On the Uncontended Complexity of Consensus

Victor Luchangco[1] and Mark Moir[1] and Nir Shavit[2]

[1] Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803
[2] The School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

Abstract. Lock-free algorithms are not required to guarantee a bound on the number of steps an operation takes under contention, so we cannot use the usual worst-case analysis to quantify them. A natural alternative is to consider the worst-case time complexity of operations executed in the more common *uncontended* case.

Many state-of-the-art lock-free algorithms rely on compare-and-swap (CAS) or similar operations with high consensus number to allow effective interprocess coordination. Given the fundamental nature of consensus operations to interprocess coordination, and the fact that instructions such as CAS are usually significantly more costly than simple loads and stores, it seems natural to consider a complexity measure that counts the number of operations with higher consensus number.

In this paper we show that, despite its natural appeal, such a measure is not useful. We do so by showing that one can devise a wait-free implementation of the universal compare-and-swap operation, with a "fast path" that requires only a constant number of loads and stores when the CAS is executed without contention, and uses a hardware CAS operation only if there is contention. Thus, at least in theory, any CAS-based algorithm can be transformed into one that does not invoke *any* CAS operations along its uncontended "fast path", so simply counting the number of such operations invoked in this case is meaningless.

1 Introduction

Compare-and-swap (CAS)—an operation with infinite consensus number in the wait-free/lock-free hierarchy [9]—has been used in implementing numerous non-blocking data structures and operations. Recently, it has been used to implement non-blocking work-stealing deques [1, 8], linked lists [6, 23], hash tables [23, 26], NCAS [7, 10], and software transactional memories [14]. The implementations of these data structures adopt an *optimistic* approach, in which an operation completes quickly in the absence of contention but may incur a significant performance penalty when it encounters contention. The assumption in this approach is that contention is generally low, so most operations will complete without incurring the penalty.

Some useful non-blocking progress conditions, such as lock-freedom and the recently proposed obstruction-freedom condition [12], are not required to provide a worst-case bound on operation time complexity in the contended case. Therefore, standard worst-case analysis is not applicable to such algorithms. An alternative is to count the various types of synchronization operations executed in the presumably common *uncontended* executions, extending an approach taken by Lamport [19], who counted the number of loads and stores on the uncontended "fast path" of a mutual exclusion algorithm.

The cost of executing a hardware CAS operation has been estimated to be anywhere from 3 to 60 times higher than that of a simple load or store, depending on the architecture [2, 17], and this relative cost is expected to grow [16]. Thus, it is natural to consider a complexity measure that counts the number of CAS operations invoked on the uncontended path.

This paper shows that, however natural, such a complexity measure is not useful. We achieve this by presenting a linearizable, wait-free implementation of a shared variable that supports load, store[3] and CAS operations that do not invoke expensive hardware synchronization operations (such as CAS) in the uncontended case. We call our implementation *fast-CAS*.

Fast-CAS variables implemented using our algorithm can be allocated and deallocated dynamically. Unlike many non-blocking algorithms, our algorithm is *population-oblivious* [13]; that is, it does not require knowledge of a bound on the number of threads that access it. When there are V fast-CAS variables and T threads, our algorithm requires $O(T + V)$ space. Therefore, the incremental space overhead for each new variable and each new thread is constant.

The key to our fast-CAS implementation is the understanding that one can build a mechanism to detect contention and default to using hardware CAS only in that case, in a way that allows for a fast-path consensus mechanism that uses only a constant number of multi-writer shared registers when there is no contention. We explain this construction in stages, first exposing the intricacies of building a wait-free one-shot fast-path consensus algorithm (Section 3), and then presenting our reusable fast-path CAS implementation in detail (Section 4).

Our fast-CAS object is widely applicable because it fully complies with the semantics of a shared variable on which atomic load, store, and CAS operations can be performed repeatedly. As such, it can be used as a "black box" in any CAS-based algorithm in the literature. Thus, it implies the following theorem:

> Any CAS-based concurrent algorithm, blocking or non-blocking, can be transformed to one that does not invoke hardware CAS operations if there is no contention.

Though our algorithm is widely applicable, it should be viewed primarily as a theoretical result, as it is unlikely to provide performance benefits in the common shared-memory multiprocessor architectures in which we are most interested (the "fast" part of the name relates more to the motivation than to the end result). Our fast-CAS implementation—which requires up to six loads and four stores in the absence of contention—is likely to be at least as expensive as invoking a hardware CAS operation on such architectures. Moreover, the algorithm depends

[3] In this paper, we show only how to implement load and CAS operations; store is a straightforward extension of CAS.

on sequential consistency [18] for correctness, and must therefore be augmented with expensive memory barrier operations on many modern architectures.[4]

Nevertheless, our result implies that it is *never necessary* to invoke a hardware CAS operation if there is no contention, *regardless* of the non-blocking progress condition considered. This has important consequences both for the search for a separation between novel consistency conditions such as obstruction-freedom [12] and stronger progress properties in modern shared-memory multiprocessors [9], and for guiding our thinking when designing and evaluating non-blocking algorithms. Specifically, it shows that what we and others have considered to be a natural measure—the number of CAS operations invoked by an uncontended operation—is not a useful measure for either purpose.

Unrelated Work

Because CAS can be used to implement wait-free consensus, while loads and stores alone are not sufficient, our result may seem counterintuitive to some readers familiar with various impossibility results and lower bounds in the literature. In particular, using the bivalency proof technique of Fischer, Lynch and Paterson [5], Loui and Abu-Amara [20] showed that no deterministic, wait-free algorithm can implement consensus using only loads and stores. Furthermore, Fich, Herlihy and Shavit [4] showed that even nondeterministic consensus algorithms require $\Omega(\sqrt{T})$ space per consensus object for implementations from historyless objects, where T is the number of participants. In contrast, our algorithm is wait-free and deterministic, uses only constant space per fast-CAS variable, and uses only loads and stores in the uncontended case.

This apparent contradiction can be resolved by understanding that the bad executions at the core of these impossibility proofs rely on interleaving the steps of multiple threads. Therefore, these results do not apply to our algorithm, which can use CAS in these contended executions.

Organization of this paper: Section 2 presents some preliminaries. In Section 3, we illustrate the intuition behind the key ideas in our CAS implementation by presenting a much simpler one-shot wait-free consensus algorithm. Then, in Section 4, we present our fast-CAS implementation in detail. We sketch a linearizability proof in Section 5. Concluding remarks appear in Section 6.

2 Preliminaries

The correctness condition we consider for our implementations is that of linearizability [15]. Informally, a linearizable implementation of an object that supports a set of operations guarantees that every execution of every operation can be considered to take effect at a unique point between its invocation and response,

[4] In simple performance experiments conducted with the help of Ori Shalev, our fast-CAS implementation performed slightly worse than hardware CAS.

such that the sequence of operations taken in the order of these points is consistent with the sequential specification of the implemented operations.

For the purposes of presentation, we consider a shared-memory multiprocessor system that supports linearizable load, store, and compare-and-swap (CAS) instructions for accessing memory. A `CAS(a,e,n)` instruction takes three parameters: an address `a`, an expected value `e`, and a new value `n`. If the value currently stored at address `a` matches the expected value `e`, then CAS stores the new value `n` at address `a` and returns *true*; we say that the CAS *succeeds* in this case. Otherwise, CAS returns *false* and does not modify the memory; we say that the CAS *fails* in this case.

3 One-Shot Consensus

Before presenting our fast-CAS implementation in detail, we first illustrate the key idea behind this algorithm by presenting a simple wait-free consensus algorithm. In the consensus problem [5], each process begins with an *input value*, and each process that terminates must *decide* on a value. It is required that every process that decides on a value decides the same value, and that this value is the input value of at least one process.

```
shared variables
   val_t V;   // value
   bool C;    // contention
   val_t D;   // decision value

initially
   ¬C ∧
   D = ⊥ ∧
   V = ⊥

val_t propose(val) {
1: if splitter()
2:     V = val;
3:     if ¬C return val;
   else
4:     C = true;
5:     if V ≠ ⊥
6:         val = V;
7: CAS(&D,⊥,val);
8: return D;
```

Fig. 1. Simple wait-free consensus algorithm.

Our simple consensus algorithm, shown in Fig. 1, has the property that if a single process p executes the entire algorithm without any other process starting to execute it, then p reaches a decision using only read and write operations. The high-level intuition about how this is achieved is the same as for several read-write-based consensus algorithms in the literature. Specifically, if p runs for long enough without any other process starting to execute, then p writes information sufficient for other processes that run later to determine that p has already decided on its own value, and then determines that no other process has yet begun executing. In this case, p can return its own input value; processes that run later will determine that it has done so, and return the same value.

From results mentioned in Section 1, we know that read-write-based consensus algorithms cannot guarantee termination in bounded time in all executions. Some such algorithms (e.g., the algorithm of Saks, Shavit and Woll [25]) use randomization to provide termination with high probability. The algorithm in Fig. 1 takes a different approach: in case of contention, it employs a CAS operation to reach consensus. Below we briefly explain our algorithm and how it avoids the use of CAS in the absence of contention, and employs CAS to ensure that every process reaches a decision within a bounded number of steps in the face of contention.

Central to our consensus algorithm is the use of a "splitter"[5] function. The splitter function has the following properties:

1. if one process completes executing the splitter function before any other process starts, then that process "wins" the splitter (i.e., the function returns *true*); and
2. at most one process wins the splitter.

In addition to the shared variables used to implement the splitter function, the algorithm employs three other shared variables. `V` is used by the process (if any) that wins the splitter to record its input value. `C` is used by processes that detect contention by losing the splitter to indicate to any process that wins the splitter that contention has been detected. `D` is used by processes that detect contention to determine a consensus decision. `C` is initialized to *false*, and `V` and `D` are both initialized to $\perp$, a special value that is assumed not to be the input value of any process.

Consider what happens if process p runs alone. By Property 1 above, p wins the splitter (line 1), writes its input value to `V` (line 2), and then—because no other process is executing—reads *false* from `C` and returns p's input value. By Property 2, any subsequent process q loses the splitter, sets `C` to *true* (line 4), and then reads `V`. Because p completed before q started, q sees p's input value (which is not $\perp$) in `V`, changes its input value to that value (line 6), and attempts to CAS it into `D` (line 7). Because all such processes behave the same way, whether q's CAS succeeds or not, q returns p's input value, which it reads from `D` (line 8).

Now suppose no process runs alone. If all processes return from line 8, it is easy to see that they all return the same value. If any process p returns from line 3, then it won the splitter, so all other processes lose the splitter and return from line 8. Furthermore, p writes its input value to `V` before any process executes line 4, so all other processes see p's input value in `V` on line 5 and, as explained above, return it from line 8.

In the algorithm presented in Fig. 1, all processes except possibly one invoke a CAS operation. We can modify this algorithm so that, provided one process completes before any other starts, *none* of the processes invoke CAS: Any process

[5] Several variations on Lamport's "fast path" mechanism for mutual exclusion [19] have been used in wait-free renaming algorithms, beginning with the work of Moir and Anderson [24]. These mechanisms have come to be known as *splitters* in the renaming literature [3].

that would have returned from line 3 first writes its input value into D, and before invoking CAS, all processes check whether D $= \perp$, and if not, they instead return the value found. Thus, this algorithm can replace a consensus object (or a CAS variable that changes value at most once) in any non-blocking algorithm without changing the asymptotic time or space complexity of the algorithm. However, again, we do not believe such a substitution is likely to provide a performance benefit in practice in current architectures; the immediate value of our results is in their implications about useful complexity measures of non-blocking algorithms, and in guiding results that attempt to establish a separation between non-blocking progress conditions.

4 Wait-Free CAS Implementation

The one-shot consensus algorithm described in the previous section illustrates the basic idea behind the fast-CAS implementation presented in this section: we use a splitter to detect the simple uncontended case, and then use a hardware CAS operation to recover when contention complicates matters. However, because the CAS implementation is long-lived, supporting multiple changes of value, it is more complicated. The first step towards supporting multiple changes is showing that the splitter can be reset and reused in the absence of contention. However, *after* contention has been encountered, we still have to allow value changes, and also when the contention dies down, we want to be able to again avoid the use of the hardware CAS operation.

These goals are achieved by introducing a level of indirection: we associate with each implemented variable a fixed location which, at any point in time, contains a pointer to a memory block; we call this block the *current block* for that variable. Each block contains a value field V; under "normal" circumstances, the V field of the current block contains the abstract value of the implemented variable. A block is *active* when it becomes current, and remains active and current as long as there is no contention. While the current block is active, operations can complete by acting only on that block without using CAS or any other strong synchronization primitives. When there is contention, an operation may make the current block *inactive*, which prevents the abstract value from changing while that block remains current. Once inactive, a block remains inactive. Therefore, an operation that wishes to change the abstract value when the current block is inactive must replace the block (using CAS) with a new active block.[6]

Data structures

The structure of a block and C-like pseudocode for `WFRead` and `WFCAS` are shown in Fig. 2. For simplicity, we present the code for a single location L; it is straightforward to modify our code to support multiple locations.

[6] As discussed later, this operation invokes CAS even it executes alone. However, once the current block is replaced with an active block, no subsequent operation invokes CAS until contention is again detected.

```
struct blk_s {
    pidtype X;   // splitter
    bool Y;      //   variables
    val_t V;     // value
    bool C;      // contention
    val_t D;     // decision value
} blk_t

initially For some b,
    L = b ∧
    ¬b→Y ∧
    ¬b→C ∧
    b→D = ⊥ ∧
    b→V = initial value


val_t decide(blk_t *b) {
1:  v = b→V;
2:  CAS(&b→D,⊥,v);
3:  return b→D;
}

blk_t *get_new_block(val_t nv) {
4:  nb = malloc(sizeof(blk_t));
5:  nb→Y = false;
6:  nb→C = false;
7:  nb→D = ⊥;
8:  nb→V = nv;
9:  return nb;
}
```

```
val_t WFRead() {
10: blk_t *b = L;
11: val_t v = b→V;
12: if (¬b→C) return v;                          R1
13: return decide(b);                            R2
}

bool WFCAS(val_t ev, val_t nv) {
14: if (ev == nv) return WFRead()==ev;           C1
15: blk_t *b = L;
16: b→X = p;
17: if (b→Y) goto 27;
18: b→Y = true;
19: if (b→X ≠ p) goto 27;
20: v = b→V;
21: if (b→C) goto 28;
22: if (v ≠ ev)
        b→Y = false;
        return false;                            C2
23: b→V = nv;
24: if (b→C)
25:     if (decide(b) == nv) return true;        C3
        goto 28;
26: b→Y = false;
    return true;                                 C4
27: b→C = true;
28: if (decide(b) ≠ ev) return false;            C5
29: nb = get_new_block(nv);
30: return CAS(&L,b,nb);                         C6
}
```

Fig. 2. Wait-free implementation of `WFRead` and `WFCAS`. Statement labels indicate atomicity assumptions for proof, as explained in Section 5.

A block b has five fields: The `C` field indicates whether the block is active ($b \rightarrow$`C` holds when b is inactive, i.e., contention has been detected since b became current), and the `V` field of the current block contains the abstract value while that block is active. The `D` field is used to determine the abstract value while the block is current but inactive. (The `D` field is initialized to $\perp$, a special value that is not `L`'s initial value, nor passed to any `WFCAS` operation.) The `X` and `Y` fields are used to implement the splitter that is used to detect contention (the splitter code is included explicitly in lines 16 to 19). For this algorithm, winning the splitter is interpreted as reaching line 20; recall that a process always wins the splitter if there is no contention. If the splitter does detect contention, then the operation branches to line 27, where it makes the block inactive.

Below we describe the `WFCAS` and `WFRead` operations in more detail. These descriptions are sufficient to understand our algorithm, but do not consider all cases. We sketch a formal correctness proof in Section 5.

The `WFCAS` operation

A `WFCAS` operation whose expected and new values are equal is trivially reduced to a `WFRead` operation (discussed below); `WFCAS` simply invokes `WFRead` in this case (line 14). This is not simply an optimization; it is needed for wait-freedom, as explained later. Henceforth, assume the expected and new values differ.

Suppose a `WFCAS` operation reads b when it executes line 15—that is, b is the current block at this time—and then goes through the splitter of block b (lines 16 to 19), reaching line 20, and reads v from b→`V`. As long as b is the current block, the splitter guarantees that until this operation resets the splitter (by setting b→`Y` to *false* in line 22 or 26), no other operation executes lines 20 to 26 (see Lemma 1 in Section 5).

If b is active throughout this operation's execution (checked on lines 21 and 24), then b→`V` contains the abstract value during that interval, so the operation can operate directly on b→`V`. Specifically, if b→`V` does not contain the operation's expected value, the operation resets the splitter, and returns *false* (line 22). Otherwise, the operation stores its new value in b→`V` (line 23). Because the splitter guarantees that no other process writes b→`V` before the operation completes, this store changes the abstract value from the operation's expected value to its new value (assuming b remains active throughout the operation).

If, after writing b→`V`, the operation discovers that b is no longer active (line 24), then it does not know if its store succeeded in changing the abstract value because the store may have occurred before or after b became inactive; no process can ascertain the order of these events. Instead, processes that find (or make) the current block inactive use a simple agreement protocol (by invoking `decide`(b) in line 25 or 28) to determine the value that is the abstract value from the moment that b becomes inactive until b is replaced as the current block; we call this the *decision value of* b. Processes make this determination by attempting to change b→`D` from the special value $\perp$ to a non-$\perp$ value read from b→`V` (line 1). This is achieved using a CAS operation (line 2), so that all processes determine the same decision value (line 3) for block b.

The decision value of block b is crucial to the correctness of our algorithm; we must ensure that it is chosen such that the abstract value changes when b becomes inactive if and only if a `WFCAS` operation executing at that point, with expected and new values corresponding to the abstract values immediately before and after the change, is considered to have taken effect at that point.

As described above, the tricky case occurs when a `WFCAS` operation by some process p stores its new value to b→`V`—where b is the block it determined to be current when it executed line 15—and then determines at line 24 that b has become inactive.

If p's store occurs before b becomes inactive, then the abstract value changes at the store, and so the `WFCAS` operation must return *true*. This is ensured by our algorithm because the decision value of b is a value read from b→`V` after b becomes inactive, and the splitter ensures that no other value is stored into b→`V` after p's store and before p's operation completes (Lemma 1).

On the other hand, if p's store occurs *after* b becomes inactive then the abstract value does not change at the time of the store because, as stated above, from the time b becomes inactive until b is no longer current, the abstract value is the decision value of b. However, if the decision value of b is p's new value, then p cannot determine the relative order of p's store and b becoming inactive. Therefore, p returns *true* (line 25) in this case too, which is correct because the abstract value changes from p's expected value to p's new value when b becomes inactive. (The properties of the splitter and the test in line 22 together ensure that the abstract value is p's expected value immediately before b becomes inactive in this case.)

Otherwise, the decision value of b is *not* p's new value (so p does not return *true* from line 25). In this case, we argue that the decision value of b is the abstract value immediately before b becomes inactive; that is, the abstract value does not change when b becomes inactive. To see why, first observe that p executes the store only after determining that $b \rightarrow \texttt{V}$ is its expected value (line 22). Recall that the splitter guarantees that no process changes $b \rightarrow \texttt{V}$ between p executing statement 20 and the completion of p's operation (Lemma 1). Thus, because the decision value of b is a value read from $b \rightarrow \texttt{V}$ after b becomes inactive, this decision value can only be p's expected value or p's new value.

Recall that, once the current block becomes inactive, the abstract value cannot change until that block is replaced as the current block, as described next.

If a `WFCAS` operation finds that b is inactive and it has not changed the abstract value—that is, either the test on line 21 succeeds or the one on line 25 fails—then the operation invokes `decide`(b) to determine the decision value of b (line 28). This value is the abstract value from the time b became inactive until b is replaced as the current block, and thus, in particular, it is the abstract value at some time during the execution of this operation. If the decision value is not the operation's expected value, the operation returns *false* (line 28).

Otherwise, the operation prepares a new active block with its new value in the `V` field (line 29), and attempts to replace b with this new block using CAS (line 30). If the CAS succeeds, then the abstract value changes from the decision value of b, which is the operation's expected value, to the value in the `V` field of the new block, which is the operation's new value, so the operation returns *true*. If the CAS fails, then at some point during the execution of this operation, another `WFCAS` operation replaced b with its own block, whose `V` field contains that operation's new value, which is not the decision value of b (because that operation's expected value is the decision value of b, and is not, by the test on line 14, the operation's new value). Therefore, immediately after b is replaced, the abstract value differs from the decision value of b, which is the expected value of the `WFCAS` operation that failed to replace b, so that operation can return *false*.

The `WFRead` operation

It is now easy to see how a `WFRead` operation works. It first reads `L` to find the current block (line 10), and then it reads the `V` field of that block (line 11). If that block is active afterwards, then the value read was the abstract value when line 11 executed, so the operation returns that value (line 12). Otherwise, it returns the decision value of that block (line 13). If the block was active when the operation read `L`, then its decision value was the abstract value when the block became inactive. Otherwise, its decision value was the abstract value at the time `L` was read in line 10.

Space overhead

At any point in time, our algorithm requires one fixed location and one block—the current block—for each fast-CAS variable. In addition, deallocation of blocks that were previously current may be prevented because slow threads may still access them in the future, and some blocks may have been allocated but not (yet) made current. (The determination of when a block can be safely freed can be made using a non-blocking memory management scheme such as the one described in [11].) Each one of these additional blocks is associated with at least one thread, and each thread is associated with at most one block. Therefore, the total space requirement of our algorithm when used to implement V variables for T threads is $O(T + V)$; that is, there is a constant additional space cost for each new fast-CAS variable and each new thread.

A note on contention

As we have stated, if there is no contention, the `WFRead` and `WFCAS` operations do not invoke CAS. However, if there is contention, some operations may invoke CAS—even operations that do not execute concurrently with that contention. For example, a pair of concurrent `WFCAS` operations can leave the algorithm in a state in which the current block is no longer active; in this case, a later `WFCAS` operation that executes alone will have to execute a CAS in order to install a new block. Unless contention arises again later, all subsequent operations will again complete without invoking CAS.

5 Linearizability Proof Sketch

In this section, we sketch a proof that our algorithm is a linearizable implementation of `WFRead` and `WFCAS`. In the full paper, we present a complete and more detailed proof [21]. Specifically, given any execution history, we assign each operation a *linearization point* between its invocation and response such that the value returned by each operation is consistent with a sequential execution in which the operations are executed in the order of their linearization points.

To simplify the proof, we consider only complete execution histories (i.e., histories in which every operation invocation has a corresponding response). Because the algorithm is wait-free, the linearizability of all complete histories implies the linearizability of all histories. We assume that every newly allocated block has never been allocated before, which is equivalent to assuming an environment that provides garbage collection. Also, because blocks are not accessed before initialization, we can assume that they already contain the values they will be initialized with before they are allocated and initialized. Thus, for this proof, initialization is a no-op. Finally, we assume that any sequence of instructions starting from one statement label in Fig. 2 and ending immediately before the next statement label is executed atomically, except when that statement calls `decide` or `WFRead`, in which case the actions of the invoked procedure occur before (and not atomically with) the action of the statement that invokes that procedure.

Throughout the proof, references to "any block" or "all blocks" are quantified only over those blocks that are at some time installed as the current block (i.e., the block pointed to by `L`); those that are never installed are accessed only during initialization and so are of no concern.

All blocks evolve in a similar way, as shown in Fig. 3. Specifically, for any block b, initially and until b is installed, $\neg b{\rightarrow}\mathtt{C}$ and $b{\rightarrow}\mathtt{D} = \bot$ hold. Because $b{\rightarrow}\mathtt{C}$ and $b{\rightarrow}\mathtt{D}$ are changed only by lines 27 and 2 respectively, $b{\rightarrow}\mathtt{C}$ and $b{\rightarrow}\mathtt{D} = v$ for any $v \neq \bot$ are stable properties.[7] Furthermore, we have the following properties:

1. $b{\rightarrow}\mathtt{D}$ is set to a non-$\bot$ value by the first execution, if any, of line 2 within an invocation of $\mathtt{decide}(b)$;
2. whenever $\mathtt{decide}(b)$ is invoked, $b{\rightarrow}\mathtt{C}$ already holds; and
3. some invocation of $\mathtt{decide}(b)$ (i.e., the one on line 28) completes before b is replaced as the current block (in line 30).

Recall that b is *active* if $\mathtt{L} = b$ and $\neg b{\rightarrow}\mathtt{C}$. We say that b is *deciding* if it is inactive and $b{\rightarrow}\mathtt{D} = \bot$, and that b is *decided* if $b{\rightarrow}\mathtt{D} \neq \bot$. Following the discussion above, a block is active when it is installed, and must become deciding and then decided before it is replaced. For any block b, at most one non-$\bot$ value is written into $b{\rightarrow}\mathtt{D}$; this value is returned by every invocation of $\mathtt{decide}(b)$.

Let $\delta_{\mathtt{C}}(b)$ be the event, if any, that makes b inactive, $\delta_{\mathtt{D}}(b)$ be the event, if any, that makes b decided, and $dv(b)$ be the non-$\bot$ value, if any, that is written into $b{\rightarrow}\mathtt{D}$. For any block b other than the final block (i.e., the block that is current at the end of the execution history), $\delta_{\mathtt{C}}(b)$, $\delta_{\mathtt{D}}(b)$ and $dv(b)$ are well-defined. If the final block is made inactive, it is decided before the operation that made it inactive completes, so because we are considering only complete histories, either $\delta_{\mathtt{C}}$, $\delta_{\mathtt{D}}$ and $dv(b)$ are all defined for the final block, or none of them are.

If b is the current block (i.e., $\mathtt{L} = b$), the abstract value of `L` is

$$\mathtt{AV} \equiv \begin{cases} b{\rightarrow}\mathtt{V} & \text{if } \neg b{\rightarrow}\mathtt{C} \text{ holds (i.e., } b \text{ is active)} \\ dv(b) & \text{otherwise.} \end{cases}$$

Note that $dv(b)$ is defined whenever $b{\rightarrow}\mathtt{C}$ holds. While b is active, $\mathtt{AV} = b{\rightarrow}\mathtt{V}$; while b is current but inactive (either deciding or decided), $\mathtt{AV} = dv(b)$.

Linearization points

Below we specify the linearization point for each operation. We categorize operations by the statement from which they return, using the labels shown in Fig. 2.

[7] That is, if either property holds in any state, it holds in all subsequent states.

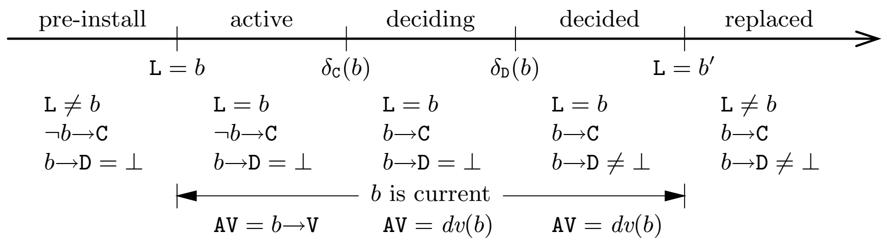
pre-install
active
deciding
decided
replaced
$\mathtt{L} = b$
$\delta_{\mathtt{C}}(b)$
$\delta_{\mathtt{D}}(b)$
$\mathtt{L} = b'$
$\mathtt{L} \neq b$
$\neg b{\rightarrow}\mathtt{C}$
$b{\rightarrow}\mathtt{D} = \bot$
$\mathtt{L} = b$
$\neg b{\rightarrow}\mathtt{C}$
$b{\rightarrow}\mathtt{D} = \bot$
$\mathtt{L} = b$
$b{\rightarrow}\mathtt{C}$
$b{\rightarrow}\mathtt{D} = \bot$
$\mathtt{L} = b$
$b{\rightarrow}\mathtt{C}$
$b{\rightarrow}\mathtt{D} \neq \bot$
$\mathtt{L} \neq b$
$b{\rightarrow}\mathtt{C}$
$b{\rightarrow}\mathtt{D} \neq \bot$
b is current
$\mathtt{AV} = b{\rightarrow}\mathtt{V}$
$\mathtt{AV} = dv(b)$
$\mathtt{AV} = dv(b)$

Fig. 3. The evolution of block b. $\delta_{\mathtt{C}}(b)$ and $\delta_{\mathtt{D}}(b)$ are the events that change $b{\rightarrow}\mathtt{C}$ and $b{\rightarrow}\mathtt{D}$; $dv(b)$ is the decision value of block b; and $\mathtt{AV}$ is the abstract value (see text).

In some cases, we have further subcases. We use $o.k$ to denote the execution by operation o of line k, and $o.\mathtt{x}$ to denote the value of the o's local variable `x` after it is set (an operation sets each local variable at most once).

R1 (returns $\boldsymbol{o}.\mathtt{v}$) Linearize to $o.11$.
R2 (returns $\boldsymbol{dv}(\boldsymbol{o}.\mathtt{b})$), $\boldsymbol{o}.\mathtt{b}$ decided at $\boldsymbol{o}.\mathbf{10}$ Linearize to $o.10$.
R2 (returns $\boldsymbol{dv}(\boldsymbol{o}.\mathtt{b})$), $\boldsymbol{o}.\mathtt{b}$ not decided at $\boldsymbol{o}.\mathbf{10}$ Linearize to $\delta_{\mathtt{D}}(o.\mathtt{b})$.
C1 Linearize to the linearization point of the `WFRead` invoked on line 14.
C2 (returns *false*) Linearize to $o.20$.
C3 (returns *true*), $\boldsymbol{o}.\mathtt{b}$ is active at $\boldsymbol{o}.\mathbf{23}$ Linearize to $o.23$.
C3 (returns *true*), $\boldsymbol{o}.\mathtt{b}$ is not active at $\boldsymbol{o}.\mathbf{23}$ Linearize to $\delta_{\mathtt{C}}(o.\mathtt{b})$.
C4 (returns *true*) Linearize to $o.23$
C5 (returns *false*), $\boldsymbol{o}.\mathtt{b}$ is decided at $\boldsymbol{o}.\mathbf{15}$ Linearize to $o.15$.
C5 (returns *false*), $\boldsymbol{o}.\mathtt{b}$ is not decided at $\boldsymbol{o}.\mathbf{15}$ Linearize to $\delta_{\mathtt{D}}(o.\mathtt{b})$.
C6, returns *true* Linearize to $o.30$.
C6, returns *false* Sometime between $o.15$ and $o.30$, $o.\mathtt{b}$ is replaced as the current block. If $dv(o.\mathtt{b}) \neq o.\mathtt{ev}$ then linearize the operation immediately before $o.\mathtt{b}$ is replaced; otherwise, linearize it immediately after $o.\mathtt{b}$ is replaced.

We first consider the operations that do not change the abstract value. With one exception, it is easy to see, using Fig. 3, that the abstract value is consistent with the operation executing atomically at its linearization point. For example, if `WFRead` returns from R1, then $o.\mathtt{b}$ is active at $o.12$, so it is also active at the operation's linearization point $o.11$; at this point $\mathtt{AV} = o.\mathtt{b}{\rightarrow}\mathtt{V} = o.\mathtt{v}$, as required.

The exception is when a `WFCAS` operation returns *false* from C6. In this case, some other `WFCAS` operation o' replaced $o.\mathtt{b}$ with $o'.\mathtt{nb}$, and because of the tests on lines 14 and 28, $o'.\mathtt{nv} \neq dv(o.\mathtt{b})$. $\mathtt{AV} = dv(o.\mathtt{b})$ holds immediately before $o.\mathtt{b}$ is replaced, which is o's linearization point if $dv(o.\mathtt{b}) \neq o.\mathtt{ev}$. Otherwise, o's linearization point is immediately after $o.\mathtt{b}$ is replaced, when $\mathtt{AV} = o'.\mathtt{nv} \neq o.\mathtt{ev}$.

If a `WFCAS` operation o returns *true* from C6, the CAS on line 30 succeeds. Thus, immediately before $o.30$, $o.\mathtt{b}$ is current and decided, so $\mathtt{AV} = dv(o.\mathtt{b}) = o.\mathtt{ev}$ (see line 28), and immediately after $o.30$, $\mathtt{AV} = o.\mathtt{nb}{\rightarrow}\mathtt{V} = o.\mathtt{nv}$.

For the final cases (when a `WFCAS` operation returns from C3 or C4), we use the following lemma, which says that at most one process has won (and not subsequently reset) the splitter of a block.

Lemma 1. *For any block b, in any state of the history, there exists at most one process p such that $p.\mathtt{b} = b$ and p has reached line 20 but has not subsequently completed its `WFCAS` operation.*

A `WFCAS` operation that returns from C3 or C4 has won the splitter for $o.\mathtt{b}$. Because $o.\mathtt{b}{\rightarrow}\mathtt{V}$ is changed only by line 23, Lemma 1 implies that after $o.20$ until immediately before $o.23$, $o.\mathtt{b}{\rightarrow}\mathtt{V} = o.\mathtt{v}$, which is $o.\mathtt{ev}$ (see line 22). If $o.\mathtt{b}$ is active at $o.23$, then $\mathtt{AV} = o.\mathtt{b}{\rightarrow}\mathtt{V}$ at $o.23$, which changes $o.\mathtt{b}{\rightarrow}\mathtt{V}$ from $o.\mathtt{ev}$ to $o.\mathtt{nv}$. Thus, if a `WFCAS` operation o returns from C4 (Fig. 3 and the test at line $o.24$ imply that $o.\mathtt{b}$ is active at $o.23$ in this case), or if it returns from C3 and o.b is active at $o.23$, then o is correctly linearized at $o.23$. If the operation returns from C3 and o.b is *not* active at $o.23$, then it is linearized to $\delta_{\mathtt{C}}(o.\mathtt{b})$, which occurs between $o.21$ and $o.23$. Immediately before $\delta_{\mathtt{C}}(o.\mathtt{b})$, $\mathtt{AV} = o.\mathtt{b}{\rightarrow}\mathtt{V} = o.\mathtt{ev}$, and immediately after $\delta_{\mathtt{C}}(o.\mathtt{b})$, $\mathtt{AV} = dv(o.\mathtt{b})$, which is $o.\mathtt{nv}$ (see line 25). Thus, `AV` changes from $o.\mathtt{ev}$ to $o.\mathtt{nv}$ at $\delta_{\mathtt{C}}(o.\mathtt{b})$.

Finally, we argue that `AV` changes only at the linearization points of `WFCAS` operations that return *true*. This is the trickiest part of the proof, and depends heavily on the observation the decision value of a block b must have been in $b{\rightarrow}\mathtt{V}$ while b was deciding.

Lemma 2. *If $dv(b)$ is defined, then $b{\rightarrow}\mathtt{V} = dv(b)$ at some point between $\delta_{\mathtt{C}}(b)$ and $\delta_{\mathtt{D}}(b)$.*

There are only three ways in which `AV` may change:

`L` changes Only a successful CAS on line 30 changes `L`, which is done by a `WFCAS` operation that returns *true*.

$b{\rightarrow}\mathtt{V}$ changes while b is active Only $o.23$ for some `WFCAS` operation o changes $b{\rightarrow}\mathtt{V}$. If b is active when $o.24$ is executed, then o returns from C4. Otherwise, $\delta_{\mathtt{C}}(b)$ occurs after $o.23$, so Lemma 1 implies that $b{\rightarrow}\mathtt{V} = o.nv$ from $\delta_{\mathtt{C}}(b)$ until o completes. Because o invokes $\mathtt{decide}(b)$ in this case, $\delta_{\mathtt{D}}(b)$ occurs before o completes, so by Lemma 2, $dv(b) = o.nv$. Thus, o returns *true* from C3, and is linearized to $o.23$ in this case.

$b{\rightarrow}\mathtt{C}$ changes while b is current and $b{\rightarrow}\mathtt{V} \neq dv(b)$ Only line 27 changes $b{\rightarrow}\mathtt{C}$; this event is $\delta_{\mathtt{C}}(b)$. By Lemma 2, $dv(b)$ is stored by some process p into $b{\rightarrow}\mathtt{V}$ after $\delta_{\mathtt{C}}(b)$. Because b is inactive when p executes line 23, its subsequent test on line 24 succeeds. Therefore, p's operation returns *true* from C3 in this case. The linearization point of p's `WFCAS` operation is defined to be $\delta_{\mathtt{C}}(b)$ in this case, as required.

6 Concluding Remarks

We have shown how to implement a linearizable shared variable supporting constant-time read and CAS operations that do not invoke synchronization primitives in the absence of contention. It is straightforward to extend our algorithm to also support a store operation with the same property.

Because of the number of loads and stores performed by our algorithm—even in the absence of contention—it is not clear that our algorithm would ever provide a performance improvement in practice. Nonetheless, our results have important implications for the way we measure non-blocking algorithms and for the study of the differences between various non-blocking progress conditions. Specifically, they show that any non-blocking algorithm (including wait-free ones) that uses CAS can be transformed to one that does not invoke CAS in the absence of contention. Thus, the measure of the number of CAS's invoked in the absence of contention should not be used when comparing algorithms and is also not useful for establishing a separation between obstruction-freedom and stronger progress conditions.

It is interesting to note that the abstract value of the implemented fast-CAS location in a particular state cannot always be determined simply by examining that state (because the abstract value in some states is determined by the decision value of the current block, which is only determined in the future by invoking `decide`). This means that a formal automata-based proof would require a backward simulation, rather than a more straightforward forward simulation [22].

Given the negative results with respect to establishing a separation between obstruction-freedom and stronger non-blocking progress conditions, relevant future work includes seeking alternative ways to achieve such a separation.

Acknowledgments We are grateful to Ori Shalev for help with preliminary performance experiments, and to Simon Doherty for useful discussions.

References

1. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.
2. B. N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings 13th IEEE International Conference on Distributed Computing Systems*, pages 264–273, 1993.
3. H. Buhrman, J. Garay, J. Hoepman, and M. Moir. Long-lived renaming made fast. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 194–203, 1995.
4. F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, 1998.
5. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, pages 374–382, 1985.
6. T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, 2001.
7. T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of 16th International Symposium on DIStributed Computing*, 2002.
8. D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 280–289, 2002.

9. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
10. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free software NCAS and transactional memory. Unpublished manuscript, 2002.
11. M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on DIStributed Computing*, 2002. A improved version of this paper is in preparation for journal submission; please contact authors.
12. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.
13. M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. In *Proceedings of Computing: The Australasian Theory Symposium (CATS)*, 2003.
14. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory of dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, 2003.
15. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
16. K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 130–141, 2002.
17. A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 130–140, Los Angeles, CA, 1994.
18. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
19. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.
20. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, 1987.
21. V. Luchangco, M. Moir, and N. Shavit. On the uncontended complexity of consensus. In preparation, 2002.
22. N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.
23. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.
24. M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25:1–39, 1995. A preliminary version appeared in *Proceedings of the 8th International Workshop on Distributed Algorithms*, 1994, pp. 141-155.
25. M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus — making resilient algorithms fast in practice. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete algorithms*, pages 351–362, 1991.
26. O. Shalev and N. Shavit. Split-ordered lists — lock-free resizable hash tables. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, 2003.