Efficient Fetch-and-Increment*

Faith Ellen[1], Vijaya Ramachandran[2], and Philipp Woelfel[3]

[1] University of Toronto
faith@cs.toronto.edu
[2] University of Texas at Austin
vlr@cs.utexas.edu
[3] University of Calgary
woelfel@ucalgary.ca

Abstract. A Fetch&Inc object stores a non-negative integer and supports a single operation, fi, that returns the value of the object and increments it. Such objects are used in many asynchronous shared memory algorithms, such as renaming, mutual exclusion, and barrier synchronization. We present an efficient implementation of a wait-free Fetch&Inc object from registers and load-linked/store-conditional (ll/sc) objects. In a system with p processes, every fi operation finishes in $O(\log^2 p)$ steps, and only a polynomial number of registers and $O(\log p)$-bit ll/sc objects are needed. The maximum number of fi operations that can be supported is limited only by the maximum integer that can be stored in a shared register. This is the first wait-free implementation of a Fetch&Inc object that achieves both poly-logarithmic step complexity and polynomial space complexity, but does not require unrealistically large ll/sc objects or registers.

1 Introduction

A Fetch&Inc object stores a non-negative integer and supports a single operation, fi, that returns the value of the object and increments it. Such objects are fundamental synchronization primitives which have applications in many asynchronous shared memory algorithms. For example, a one-shot Fetch&Inc object, which allows at most one fi operation per process, can be used to solve the one-shot renaming problem: assign unique names from a small name space to participating processes. Each participating process performs fi and uses the result as its name. Thus, if k processes participate, they get unique names in the optimal range $\{0, \ldots, k-1\}$. Fetch&Inc objects have also been used in algorithms for mutual exclusion [5], barrier synchronization [10], work queues [11], and producer/consumer buffers [12,6].

We consider *wait-free*, *linearizable* implementations of Fetch&Inc objects in the standard asynchronous shared memory system with p processes with unique

* This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Grant CCF-0830737 of the US National Science Foundation (NSF).

M.K. Aguilera (Ed.): DISC 2012, LNCS 7611, pp. 16–30, 2012.

identifiers, $1, \ldots, p$. Wait-freedom means that each FETCH&INC operation finishes within a finite number of (its own) steps. Linearizability imposes the condition that when some instance *op* of FI returns the value v, the total number of completed FI operations (including *op*) is at most $v + 1$, and the total number of completed and pending FI operations is at least $v + 1$.

FETCH&INC objects have consensus number two, which means that they can be used to solve wait-free consensus for two processes, but not three. It is not possible to implement FETCH&INC objects just from registers. This is in contrast to *weak counter* objects, which support two separate operations, INCREMENT and READ, where INCREMENT increases the value of the counter by one but does not return anything, and READ returns the counter value. Unlike FETCH&INC objects, weak counters have wait-free implementations from registers. Our FETCH&INC implementation also supports a READ operation that returns the object value and, thus, is strictly stronger than a weak counter.

To implement FETCH&INC objects, the system needs to provide primitives of consensus number at least two. Implementations from TEST&SET and SWAP objects exist [2], but are inefficient. In fact, a lower bound by Jayanti, Tan, and Toueg [17] implies that for any weak counter implementation from resettable consensus and arbitrary history-less objects (and thus from TEST&SET and SWAP objects), some operations may require $\Omega(p)$ shared memory accesses. However, non-linearizable counters, such as those obtained from counting networks [6], can be more efficient. But a linear lower bound on the depth of linearizable counting networks [15] shows that such networks cannot be used to obtain efficient linearizable FETCH&INC implementations.

We consider implementations of FETCH&INC from load-linked/store-conditional (LL/SC) objects. An LL/SC object O provides three operations: LL, VL, and SC. LL(O) returns the value of object O. VL(O) returns TRUE or FALSE and, like LL(O), does not change the value of the object. SC(O, x) either sets the value of object O to x and returns TRUE or does not change the value of O and returns FALSE. A VL(O) or SC(O, x) operation by process p returns TRUE (in which case, we say that it is successful) if and only if p previously executed LL(O) and no other process has executed a successful SC on object O since p's last LL(O).

LL/SC objects allow implementations of any properly specified object using universal constructions. However, such generic universal constructions are not efficient. For example, Herlihy's standard universal constructions [13,14] require $\Omega(p)$ steps per implemented operation. As pointed out by Jayanti [16], the universal construction by Afek, Dauber and Touitou [1] can be modified so that each implemented operation takes only $O(\log p)$ steps, which is optimal. But this requires that registers can hold enough information to describe p operations. Since the description of an operation includes the identifier of the process that is executing the operation, $\Omega(p \log p)$-bit registers are necessary. Thus, this construction is impractical for systems with many processes. There are also efficient randomized FETCH&INC implementations (e.g., Alistarh etal. presented one based on repeated randomized renaming [3]), but there seems to be no obvious way to derandomize them.

In this paper, we present two FETCH&INC implementations that have polylogarithmic (in p) step complexity and do not require unrealistically large registers or LL/SC objects. In particular, $O(\log p)$ bits suffice for each LL/SC object and registers just need to be large enough to store the value of the FETCH&INC object. Our first implementation, presented in Section 2, is efficient when the number of FI operations, n, is polynomial in the number of processes. Each FI operation finishes in $O\big((\log p)(\log n)\big)$ steps, and a total of $O\big(p+n(\log p)(\log n)\big)$ shared registers and LL/SC objects are used. Then, in Section 3, we will explain how to extend this implementation (using a memory compression technique) to improve the worst case step complexity to $O\big((\log p)^2\big)$, using $O(p^3)$ shared registers and LL/SC objects. Both of our implementations support a READ operation with constant step complexity.

2 The First Implementation

The idea of our first implementation is that processes cooperate to construct (an implicit representation of) a sequence of process identifiers. The sequence has one copy of i for each instance of FI that process i performs. The values returned by these instances are the positions of i within this sequence, in increasing order.

The main data structure is a fixed balanced binary tree τ with p leaves, one per process, and height $\lceil \log p \rceil$. The representation of τ doesn't matter. For example, it can be stored implicitly in an array, like a binary heap. Let $P(v)$ denote the set of ids of processes whose leaves are in the subtree rooted at node v. At each node, v, there is an implicit representation of a sequence, $C(v)$, of ids in $P(v)$. Initially, $C(v)$ is empty. The sequence $C(v)$ at an internal node is an interleaving of a prefix of the sequence $C(left(v))$ at its left child and a prefix of the sequence $C(right(v))$ at its right child.

To perform FI, process i appends i to the sequence at process i's leaf. Then process i proceeds up the tree, trying to propagate information about the sequence at the current node, v, and the sequence at its sibling to its parent, as in [1]. It combines the current information at v and $sibling(v)$ and then tries to change $parent(v)$ so that it contains this updated information. If it doesn't succeed, it tries again. If it doesn't succeed a second time, it is guaranteed that some other process has already propagated the necessary information to $parent(v)$. Process i determines the position of its instance in $C(parent(v))$, the sequence at the parent of its current node, from the position of its instance in $C(v)$ and the number of elements from its $sibling(v)$ that precede the block containing its instance. Then process i moves to $parent(v)$. When process i reaches the root, it returns the position of its instance in the sequence at the root. The sequence $C(root(\tau))$ provides a linearization of all completed instances of FI and at most one uncompleted instance of FI by each process.

The sequence at process i's leaf is represented by a single-writer register, N_i, containing the length of the sequence. Thus $N_i = 0$ if the sequence at this leaf is empty. To append i to the sequence at this leaf, process i simply increments the value of N_i.

For any internal node, v, let $N(v)$ denote the length of the sequence $C(v)$ at v, let $NL(v)$ denote the number of elements of $C(v)$ whose leaves are in v's left subtree, and $NR(v)$ denote the number of elements of $C(v)$ whose leaves are in v's right subtree. Then $N(v) = NL(v) + NR(v)$. The sequence $C(v)$ can be implicitly represented by a sequence $I(v)$ of pairs $(side_j, size_j) \in \{L, R\} \times \mathbb{Z}^+$. Specifically, suppose the sequence $C(v)$ consists of h blocks $\ell_1, \ldots, \ell_h$ of size $x_1, \ldots, x_h$ interleaved with k blocks $r_1, \ldots, r_k$ of size $y_1, \ldots, y_k$, where $\ell_1 \cdots \ell_h$ is a prefix of the sequence $C(left(v))$ at the left child of v and $r_1 \cdots r_h$ is a prefix of the sequence $C(right(v))$ at the right child of v. Then $I(v)$ is an interleaving of the two sequences of pairs $[(L, x_1), \ldots, (L, x_h)]$ and $[(R, y_1), \ldots, (R, y_k)]$, where $(side_j, size_j) = (L, x_i)$, if the j'th block of $C(v)$ is ℓ_i, and $(side_j, size_j) = (R, y_i)$ if the j'th block of $C(v)$ is r_i. Note that $NL(v) = x_1 + \cdots + x_h$ and $NR(v) = y_1 + \cdots + y_k$. For example, if

$$\begin{aligned} C(left(v)) &= [5, 3, 1, 2], \\ C(right(v)) &= [9, 10, 15, 12, 15], \text{ and} \\ I(v) &= [(L, 1), (R, 3), (L, 3)], \end{aligned}$$

then $C(v) = [5, 9, 10, 15, 3, 1, 2]$, with $h = 2$, $\ell_1 = [5]$, $x_1 = 1$, $\ell_2 = [3, 1, 2]$, $x_2 = 3$, $k = 1$, $r_1 = [9, 10, 15]$, and $y_1 = 3$. If two consecutive blocks of $I(v)$ have the same side, they can be combined into one block, whose size is the sum of the sizes of those two blocks, without changing the sequence $C(v)$ it represents. Thus, we may assume, without loss of generality, that the sides of the blocks in $I(v)$ alternate between L and R.

Each internal node v has an LL/SC object $v.T$ containing a pointer into a persistent data structure T_v representing versions of the sequence $I(v)$ and, hence implicitly, the sequence $C(v)$. This data structure supports one update operation and two query operations. Here t is a pointer into T_v that indicates one version I of $I(v)$.

APPEND(t, x, y): return a pointer to a new version of $I(v)$ obtained from I by appending the pairs (L, x) and (R, y) to it, as appropriate. More specifically, if the last block of I is (L, z), update that block to $(L, z + x)$ and, if $y \neq 0$, append the pair (R, y). If the last block of I is (R, z), update that block to $(R, z + y)$ and, if $x \neq 0$, append the pair (L, x). When I is empty, append (L, x), if $x \neq 0$, and append (R, y), if $y \neq 0$.

Note that, if I is nonempty when $APPEND(t, x, y)$ is called, then the new sequence either has the same length as I or length one greater. If there are two pairs to append, the pair from the same side as the last pair in I is appended first. Two query operations are also supported.

BLOCKSUM(t, s, j): among the first j blocks of I, return the sum of the sizes of those blocks with first component s, i.e. BLOCKSUM$(t, s, j) = \sum\{size_h \mid side_h = s \text{ and } 1 \leq h \leq j\}$,

FINDBLOCK(t, s, m): return the minimum j with BLOCKSUM$(t, s, j) \geq m$.

Local variables:
v: a node in τ
$s \neq s'$: elements of $\{L, R\}$
t, t': pointers to nodes in T_v
j, m, h, k: nonnegative integers.

```
1   m ← READ(N_i) + 1
2   WRITE N_i ← m
3   v ← process i's leaf (in τ)
4   while v ≠ root(τ) do
5         if v = left(parent(v))
6         then s ← L
7               s' ← R
8         else  s ← R
9               s' ← L
10        v ← parent(v)
          %t is a pointer to the current root of T_v
11        t ← LL(v.T)
          %Check whether i's instance of FI has reached v,
          %i.e. C(v) contains at least m elements from side s
12        while READ(t.Ns) < m do
                %Compute the length h of C(left(v))
13              if left(v) is a leaf of τ
14              then h ← READ(N_j), where j is the index of this leaf
15              else  t' ← READ(left(v).T)
16                    h ← READ(t'.NL) + READ(t'.NR)
                %Compute the length k of C(right(v))
17              if right(v) is a leaf of τ
18              then k ← READ(N_j), where j is the index of this leaf
19              else  t' ← READ(right(v).T)
20                    k ← READ(t'.NL) + READ(t'.NR)
                %Compute a pointer t' to an updated version T' of T_v
21              t' ← APPEND(t, h − t.NL, k − t.NR)
                %Try to update v.T to point to T'
22              SC(v.T, t')
23              t ← LL(v.T)
          end while
          %Compute the position of i's current instance in C(v)
          %by finding the block j that contains the m'th element
          %from side s and the sum of all previous blocks with side s'
24        j ← FINDBLOCK(t, s, m)
25        m ← m+ BLOCKSUM(t, s', j − 1)
    end while
26  return m − 1
```

Fig. 1. Algorithm for FI performed by process i

Initially, $N_i = 0$, for $i = 1, \ldots, p$, and the sequences represented at every node are empty. Pseudocode for FI appears in Figure 1.

A persistent augmented balanced binary tree [9,8], such as a red-black tree or an AVL tree, is used to implement T_v. Each pair in the sequence $I(v)$ is represented by a node containing the side and the size of the pair. Nodes also contain pointers to their left and right children and balance information. They do not contain parent pointers. Each node u of T_v is augmented with the number of nodes in its subtree, the sum $u.NL$ of the sizes of the pairs in its subtree that have side L, and the sum $u.NR$ of the sizes of the pairs in its subtree that have side R. In particular, if T_v is nonempty, then $v.T$ is an LL/SC object that points to the root of a tree in T_v representing the sequence $I(v)$, $root(T_v).NL$ stores $NL(v)$, and $root(T_v).NR$ stores $NR(v)$. Initially, $v.T = nil$ and $NL(v) = NR(v) = 0$.

Processes do not change any information in nodes of T_v once they have been added to the data structure. Instead, when performing APPEND, they create new nodes containing the updated information. Thus, all of the ancestors of a changed node must also be changed. Although APPEND changes T_v, it does not affect $I(v)$ until $v.T$ is changed to the pointer it returns.

If $t = nil$, APPEND(t, x, y) creates a new tree containing (L, x), if $x \neq 0$, and (R, y), if $y \neq 0$. If $t \neq nil$, then APPEND(t, x, y) starts at the root in T_v pointed to by t and follows the rightmost path of its tree, making a copy of each node it encounters and pushing the copy onto a stack. If the side of the rightmost node in the tree is L, then x is added to its size and, if $y \neq 0$, the right child pointer of this node is changed from NIL to a new leaf that contains the element (R, y). Otherwise, y is added to its size and, if $x \neq 0$, the right child pointer of this node is changed from NIL to a new leaf that contains the element (L, x). Then, the stack is popped to progress back up the tree. As each node is popped, its right pointer is set to the root of the updated subtree. The information at the node, including its balance, is updated and rotations are performed, if necessary. The step complexity of APPEND is logarithmic in the number of nodes reachable from the root pointed to by t.

To perform a query operation, it suffices to perform the query as one would in the underlying augmented balanced binary tree, starting from a root. However, since the tree reachable from this root never changes while it can be accessed, there are no conflicts with update operations. Using an augmented, balanced binary tree to represent each version of $I(v)$ enables each query to be performed in time logarithmic in the length of the version to which it is applied.

Theorem 1. *A wait-free, linearizable, unbounded* Fetch&Inc *object shared by p processes on which at most n* FI *operations are performed can be implemented so that each* FI *takes $O(\log p \log n)$ steps and each* READ *takes $O(1)$ steps.*

Proof (sketch). An instance of FI is linearized when $root(\tau).T$ is first updated during an APPEND (not necessarily performed by the same process) to point to the root a tree that contains information about this instance. At each node v of τ, the length of the sequence represented by any tree in T_v is at most n, so each operation on T_v can be performed in $O(\log n)$ steps. Since the tree τ has height $\Theta(\log p)$ and a process performs only a constant number of operations at

each node on the path from its leaf to the root during an instance of FI, each FI operation takes $O(\log p \log n)$ steps.

To READ the value of the FETCH&INC object, a process reads $root(\tau).T$ to get a pointer to the current root of the tree representing $I(root(\tau))$. The READ is linearized at this step. If t is NIL, then the FETCH&INC object has its initial value, 0. Otherwise, its value is the sum of the persistent values $t.NL$ and $t.NR$, which is the length of $C(root(\tau))$ at the linearization point. This takes a constant number of steps. ■

Initially, this implementation uses $\Theta(p)$ space. Each FI operation adds $O(\log n)$ nodes to the data structure T_v, for each node v on the path from some leaf of τ to its root. Since τ has height $O(\log p)$, the total space used by this implementation to perform n operations is $O(p + n \log n \log p)$.

For a one-shot FETCH&INC object, $n \leq p$, so $O(\log^2 p)$ steps are used to perform each instance of FI and $O(p \log^2 p)$ registers and LL/SC objects are used.

3 The Second Implementation

We now present a more efficient implementation, which is obtained by compressing the tree in the data structure T_v that is pointed to by each node v of τ. Specifically, if there are $\ell = \ell(v)$ leaves in the subtree of τ rooted at v, we show how to ensure that the number of nodes reachable from each root of T_v is $O(\ell^2)$. This results in an implementation whose worst-case step complexity is $O(\log^2 p)$.

If $C(v) = [c_0, \ldots, c_{k-1}]$, we define $Q(v) = \{(j, c_j) \mid j = 0, \ldots, k-1\}$ to be the set of all position-id pairs. We say that a position j is *old at* v if there exists $i \in P(v)$ and $j' > j$ such that $(j, i), (j', i) \in Q(v)$, i.e., some id i occurs at position j in $C(v)$, but this is not the last occurrence of i. A position is *current at* v, if it is not old at v.

We take advantage of the fact that once a position becomes old at v, the identifier at that position is no longer accessed by any process. Thus, the identifiers at old positions can be permuted without affecting the outcome of pending or future FI operations. For example, if $I(v)$ contains three consecutive blocks (L, x), (R, y), (L, z), which represent $x + y + z$ old positions in $C(v)$, then we can replace these blocks with two blocks $(L, x+z), (R, y)$. Because the permuted sequence has fewer blocks, it can be represented by a tree with fewer nodes. An algorithm to compress ℓ consecutive positions of $C(v)$ is presented in Section 3.1. It has $O(\log \ell)$ step complexity.

We add a *deletion structure* $\Delta(v)$ at v to facilitate the identification of sequences of ℓ consecutive old positions to compress. It contains an array of $2\ell + 1$ *status-units*, which are described in Section 3.2. Each status-unit is associated with ℓ consecutive positions of the sequence $C(v)$. A persistent balanced binary search tree, A_v, enables processes to find the status-unit associated with any current position. When a position j becomes old, the process whose id is at position j of $C(v)$ records that fact in the status-unit associated with position j. The status-unit is also used to determine when all of its associated positions

are old. A fixed *deletion tree*, D_v, with $2\ell + 1$ leaves, described in Section 3.4, is used to keep track of such status units.

After all the positions associated with a status-unit have been compressed, the status-unit is recycled. This means that it is reinitialized and associated with a new sequence of positions. This is described in Section 3.5.

To perform FI, a process proceeds up the tree τ, as in the first implementation, starting from its leaf. Before propagating information up to a node v from its children, process i finds the status-unit associated with the position j in $C(v)$ that contains the id i added to the sequence when i last performed FI. Then it *marks* position j in that status-unit, to indicate that the position is old at v. If there is a status-unit whose associated positions are all old, process i also tries to compress these positions in the tree in T_v rooted at $v.T$ and recycle the status unit. The algorithm for performing FI is described in more detail in Section 3.6.

3.1 Compression

The idea of compression is as follows: Once all positions in $C(v)$ that correspond to a block (s, x), $s \in \{L, R\}$, of $I(v)$ are marked, the entire block can be marked by changing its side s to $s' \in \{L', R'\}$, i.e., by replacing (L, x) with (L', x) or (R, x) with (R', x). A block (s', x) with $s \in \{L', R'\}$ is called a *marked block*. Two adjacent marked blocks (s', y) and (s', z) with the same side s' can be replaced by a single marked block $(s', y + z)$. A sequence of consecutive marked blocks, $(s'_j, x_j), (s'_{j+1}, x_{j+1}), \ldots, (s'_k, x_k)$, containing at least one with side L' and one with side R' can be replaced by two marked blocks (L', y) and (R', z), where $y = \sum\{x_i \mid j \leq i \leq k \text{ and } s'_i = L'\}$ and $z = \sum\{x_i \mid j \leq i \leq k \text{ and } s'_i = R'\}$. This is equivalent to permuting the elements in the corresponding locations of $C(v)$.

Suppose t is a pointer into T_v indicating a version I of $I(v)$ and suppose the ℓ consecutive positions $m, \ldots, m + \ell - 1$ in a status-unit are all marked. These positions in I are compressed by updating T_v as follows: We assume $m > 0$; the special case $m = 0$ can be handled analogously. First, FINDBLOCK(t, m) and FINDBLOCK$(t, m + \ell + 1)$ are used to find the blocks (s_j, x_j) and (s_k, x_k) that represent positions $m - 1$ and $m + \ell$, respectively.

If $j = k$, then (s_j, x_j) is partitioned into three blocks, (s_j, x'_j), (s'_j, ℓ), and (s_j, x''_j), where x'_j is the number of positions less than m represented by (s_j, x_j) and x''_j is the number of positions greater than or equal to $m + \ell$ represented by (s_j, x_j). Note that block (s'_j, ℓ) is now marked, and all positions represented by that block are old. Moreover, the number of blocks in $I(v)$ and, hence, the number of nodes in T_v increased by 2.

Now suppose that $j \neq k$. If $j < k - 1$, then all of the positions represented by blocks (s_i, x_i), $j < i < k$, are marked. These blocks are removed from the tree rooted at t.

If m is represented by block (s_j, x_j), then (s_j, x_j) is conceptually partitioned into two blocks (s_j, x'_j) and (s'_j, x''_j), where x'_j is the number of positions less than m represented by (s_j, x_j) and x''_j is the number of positions greater than or equal to m represented by (s_j, x_j). The block (s'_j, x''_j) is removed. This is

accomplished by changing x_j to x'_j. Similarly, if $m+\ell-1$ is represented by block (s_k, x_k), then (s_k, x_k) is conceptually partitioned into two blocks (s'_k, x'_k) and (s_k, x''_k) where x'_k is the number of values less than $m+\ell$ represented by (s_k, x_k) and x''_k is the number of values greater than or equal to $m+\ell$, represented by (s_k, x_k). The block (s'_k, x'_k) is also removed.

If block (s_j, x_j) is marked and does not represent m, then it is removed and, if it is immediately preceded by a marked block, that block is removed, too. Similarly, if block (s_k, x_k) is marked and does not represent $m+\ell-1$, then it is removed, together with the next block, if it exists and is also marked. Note that, if there is a block immediately preceding or immediately following the removed blocks, it is unmarked.

Let x be the sum of the sizes of all the removed blocks with $side = L$. This can be computed in $O(\log \ell)$ steps directly from the tree rooted at t in T_v as the blocks are removed, using the augmented information at each node.

Similarly, let y be the sum of the sizes of all removed blocks with $side = R$. Finally, in place of the removed blocks, add the new marked block (L', x), if $x > 0$, and the new marked block (R', y), if $y > 0$. This maintains the invariant that there is at least one unmarked block between any two marked blocks with the same side. Since $I(v)$ contains $O(\ell^2)$ unmarked blocks, $I(v)$ contains $O(\ell^2)$ blocks in total. Hence, the tree in T_v that represents $I(v)$ has $O(\ell^2)$ nodes.

3.2 Status-Units

The deletion structure $\Delta(v)$ contains a collection of $2\ell+1$ *status-units*, $S_v[j]$, for $1 \le j \le 2\ell+1$. A status-unit has three parts: a *name*, a *flag*, and a *progress tree*. A name is a non-negative integer that can increase during an execution. When its name is g, the status-unit is associated with the ℓ consecutive positions $\ell g, \ldots, \ell g+\ell-1$. Initially, status-unit $S_v[j]$ has name $j-1$, for $j = 1, \ldots, 2\ell+1$.

An LL/SC object can be used to store the name of a status unit. However, the name of a status unit grows each time it is recycled. To avoid using large LL/SC objects, we represent the name of a status unit using an LL/SC object *namer*, which stores a process identifier in $P(v)$, and an array, *names*, of ℓ single-writer registers, indexed by $P(v)$. At any time, the name of the status unit is the value of $names[namer]$.

The *flag* is a single-bit LL/SC object. It is initially 0 and it is reset to 0 whenever the status-unit changes its name. After all of its associated positions have been marked, its *flag* is changed from 0 to 1. This indicates that these positions can be compressed. After that has been done, the status-unit can be reused.

The *progress tree* is a fixed full binary tree on ℓ leaves, represented implicitly by an array $progress[1..2\ell-1]$ of $\ell-1$ single-bit LL/SC objects and ℓ single-bit registers. It enables processes to determine when all the positions, $\ell g, \ldots, \ell g + \ell - 1$, represented by a status-unit with name g are old at v. Progress trees were introduced for processes to keep track of their collective progress performing a collection of tasks [4,7].

When a status-unit is reused, its progress tree also needs to be reused. Because it has $2\ell-1$ fields, it would take too much time to reinitialize all of them to 0.

Thus, we need an implementation of a progress tree that can be reused without being reinitialized. This is discussed in Section 3.3.

Each node of the tree A_v stores the name of one status-unit. The index of the status-unit with that name is stored as auxiliary data. Processes make updates to A_v similarly to the way they make updates to T_v. Initially, A_v stores the initial names of all $2\ell + 1$ status-units (i.e., name $j - 1$ and auxiliary data j for status-unit j, $1 \leq j \leq 2\ell + 1$).

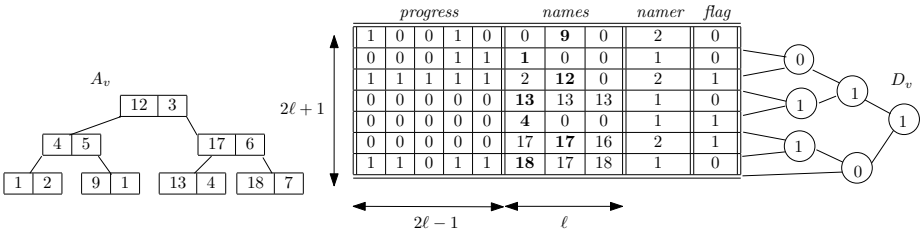

Fig. 2. An example of a deletion structure $\Delta(v)$ for $\ell = 3$

Figure 2 presents an example of a deletion structure for a node v of τ with $\ell = 3$ leaves in its subtree. The j'th row in the table represents the status-unit $S_v[j]$, for $j = 1, \ldots, 7 = 2\ell + 1$. Its name is indicated in bold. The tree on the left is A_v and the tree on the right is D_v.

3.3 Reusable Progress Trees

A reusable progress tree is represented implicitly using an array of length $2\ell - 1$ (as in a binary heap). We will use *leaf*(m) to denote the location of the m'th leaf in this array and use $parent(u)$, $left(u)$, and $right(u)$ to represent the locations of the parent, left child, and right child, respectively, of node u.

Each node of the progress tree stores a single bit. When the $flag$ of a status-unit is 1, all bits of its progress tree are the same. Before any nodes in the progress tree are changed, the $flag$ is reset to 0. Only the process that received position $g\ell + m - 1$ can change the value of *leaf*(m) in the progress tree of the status-unit with name g. Say it changes this bit from $1 - b$ to b. After doing so, the process progresses up the tree, setting the bit at each ancestor of *leaf*(m) to b, if the bits at both of the children of that ancestor are b. Thus, an internal node of the tree is changed only after all of the leaves in its subtree have been changed. When the bit at the root of the progress tree changes to b, all of the bits in the tree are b and the $flag$ can be changed from 0 to 1. After that, the $flag$ is not reset to 0 again until *namer* is changed.

A process that is progressing up the tree, but falls asleep for a long time, should not change bits in the *progress* tree of a status-unit that has since changed its name. To ensure this, each bit corresponding to a non-leaf node of the *progress* tree is an LL/SC object. Before changing its leaf, the process performs LL(*namer*). To change the bit at an internal node u to b, the process

performs LL(u) followed by VL($namer$) and only performs SC(u, b) and continues to $parent(u)$ if the validation indicates that $namer$ has not been updated. If the validation is unsuccessful, the process is done, since the bit at the root has already been changed to b. After a process changes the bit at the root of the progress tree to b, it performs LL($flag$) and VL($namer$) to verify that $flag$ has value 0 and $namer$ has not changed. If successful, it then performs SC($flag, 1$) to change $flag$ to 1.

Pseudocode is presented on lines 1–10 of Figure 3.

```
1   u ← leaf(m)
2   b ← ¬S_v[j].progress[u]
3   LL(S_v[j].namer)
    %Change the mark at leaf(m)
4   S_v[j].progress[u] ← b
5   while u ≠ root do
6         if S_v[j].progress[sibling(u)] = ¬b then return
7         u ← parent(u)
8         LL(S_v[j].progress[u])
9         if ¬ VL(S_v[j].namer) then return
10        SC(S_v[j].progress[u], b)
    end while
11  LL(S_v[j].flag)
12  if ¬ VL(S_v[j].namer) then return
13  if ¬ SC(S_v[j].flag, 1) then return
14  u ← j'th leaf of D_v
15  while u ≠ root do
16        u ← parent(u)
17        LL(D_v[u])
18        if ¬ VL(S_v[j].namer) then return
19        if SC(D_v[u], 1) = false
20        then if LL(D_v[u]) = 0
21                then if ¬ VL(S_v[j].namer) then return
22                     SC(D_v[u], 1)
    end while
23  return
```

Fig. 3. Algorithm to mark position $\ell n' + m - 1$ in status-unit $S_v[j]$ with name n' and add j to D_v, if necessary

3.4 Deletion Tree D_v

The deletion tree D_v is used to represent the marked status-units, i.e. whose flags are set. Hence, the positions associated with these status-units can be compressed. This data structure allows a process to efficiently find such a status-unit. It can also be updated efficiently. It is a fixed full binary tree whose leaves are $S_v[1].flag, \ldots, S_v[2\ell + 1].flag$. Each non-leaf node is a single-bit LL/SC object, which is initially 0.

When a process changes the flag of the status unit $S_v[j]$ from 0 to 1, it adds j to the set by walking up the tree D_v starting from the parent of this leaf, trying

to set every LL/SC object it visits on this path to 1. Because the process may be slow or may fall asleep for a long time, status-unit $S_v[j]$ may be reallocated before the process reaches the root of D_v. To prevent this from causing problems, the process proceeds as in a reusable progress tree: For each node u that the process visits on the path it first executes LL($D_v[u]$), then VL($S_v[j].namer$) and finally performs SC($D_v[u], 1$) if and only if the validation of $S_v[j].namer$ was successful. If the validation was successful, but the SC fails and $D_v[u]$ is still 0 after the unsuccessful SC, the process repeats the LL and VL a second time and, if the validation is now successful, it also performs SC a second time. If the validations are successful, the process proceeds to $parent(u)$. If any validation is unsuccessful, $S_v[j]$ has already been recycled and the process does not continue. Pseudocode appears on lines 11–23 of Figure 3.

While a status-unit $S_v[j]$ is being recycled, its flag gets reset to 0, and j has to be removed from D_v. To do so, a process proceeds up the tree D_v on the path from the i'th leaf to the root, trying to reset the bit at each node to 0 until it finds a node which has a child with value 1, indicating the presence of a leaf with value 1 in its subtree. Specifically, at the non-leaf node u, the process performs LL($D_v[u]$) and, if 0 is returned, it proceeds to $parent(u)$ (or is done, if u is the root). If u was 1, then the process performs LL($D_v[left(u)]$) and LL($D_v[right(u)]$). If at least one of the LL operations returns 1, the process is done. Otherwise, it performs VL(v). If the validation is unsuccessful, the process is done. Otherwise, it performs SC($D_v[u], 0$). If $D_v[u]$ is still 1 after the SC, the process repeats the LL's follows by a VL a second time and, if no child of u has value 1 and the validation is successful, the process also performs SC a second time. If u is 0 after either of these SC's, the process continues to $parent(u)$ (or is done, if u is the root). If not, some other process adding some value j into D_v performed LL($D_v[u]$) and SC($D_v[u], 1$) between the first process's first LL($D_v[u]$) and its last SC($D_v[u], 0$). In this case, $S_v[j].flag$ is a leaf of node u and had value 1 between the LL and SC, and the first process can stop.

More generally, the following invariants will be maintained:

— if a non-leaf node u of D_v is 0, then either there are no leaves with value 1 in the subtree of D_v rooted at u or there is some leaf in its subtree that has value 1 and the process that last changed this leaf to 1 is at or below node u, and
— if a non-leaf node u of D_v is 1, then there is a leaf in the subtree of D_v rooted at u that either has value 1 or is being recycled by some process that is at or below node u.

Note that multiple status-units can be added to D_v concurrently, but only one status unit is removed from D_v at a time.

To find a marked status-unit (i.e., one whose flag is set), a process walks down the tree from the root to a leaf, at each node reading the values of its children, and proceeding to a child whose value is 1, if such a child exists. If the process reaches a leaf j with value 1, this means that $S_v[j].flag$ is set. It may happen that the process gets stuck at a node whose children both have value 0, in which case the process aborts its attempt to find a marked status-unit.

3.5 Recycling

Status units are recycled one at a time. The node v has a field $v.E$ that indicates which status unit is to be recycled next. When a process recycles a status unit $S_v[j]$, it first tries to change its name. Process i begins by writing a proposed new name into the single-writer register $S_v[j].names[i]$. Then it tries to change $S_v[j].namer$ to i by performing LL($S_v[j].namer$) followed by SC($S_v[j].namer, i$). If the SC is successful, this changes the name of $S_v[j]$ to the name it proposed. To prevent a slow process from accidentally changing the name of a status-unit that has already been recycled, process i performs a VL at v between the LL and the SC. If the validation is unsuccessful, then the recycling of status-unit $S_v[j]$ has already been completed and process i does not continue trying to recycle it.

Next, process i tries to change $S_v[j].flag$ from 1 to 0 by executing LL($S_v[j].flag$) and, if that returns 1, performing SC($S_v[j].flag, 0$). Again, process i performs a VL at v between each LL and matching SC to see whether $S_v[j]$ has finished being recycling and, if so, does not continue to try to recycle it. Then process i removes j from the set represented by D_v, as described in Section 3.4.

3.6 Overall Algorithm

Instead of the LL/SC object $v.T$, as in the first implementation, we now have an LL/SC object with three fields, $v.T$, $v.A$, and $v.E$. To avoid having an LL/SC object with multiple fields, we could use indirection and, instead, have the LL/SC object contain a pointer to a record with three registers.

The first two fields contain (pointers to) the trees of T_v and A_v, rooted at $v.T$ and $v.A$, respectively. The last field contains an element e of $\{0, \ldots, 2\ell+1\}$. When $e \neq 0$, $S_v[e]$ is a status unit that is ready to be recycled, i.e. it has been marked, the positions associated with its current name have been compressed, and there is no node in A_v whose key is this name.

The algorithm to perform FI is a modification of the algorithm in Figure 1. Lines 11 and 23 are replaced by $(t, a, e) \leftarrow$LL$(v.T, v.A, v.E)$. Since T_v and A_v are persistent data structures, the trees rooted at t and a do not change during an iteration of the while loop beginning on line 12.

Before the body of this loop is performed, process i finds the largest name n'' stored in the tree rooted at a. If process i has previously performed an instance of FI, it determines the name of the status-unit that is associated with the last position m' of i in $C(v)$. Then it searches in the binary search tree rooted at a for the key with this name. Since position m' is not yet marked, it can be shown that a node having this name will be found. Let j' be the index of the status-unit, which is also stored in this node. Process i marks m' as old in $S_v[j']$ and propagates this change up the progress tree, as described in Section 3.3.

If $e \neq 0$, then process i tries to recycle the status-unit $S_v[e]$ by updating its name to $n'' + 1$, changing $S_v[e].flag$ from 1 to 0, and deleting e from the set represented by D_v, as described in Section 3.5.

Next, process i tries to find a status-unit whose flag is 1, using the deletion tree D_v as described in Section 3.4. If it finds such a status unit e', then process i compresses the positions associated with the status-unit $S_v[e']$ in the tree rooted

at t, as described in Section 3.1, and uses the root of the resulting tree in place of t in line 21. Otherwise, $e' = 0$.

If $e' = e = 0$, let $a' = a$. Otherwise, process i creates a new tree in A_v starting from the tree with root a by adding a node with key $n'' + 1$ and auxiliary data e (if $e \neq 0$) and removing the node with key $S_v[e'].names[S_v[e'].namer]$ (if $e' \neq 0$). Let a' denote the root of this tree.

Finally, line 22 in Figure 1 is replaced with SC$((v.T, v.A, v.E), (t', a', e'))$, where t' is the result computed on line 21. Note that unless this SC is successful, process i makes no modifications to the trees of T_v and A_v, rooted at $v.T$ and $v.A$, respectively. However, the changes made by each process to status-units and D_v occur asynchronously before it attempts this SC at the end of the iteration.

At any point in time, at most 2ℓ of the positions in $\{0, \ldots, |C(v)| - 1\}$ are not marked: the current position for each process and possibly its previous position. Since each status-unit is associated with ℓ positions, there are $O(\ell^2)$ positions represented in the uncompressed portion of the tree in T_v rooted at $v.T$. It follows that this tree has $O(\ell^2)$ nodes. We now state our main theorem.

Theorem 2. *A wait-free, linearizable, unbounded* Fetch&Inc *object shared by p processes can be implemented so that each* Fetch&Inc *takes $O(\log^2 p)$ steps and each READ takes $O(1)$ steps, regardless of the number of* Fetch&Inc *operations performed. Assuming garbage collection is performed, the number of registers and LL/SC objects needed is $O(p^2)$.*

Proof (sketch). The linearization points of FI and READ are the same as in the proof of Theorem 1. Suppose v is a node of τ with $\ell \leq p$ leaves in its subtree. Since $O(\ell)$ and $O(\ell^2)$ nodes are reachable from any root of a tree in A_v and T_v, respectively, searches and updates in these persistent data structures take $O(\log p)$ steps. Likewise, D_v and the progress trees are fixed balanced trees with $2\ell + 1$ and ℓ leaves, respectively, so operations on them also take $O(\log p)$ steps. Since τ has height $O(\log p)$, it follows that FI has $O(\log^2 p)$ step complexity. As in the first implementation, READ can be performed in a constant number of steps.

Excluding A_v, $O(\ell^2)$ registers and LL/SC objects are used to represent $\Delta(v)$. There are $O(\ell^2)$ nodes, each consisting of a constant number of objects, reachable from the roots of A_v and T_v. Summing over all internal nodes v of τ gives a total of $O\left(\sum_{j=0}^{\lceil \log_2 p \rceil - 1} 2^j (2^{\lceil \log_2 p \rceil - j})^2\right) = O(p^2)$ objects.

Although only a bounded number of nodes in A_v and T_v are reachable from $v.A$ and $v.T$, the total number of nodes in these persistent data structures grows with n. However, full persistence is not needed by our implementation: When a node is no longer reachable from v or the local pointer of any process, it can be removed to save space, since it will not be accessed from then on. At any point in time, each process has a constant number of local pointers into persistent structures and $O(p^2)$ nodes are reachable from each of them. Thus, the total number of registers and LL/SC objects used by our implementation is $O(p^3)$, assuming garbage collection is performed.

4 Extensions

Our implementations of FETCH&INC can be extended to FETCH&ADD by having each element of the sequence $C(v)$ contain the input to each instance of FA, together with the identifier of the process that performed the instance. Likewise, each block of $I(v)$ can be augmented with the sum of the inputs to all instances occurring in or before this block and that are from the same side. Details of this algorithm and a proof of its correctness will appear in the full version of the paper. Ways to remove nodes from A_v and T_v that are no longer reachable will also be addressed.

References

1. Afek, Y., Dauber, D., Touitou, D.: Wait-free made fast. In: Proc. of 27th ACM STOC, pp. 538–547 (1995)
2. Afek, Y., Weisberger, E., Weisman, H.: A completeness theorem for a class of synchronization objects. In: Proc. of 12th PODC, pp. 159–170 (1993)
3. Alistarh, D., Aspnes, J., Censor-Hillel, K., Gilbert, S., Zadimoghaddam, M.: Optimal-time adaptive strong renaming, with applications to counting. In: Proc. of 30th PODC, pp. 239–248 (2011)
4. Anderson, R.J., Woll, H.: Algorithms for the certified write-all problem. SIAM J. Comput. 26(5), 1277–1283 (1997)
5. Anderson, T.: The performance of spin lock alternatives for shared-money multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1), 6–16 (1990)
6. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. of the ACM 41(5), 1020–1048 (1994)
7. Buss, J.F., Kanellakis, P.C., Ragde, P., Shvartsman, A.A.: Parallel algorithms with processor failures and delays. J. Algorithms 20(1), 45–86 (1996)
8. Clements, A.T., Kaashoek, M.F., Zeldovich, N.: Scalable address spaces using RCU balanced trees. In: 17th ASPLOS, pp. 199–210 (2012)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2001)
10. Freudenthal, E., Gottlieb, A.: Process coordination with fetch-and-increment. In: Proc. of ASPLOS-IV, pp. 260–268 (1991)
11. Goodman, J., Vernon, M., Woest, P.: Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In: Proc. of ASPLOS-III, pp. 64–75 (1989)
12. Gottlieb, A., Lubachevsky, B., Rudolph, L.: Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. ACM Trans. Program. Lang. Syst. 5(2), 164–189 (1983)
13. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
14. Herlihy, M.: A methodology for implementing highly concurrent objects. ACM Trans. Program. Lang. Syst. 15(5), 745–770 (1993)
15. Herlihy, M., Shavit, N., Waarts, O.: Linearizable counting networks. Distr. Comp. 9(4), 193–203 (1996)
16. Jayanti, P.: A time complexity lower bound for randomized implementations of some shared objects. In: Proc. of 17th PODC, pp. 201–210 (1998)
17. Jayanti, P., Tan, K., Toueg, S.: Time and space lower bounds for nonblocking implementations. SIAM J. Comput. 30(2), 438–456 (2000)