x_{2,j_2},\ldots,x_{m,1},\ldots,x_{m,j_m}) = \psi(\psi_1(x_{1,1},\ldots,x_{1,j_1}),\ldots,\psi_m(x_{m,1},\ldots,x_{m,j_m}))$. Then, A' is isomorphic to a subtype of A.

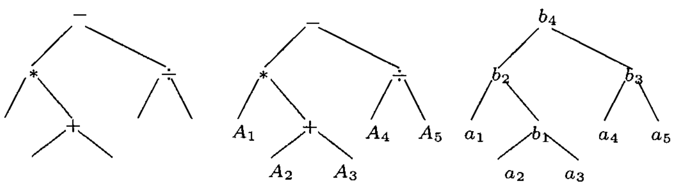

Figure 3: Function tree, type tree and object tree

Proof Similar to the proof of Observation 1. ■

In the above lemma, there is no loss of generality in assuming that each A_i is an array type: if some A_i is not (*e.g.*, A_i is register), we can still regard it as the array type ψ_i-array$[A_i]$, where ψ_i is the identity function.

4.2 Tree Algorithm

In the example of the previous subsection, type A was expressed in terms of B and C which, in turn, were expressed in terms of D, E and F, G, H, respectively. In general, some of D, E, F, G, H may also be f-arrays and this nesting could be arbitrarily deep. In this section, we generalize the ideas of the previous subsection to types that are expressed with arbitrary levels of nesting. The result is a "Tree Algorithm" that implements many types of f-arrays more efficiently than the algorithm in Figure 1.

A *function tree* is a tree where every internal node x is labeled by a function of the same arity as the number of x's children. The first tree in Figure 3 is an example of a function tree. If leaves are assigned values, these values propagate up the tree in a natural way according to the functions labeling the internal nodes. We say a function tree τ that has m leaves *represents function* f *of arity* m if, for every assignment $v_1, \ldots, v_m$ to the leaves of τ, the value that propagates to the root is $f(v_1, \ldots, v_m)$. For instance, the first tree in Figure 3 is a function tree that represents $f(x_1, x_2, x_3, x_4, x_5) = x_1 * (x_2 + x_3) - x_4/x_5$.

A *type tree* is a function tree where each leaf is labeled by a type. For instance, the second tree in Figure 3 is a type tree whose leaves are labeled by types $A_1, \ldots, A_5$. Each node x of a type tree *represents* a type, specified as follows: If x is a leaf, it represents the type that it is labeled by; if x is an internal node labeled by a function f and x has k children representing types $B_1, \ldots, B_k$, then x represents the type f-array$[B_1, \ldots, B_k]$. We say a type tree represents type T if its root represents T. For example, the second tree in Figure 3 represents the type minus-array$[B, C]$, where B is product-array$[A_1, D]$, D is sum-array$[A_2, A_3]$ and C is div-array$[A_4, A_5]$. It is easy to see (by repeated application of Lemma 1) that the type represented by this tree is essentially the same as the type f-array$[A_1, A_2, A_3, A_4, A_5]$, where $f(x_1, x_2, x_3, x_4, x_5)$ is the function $x_1 * (x_2 + x_3) - x_4/x_5$. This fact is captured by the next lemma.

LEMMA 2. *Let τ be any type tree, A be the type represented by τ, and $A_1, \ldots, A_m$ be the types labeling the leaves of τ. Let f be the function represented by the function tree corresponding to τ. Then, f-array$[A_1, \ldots, A_m]$ is isomorphic to a subtype of A.*

Shared Objects

τ': Object tree corresponding to a type tree τ

```
procedure read()
v = read(root(τ'))
return v

procedure apply(op, i)
Let L_i be the ith leaf of τ'
res = apply(L_i, op)
currentNode = L_i
repeat
    currentNode = parent(currentNode)
    if ¬refresh(currentNode)
        refresh(currentNode)
until currentNode = root(τ')
return res

procedure refresh(currentNode)
Let O_1, ..., O_k be currentNode's children
and g be the function labeling the node in τ
corresponding to currentNode
LL(currentNode)
for i := 1 to k
    v_i = read(O_i)
return SC(currentNode, g(v_1, ..., v_k))
```

Figure 4: Tree Algorithm

Proof By repeated application of Lemma 1. ■

By the above lemma, implementing f-array$[A_1, \ldots, A_m]$ reduces to implementing the type represented by the root of τ, which is accomplished by the following obvious algorithm: implement the types represented by the internal nodes of τ bottom-up, using the algorithm in Figure 1. We call this the *Tree Algorithm* and present below a concrete description of it.

The *object tree* τ' corresponding to a type tree τ is defined as follows: (i) τ' has the same tree structure as τ, (ii) if a leaf of τ is labeled by type T, the corresponding leaf of τ' has an object of type T, and (iii) if an internal node of τ is labeled by a function g, the corresponding node of τ' has an LL/SC object that is large enough to hold any value in the range of g. The second and third trees in Figure 3 depict a type tree and the corresponding object tree (assuming that $a_1, \ldots, a_5$ are objects of types $A_1, \ldots, A_5$, respectively, and $b_1, \ldots, b_4$ are LL/SC objects).

THEOREM 5 (TREE ALGORITHM). *Let τ be any type tree, A be the type represented by τ, and $A_1, \ldots, A_m$ be the types labeling the leaves of τ. Let f be the function represented by the function tree corresponding to τ, and τ' be the object tree corresponding to τ. Then, we have the following:*

(1). The algorithm in Figure 4 is a linearizable, wait-free implementation of an f-array$[A_1, \ldots, A_m]$ object that can be accessed by an arbitrary number of processes.

(2). If d is the maximum degree of any internal node of τ and A_i is at depth k, then `apply`(op, i) *performs at most $O(kd)$ operations on primitive objects.*

Proof Follows from Lemma 2 and the algorithm in Figure 1. ■

Afek, Dauber and Touitou [2] designed a similar, but a less general algorithm for computing the list of active processes (in the context of a universal construction). As we now describe, our Tree algorithm is more general than theirs in three ways and each generalization has made our algorithm more useful. First, [2] works on a binary tree while the shape of the tree is unrestricted in our algorithm. This feature has made possible the snapshot implementation, stated in Theorem 3. Second, in [2], every internal node computes the same function, *viz.*, the union of the lists at its children. Our algorithm, on the other hand, allows internal nodes to compute arbitrary (and different) functions. As we will see in the next section, this feature makes our algorithm useful for diverse problems: we employ the sum function to implement a counter and the min function to implement a priority queue. The final difference is that, in [2], the array components (*i.e.*, leaves) are registers. In our algorithm each component may be of any type, may be accessed by any number of processes and, when updated, the component not only undergoes a change of state, but also returns a response (Line 10). This feature has made possible the generalized snapshot implementation, stated in Theorem 4.

5. APPLICATIONS OF F-ARRAYS

In this section, we use Tree Algorithm to obtain efficient wait-free implementations of some useful objects.

5.1 Computing array aggregates

The following theorem lists some common aggregates that can be computed efficiently. The algorithm is efficient: reading the aggregate value takes only constant time, and the time to write to a component is logarithmic in array size. A significant feature of the algorithm is that it supports an arbitrary number of processes and its time complexity is independent of the number of processes.

THEOREM 6. *Consider the following list of functions:*

(a). $f(v_1, \ldots, v_m) = v_1 + \ldots + v_m$
(b). $f(v_1, \ldots, v_m)$ *is minimum (resp., maximum) of* $v_1, \ldots, v_m$
(c). $f(v_1, \ldots, v_m)$ = *mean of* $v_1, \ldots, v_m$
(d). $f(v_1, \ldots, v_m)$ = *standard deviation of* $v_1, \ldots, v_m$
(e). $f(v_1, \ldots, v_m)$ *is true if and only if* $v_1, \ldots, v_m$ *are in increasing order.*
(f). $f(v_1, \ldots, v_m)$ *is true if and only if all of* $v_1, \ldots, v_m$ *are primes.*

For any function f listed above, we can implement an f-array $\mathcal{O}$ of m registers. The implementation is linearizable, wait-free and can be accessed concurrently by an arbitrary number of processes. $\mathcal{O}$ is implemented from $O(m)$ registers and LL/SC objects. The time complexity of read is $O(1)$ and of writing to a component is $O(\log m)$.

Proof For simplicity, assume that m is a power of 2. In each case, we show that the function is represented by a complete binary function tree τ that has m leaves and $\log m$ height. This, together with Theorem 5, implies the result.

For Part (a), let each internal node of τ compute the sum of its two children. Then, τ represents the function in Part (a). For Part (b), let each internal node of τ compute the minimum (resp., maximum) of its two children. The remaining cases are also easy and we omit their proofs. ■

5.2 Implementing adaptive counter

Example 4 of Section 2 describes how to implement a counter, shared by n processes, using a sum-array of n registers. Combining this with the implementation of the sum-array in Part (a) of Theorem 6, we obtain a counter implementation with a time complexity of $O(1)$ for the read and $O(\log n)$ for the increment operation. In this section, we present a more sophisticated implementation that achieves *adaptive* time complexity, stated as follows.

In the context of a wait-free implementation, a process is considered *active* if it started executing an operation on the implementation, but has not yet completed it. *Contention at time t*, denoted $c(t)$, is the number of active processes at t. *Contention during an interval of time* is the maximum, over all t in the interval, of $c(t)$ (this definition of contention is known as *point contention*). We present a counter implementation whose the time complexity is $O(1)$ for the read and $O(\min(k, \log n))$ for the increment, where k is the contention during the increment operation. To achieve this result, we use adaptive renaming, which is described next.

5.2.1 Adaptive renaming

Afek, Dauber and Touitou designed an adaptive n-process longlived renaming algorithm [2]. In Figure 5 we present a minor variation of their algorithm in which an attempt by a process p to acquire a name might fail if too many processes compete. However, if p's attempt succeeds, the acquired name is guaranteed to be very small. This algorithm will be useful in our counter implementation.

When a process p wishes to acquire a name, it executes *AcquireName()*. This procedure either fails (i.e., returns $\perp$ by Line 6) or successfully returns a name in the range $\{1, \ldots, \log n\}$ (by Line 5), where n is the number of processes that the algorithm is designed to handle. After p acquires a name x, it must release the name by executing *ReleaseName*(x), before again attempting to acquire a name. *Process p owns name x* from the point when *AcquireName()* returns x to the point when p subsequently initiates *ReleaseName*(x).

We consider a process p to be *active* during each interval of time lasting from the initiation of a successful *AcquireName()* by p to the completion of the matching *ReleaseName* by p.

LEMMA 3 ([2]). *The algorithm in Figure 5 satisfies two properties:*

1. *Each name in $\{1, \ldots, \log n\}$ is owned by at most one process at any time.*
2. *Consider an execution of AcquireName() by p. If p completes k iterations of the for-loop without acquiring a name in the range $\{1, 2, \ldots, k\}$, then at some instant*

Shared Variables

free: **array** $[1 \cdots \log n]$ **of boolean**
(Initially *free*$[i]$ is *true*, for all $1 \leq i \leq \log n$.)

procedure AcquireName()
1. **for** $i := 1$ **to** $\log n$
2. $\quad$ *available* := LL(*free*$[i]$)
3. $\quad$ **if** *available*
4. $\quad\quad$ **if** SC(*free*$[i]$, *false*)
5. $\quad\quad\quad$ **return** i
6. **return** $\perp$

procedure ReleaseName(i)
7. *free*$[i]$ = *true*

Figure 5: A minor variation of Afek, Dauber and Touitou's renaming algorithm

during the first k iterations there were at least k active processes excluding p.

5.2.2 *Adaptive counter*

We now describe our adaptive algorithm for implementing an n-process counter. The algorithm is based on an array object $\mathcal{O}$ with the following properties:

P1. $\mathcal{O}$ has $n + \log n$ components.

P2. The read operation on $\mathcal{O}$ returns the sum of all $n+\log n$ components.

P3. Writing to any of the first $\log n$ components of $\mathcal{O}$ is much faster than writing to any of the last n components. Specifically, it takes $O(\log n)$ time to write to any of the last n components, but it takes only $O(\log k)$ time to write to the kth component, for all $k \in \{1, \ldots, \log n\}$.

In the following, first we show how to implement $\mathcal{O}$ from LL/SC objects, and then describe how to implement an adaptive counter from $\mathcal{O}$.

The key to implementing $\mathcal{O}$ is the function tree τ, defined as follows. Consider a complete binary tree of n leaves. Add $\log n$ more leaves to this tree by attaching one leaf to each of the first $\log n$ internal nodes, considered in breadth-first order. Number the leaves of this tree 1 through $n + \log n$, in the breadth-first order. Turn this tree into a function tree by letting each internal node compute the *sum* function (the resulting function tree τ is depicted in Figure 6 for $n = 8$). Use this function tree τ in the Tree Algorithm to implement an f-array of $n + \log n$ registers, where f is the sum function. Let $\mathcal{O}$ denote the implemented array object. It is obvious that $\mathcal{O}$ satisfies properties P1-P3, stated earlier. Also, by Tree Algorithm, the implementation of $\mathcal{O}$ requires $n+\log n$ registers and $O(n)$ LL/SC objects, for a total space complexity of $O(n)$.

Next we describe how to implement a counter from $\mathcal{O}$. The main idea is to designate the ownership of the last n components of $\mathcal{O}$ to the n processes: process p owns (*i.e.*, has exclusive access to) the component numbered $p+\log n$. The first $\log n$ components, on the other hand, have no fixed owners: processes have to compete to capture them.

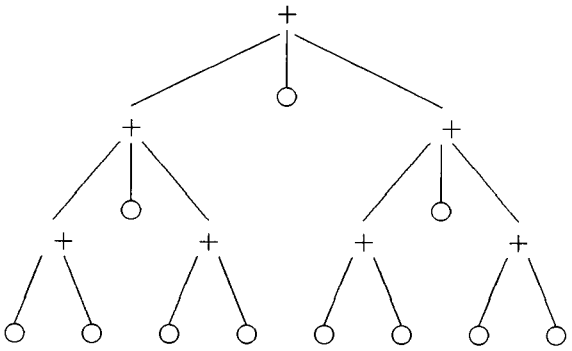

Figure 6: Function tree τ for adaptive counter

Shared Variables

$\mathcal{O}$: sum-array of $n + \log n$ components (with fast access (to first $\log n$ components, as explained in the text)

procedure inc(d)
1. j = AcquireName()
2. **if** $j = \perp$ **then** $j = p + \log n$
3. v = read($\mathcal{O}, j$)
4. write($\mathcal{O}, j, v + d$)
5. **if** $j \leq \log n$ **then** ReleaseName(j)

procedure read()
6. v = read($\mathcal{O}$)
7. **return** v

Figure 7: Implementing adaptive counter: Algorithm for process p

Figure 7 presents an implementation of a counter from $\mathcal{O}$. A process p increments the counter by d by gaining exclusive access to a component of $\mathcal{O}$, reading the value v of that component and then writing $v + d$ to that component. To minimize the time complexity, p attempts to capture the earliest among the first $\log n$ components of $\mathcal{O}$ (Line 1). If p's attempt succeeds, p increments the component j that it captured; otherwise p increments the component that it owns (Line 2). In either case, p has exclusive access to component j. So, the reading and writing on Lines 3-4 amount to atomically incrementing the component j and, hence, the value of $\mathcal{O}$ (which is the sum of all components). Finally, if, on Line 1, p gained exclusive access to one of the first $\log n$ components, it gives up that privilege on Line 5.

To read the counter, p simply reads $\mathcal{O}$ and returns that value (Lines 6-7).

We now argue that the time complexity of incrementing the counter is $O(\min(k, \log n))$, where k is the contention during the increment operation. We consider two cases: either p acquires a name $j \leq \log n$ on Line 1 or it does not. In the former case, by Property 2 of Lemma 3, the contention k is at least j. Lines 1, 2, 3, 4 and 5 take j, 1, 1, $O(\log j)$ and 1 steps, respectively, for a total of $O(j)$ steps. Since

Shared Variables
$\mathcal{O}$: min-array of $n + \log n$ components (with fast access (to first $\log n$ components, as explained in the text)

Local persistent variable at $p \in \{1, \ldots, n\}$
loc_p: $1 \cdots n + \log n$

procedure `insert`(v)
1. loc_p = AcquireName()
2. **if** $loc_p = \perp$ **then** $loc_p = p + \log n$
3. write$(\mathcal{O}, loc_p, v)$

procedure `delete`()
4. write$(\mathcal{O}, loc_p, \infty)$
5. **if** $loc_p \leq \log n$ **then** ReleaseName(loc_p)

procedure `findmin`()
6. v = read$(\mathcal{O})$
7. **return** v

Figure 8: Implementing priority process-queue: Algorithm for process p

$j \leq k$ and $j \leq \log n$, it follows that the time complexity is $O(\min(k, \log n))$.

In the other case where AcquireName() returns $\perp$ on Line 1, by Property 2 of Lemma 3, the contention k is at least $\log n$. Lines 1, 2, 3, 4 and 5 take $\log n$, 1, 1, $O(\log n)$ and 1 steps, respectively, for a total of $O(\log n)$ steps. Since $k \geq \log n$, it follows that the time complexity is $O(\min(k, \log n))$. Based on the above discussion, we have:

THEOREM 7. *A linearizable, wait-free implementation of a counter, shared by n processes, is possible from registers and LL/SC words. The time complexity of* read *is* $O(1)$ *and the time complexity of* inc(d) *is* $O(\min(k, \log n))$, *where k is the contention during the increment operation. The space complexity is* $O(n)$.

5.3 Adaptive priority process-queue

In this section, we present an adaptive algorithm for implementing the *priority process-queue* object defined in Example 5 of Section 2. The algorithm is similar to the adaptive counter algorithm of the last section, so in the following we only point out the differences.

As before, our priority queue algorithm is based on an array object $\mathcal{O}$ that has properties P1-P3, except that P2 is different: the read operation on $\mathcal{O}$ returns the minimum of the $n + \log n$ components. This difference is reflected in the function tree τ by requiring each internal node of τ to compute the *min* function. As before, we use the function tree τ in Tree Algorithm to implement $\mathcal{O}$.

Figure 8 describes the implementation of a priority process-queue object from $\mathcal{O}$. The algorithm and its time complexity analysis are based on similar ideas as before. The next theorem summarizes the result. We make two remarks on how the time complexity is stated in the theorem. First, the time complexity is stated for the pair—insert followed by delete—and not for each operation individually. This makes sense because the object's specification requires a process to delete any element that it inserted before performing a new insert. The second remark concerns how we measure contention, which is the number of active processes. We consider a process to be active from the time it initiates an insert operation up to the time it completes the matching deletion.

THEOREM 8. *A linearizable, wait-free implementation of a priority process queue, shared by n processes, is possible from registers and LL/SC words. The time complexity of findmin is* $O(1)$ *and of a matching pair of operations—insert followed by delete—is* $O(\min(k, \log n))$, *where k is the contention during the execution of insert. The space complexity is* $O(n)$.

6. FCFS & PRIORITY MUTUAL EXCLUSION

In this section, we present an n-process starvation-free mutual exclusion algorithm that satisfies First-Come-First-Served (FCFS) property and is local spinning. Later we show how to accomodate process priorities into the algorithm. Our algorithm has an adaptive time complexity of $O(\min(k, \log n))$ for both cache coherent and NUMA architectures, where k is the contention. Interestingly, the wait-free implementations of counter and priority process-queue (from the previous section) are the ones that have made this locking algorithm possible.

The FCFS property requires the entry section to begin with a bounded section of code called the *doorway* [13]. If a process p completes executing the doorway before q enters the doorway, the FCFS property requires that q may not enter the critical section (CS) before p.

Our algorithm, presented in Figure 9, is described informally as follows. When a process p wants to enter the CS, it sets its wait flag (Line 1) and proceeds to obtain a token by incrementing and then reading the counter C (Lines 2-3). Then, it inserts its name, tagged with its token, into the priority queue Q (Line 4). The code up to Line 4 constitutes the bounded doorway. After the doorway, p executes `promote` procedure (Line 5) whose job is to seize the CS for the highest priority waiting process q in Q and inform q that it no longer needs to wait. Following this, p busy-waits until it is informed that it no longer needs to wait (Line 6). Before entering the CS, p removes itself from the queue (Line 7). When done with the CS, p makes it known that the CS is free (Line 8), and then executes `promote` procedure (Line 9) to ensure that, in the event that there are waiting processes, the highest priority process is advanced to CS.

We now describe the `promote` procedure, which advances the highest priority waiting process in Q to the CS, if the CS is free. A process p executing `promote` first checks if the CS is busy (Line 10). If it is, p simply returns. Otherwise, p applies *findmin* operation to learn the name q of the process in the queue with the smallest token and hence, the highest priority (Line 11). If the queue is not empty, p attempts to capture the CS for q (Lines 12-13). If p succeeds, it informs q that q no longer needs to wait to enter the CS (Line 14).

THEOREM 9. *The algorithm in Figure 9 satisfies mutual exclusion, FCFS and is starvation-free. If the the vari-*

```
Type
    QueueItem: record pid: 1 ··· n; token: integer end

Shared variables
    C: Counter, initially 0
    Q: Priority Process-Queue, initially empty
    wait: array [1 ··· n] of QueueItem;
    csbusy: boolean, initially false

1.  wait[p] := true
2.  inc(C, 1)
3.  v = read(C)
4.  insert(Q, [p, v])
5.  promote()
6.  wait until wait[p] is false
7.  delete(Q)
    CRITICAL SECTION
8.  csbusy = false
9.  promote()

    procedure promote()
10. if LL(csbusy) return
11. e = findmin(Q)
12. if e ≠ ∞
13.     if SC(csbusy, true)
14.         wait[e.pid] = false
```

Figure 9: Adaptive FCFS mutual exclusion

ables C and Q are implemented according to Theorems 7 and 8, respectively, the algorithm has a time complexity of $O(\min(k, \log n))$*, where k is the contention during p's doorway. The algorithm requires* $O(n)$ *shared variables, where each variable is either a register or a LL/SC word.*

6.1 Incorporating process priorities

In a mutual exclusion algorithm that accommodates process priorities, waiting processes enter the CS in the order of their priority; processes with the same priority enter in the FCFS order, as before. Significantly, we can obtain such an algorithm by making a very minor modification to the algorithm in Figure 9. On Line 4, require p to insert a triple $[p, v, v']$, where v' is p's priority, and define the "less than" relation as follows: $[p, v, v'] < [q, u, u']$ if and only if one of the following three conditions holds: (i) $v' > u'$, or (ii) $v' = u'$ and $v < u$, or (iii) $v' = u'$, $v = u$ and $p < q$.

Acknowledgement

We thank an anonymous PODC referee for valuable comments that helped improve the presentation, and Michael Fromberger and Srdjan Petrovic for a careful reading of an earlier draft.

7. REFERENCES

[1] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. Atomic snapshots of shared memory. In *Proceedings of the 9th Annual Symposium on Principles of Distributed Computing* (1990), pp. 1–14.

[2] Afek, Y., Dauber, D., and Touitou, D. Wait-free made fast. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing* (1995), pp. 538–547.

[3] Anderson, J. Multi-writer composite registers. *Distributed Computing 7*, 4 (1994), 175–195.

[4] Anderson, J., and Kim, Y. J. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing* (October 2000), pp. 29–43.

[5] Anderson, J., and Moir, M. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms* (1995), pp. 168–182.

[6] Anderson, J., and Moir, M. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (August 1995), pp. 184–194.

[7] Craig, T. S. Queuing spin-lock algorithms to support timing predictability. In *Proceedings of the Real Time Systems Symposium* (1993), pp. 148–157.

[8] Herlihy, M. Wait-free synchronization. *ACM TOPLAS 13*, 1 (1991), 124–149.

[9] Herlihy, M. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems 15*, 5 (1993), 745–770.

[10] Herlihy, M., and Wing, J. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS 12*, 3 (1990), 463–492.

[11] Jayanti, P. Efficient constructions of LL/SC variables. Unpublished manuscript, February 2002.

[12] Johnson, T., and Harathi, K. A prioritized multiprocessor spin lock. *IEEE Transactions on Parallel and Distributed Systems 8*, 9 (September 1997), 926–933.

[13] Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*, 8 (1974), 453–455.

[14] Lamport, L. Concurrent reading and writing. *Communications of the ACM 20*, 11 (1977), 806–811.

[15] Markatos, E. Multiprocessor synchronization primitives with priorities. In *Proceedings of the 1991 IFAC Workshop on Real-Time Programming* (1991), Pergamon Press, pp. 1–7.

[16] Moir, M. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing* (August 1997), pp. 219–228.

[17] Peterson, G. L. Concurrent reading while writing. *ACM TOPLAS 5*, 1 (1983), 56–65.