University of Groningen

High-contention mutual exclusion by elevator algorithms

Buhr, Peter A. ; Dice, Dave; Hesselink, Wim H.

Published in:
Concurrency and Computation

DOI:
[10.1002/cpe.4475](https://doi.org/10.1002/cpe.4475)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2018

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

Buhr, P. A., Dice, D., & Hesselink, W. H. (2018). High-contention mutual exclusion by elevator algorithms. *Concurrency and Computation*, 30(18), [4475]. <https://doi.org/10.1002/cpe.4475>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

RESEARCH ARTICLE

High-contention mutual exclusion by elevator algorithms

Peter A. Buhr¹ | David Dice² | Wim H. Hesselink³

¹Cheriton School of Computer Science,
University of Waterloo, Waterloo, ON, Canada

²Oracle Labs, Burlington, MA, USA

³Department of Computing Science, University
of Groningen, 9700 AK Groningen,
The Netherlands

Correspondence

Peter A. Buhr, Cheriton School of Computer
Science, University of Waterloo, 200 University
Avenue West, Waterloo, ON N2L 3G1, Canada.
Email: pabuhr@uwaterloo.ca

Funding information

Natural Sciences and Engineering Research
Council of Canada

Summary

This paper presents new starvation-free hardware-assisted and software-only algorithms for the N -thread mutual-exclusion problem. The hardware-assisted versions use a single atomic-CAS instruction and no fences. The software-only algorithms simulate the CAS instruction using a variation of Burns-Lamport (1 fence) or Lamport's fast algorithm (3 fences). The algorithms are based on Attiya et al, where every thread in the critical section chooses its successor (if one is available). While Attiya et al use a binary tree for this purpose, it can also be done with a linear search. Surprisingly, all software-only algorithms perform equally well under maximal contention on three different computer architectures; the hardware-assisted versions perform better under minimal contention. The new algorithms are between -5% to 50% slower for maximal contention than the starvation-free first-come first-served hardware-assisted MCS algorithm, which uses two atomic instructions (fetch-store and CAS); they are between 10% to 50% slower than MCS for minimal contention.

KEYWORDS

concurrency, mutual exclusion, performance experiment, software solutions

1 | INTRODUCTION

The problem of mutual exclusion was proposed in 1965 by Dijkstra^{1,2} and is formulated as follows. Start with a system of N concurrent threads or processes communicating by shared memory. From time to time, these threads need exclusive access to a shared resource, called a critical section (CS), and then release the CS, continuing in the non-critical section (NCS). When a thread is in a CS, other threads needing that resource must wait, and it is assumed the CS eventually terminates. Mutual exclusion (MX) is the design of an entry and exit protocol that ensures there is never more than one thread in a CS. Furthermore, when threads are waiting in the entry protocol, eventually some waiting thread must enter the CS, called collective progress (the absence of indefinite postponement or livelock). A stronger requirement is that, when a thread needs to enter the CS, it eventually enters, called individual progress (the absence of starvation or lockout). Over the years, many software MX-algorithms have been proposed. Recently, we have performed an investigation of twenty of these algorithms³; the algorithms were implemented and their performance compared under both zero and high contention. In contrast to software solutions are hardware-assisted solutions, which have additional atomicity, eg, test-and-set,⁴ and pure hardware solutions⁵ requiring little or no software.

On current hardware, the implementation of concurrent software and hardware-assisted MX algorithms often requires the insertion of memory fences (also called memory barriers) to ensure correct execution. Fences preclude optimization by the hardware, and therefore, slow down execution. Therefore, it is important to minimize the number of fence operations as much as possible. Recently, Attiya et al⁶ have proposed a MX algorithm, in which the number of fence operations per CS is bounded by a small constant, and the number of remote memory references (RMRs) is logarithmic in N . We call this the RMRs algorithm. The downside of the algorithm is the use of two CAS (compare and set) instructions (see lines 23 and 26 in algorithm 1 in the work of Attiya et al⁶), where CAS is an atomic instruction equivalent to the Boolean function, ie,

$$\text{CAS}(v, c, n) : (\text{if } v = c \text{ then } v \leftarrow n; \text{return true else return false endif}). \quad (1)$$

Atomic instructions that both read and write, like CAS, are also implicit fences. Attiya et al indicated the CAS instructions can be eliminated by standard techniques described in the literature.^{6(p.228)} Inspired by RMRs, we generalize the approach and present several similar algorithms with only one and no CAS instructions. These new algorithms are simpler than RMRs, easier to prove correct, and for maximal contention, 50% faster on SPARC and 250% to 300% (2.5 to 3 times) faster on AMD/Intel, and for minimal contention, 40% faster on SPARC and 25% to 8% faster on AMD/Intel.

1.1 | Overview

This paper presents a family of MX algorithms. Section 3 starts with an abstract algorithm and various options for its implementation. These design options lead to twelve algorithms in Section 4. The main idea is that the thread that holds the CS selects its successor, if available. This selection can be done by a linear or tree search. The linear-search algorithm using one CAS operation is verified for safety and progress in Sections 5 and 6. The tree-search algorithm, using one CAS operation and flags to minimize remote memory references, is verified in Sections 7 and 8. Section 9 extends the verifications of Sections 7 and 8 to the version in which the CAS operation is replaced by a variation of Lamport's fast algorithm.⁷ Section 10 deals with implementation correctness: how to use the **volatile** qualifier and fences to prevent incorrect optimizations by the compiler and hardware. Section 11 explains the implementation of the algorithms in a shared-memory system, how they are tested along with competing algorithms, and evaluates the results. Conclusions are drawn in Section 12.

2 | MOTIVATION AND POTENTIAL USAGE*

Why care about software solutions? A common misconception is that the class of processors used in laptops, PCs, and servers are the majority of processors. In fact, these processors compose only a small fraction of total processors manufactured each year. Simple microcontrollers (eg, Arm Cortex M0-3⁸ and Microchip PIC24 8/16-bit⁹) are used in low-cost devices covering basic cell phones, cameras, printers, music players, toys, and even singing greeting cards. Billions of these low-cost devices are created annually and will continue to exist for years to come. Most of these simple devices provide complex functionality, and it is useful to structure their (real-time) software using preemptive threads.

However, the low-cost processors within these devices may not have atomic instructions, possibly not even atomic reads and writes, even with multiple cores, because cache coherency is complex, expensive, and creates energy/heat issues. For embedded systems using this hardware, it is often possible to establish a maximum bound on the number of threads that interact and compete for shared resources, and hence, to use a software solution for synchronization and mutual exclusion rather than restructuring the software sequentially using an event-driven approach. For example, in the Internet of Things (IoT), a \$50 home thermostat may have Wi-Fi/Internet access, manage multiple devices, such as a furnace, heat pump, air conditioner, and hot-water tank, while providing speech generation and recognition, all using a small embedded real-time multithreaded operating-system on low-cost hardware without atomic instructions. In fact, it is highly likely many people are carrying around a software solution for mutual exclusion in one of their portable devices.

However, software solutions require a minimum of $\mathcal{O}(N \log N)$ total storage, where N is the number of threads, and $\log N$ is the number of bits needed to identify a thread. While this space requirement is impractical for general concurrent systems with a large amount of dynamically created, fine-grain locks and hundreds/thousands of threads, it is not an issue for small embedded systems, where the number of locks and threads is small. For example, assume 10 locks, 30 threads, and 8 bytes per thread per lock for a particular software algorithm = 2400 bytes, which is still small, as embedded processors often have 8-16 M bytes of memory on chip.

Another potential usage is by hardware engineers, who only have N of everything when building hardware. They may be able to adapt software algorithms into hardware circuits to provide complex synchronization or mutual exclusion among the many forms of parallelism that exist on current and future hardware.

3 | ELEVATOR ALGORITHMS

Figure 1 shows an abstract nondeterministic MX algorithm for N threads, which generalizes the RMRs algorithm. We call this the Elevator algorithm because each thread exiting the CS elevates a successor (lines 6-8), if available. The threads are numbered from 0 to $N - 1$. The number N is used to represent Not a Thread. Synchronization is established by the atomic statement at line 3. Here, thread p waits until either $first = p$ or $first = N$. When one of these conditions is true, p assigns $first \leftarrow p$ and goes to line 4 in the same atomic step. It follows that all threads q at lines 4-8 satisfy $q = first$. This implies MX because, if q and r are different threads in the CS, they are both at line 5, and $first$ cannot equal both. The technique of transferring MX directly to another thread in the exit protocol, versus competing for it, comes from Andrew's baton passing.¹⁰

The abstract algorithm of Figure 1 can be refined and adapted in various ways. Section 3.1 is devoted to the implementation of line 3 by **trylock**. Section 3.2 deals with the remote memory references to $first$ in lines 3, 7, and 8. Finally, Section 3.3 deals with the search for a successor in line 6.

3.1 | Using trylock

A **trylock** construction is used to implement the atomic command for line 3 of Figure 1. In general, **trylock** is a Boolean function that atomically returns **true** and toggles a lock closed, allowing a single thread to enter the CS. If the lock is closed, **trylock** does not wait but returns **false**. A closed lock is

* This section is based on section 2 in the work of Hesselink et al¹¹ as the general motivation is similar.

```

var first : 0 ... N  $\leftarrow$  N;
var apply : array[N] of bool  $\leftarrow$  [N] false;

thread(p : 0 ... N - 1) =
  loop
  1   NCS;
  2   apply[p]  $\leftarrow$  true; // entry protocol
  3    $\langle$  await first = p  $\vee$  first = N then first  $\leftarrow$  p  $\rangle$ ;
  4   apply[p]  $\leftarrow$  false;
  5   CS;
  6   if possible choose thr with apply[thr] // exit protocol
  7   then first  $\leftarrow$  thr;
  8   else first  $\leftarrow$  N; endif
  endloop .

```

FIGURE 1 Abstract Elevator Algorithm

opened by *unlock*. For certain versions of *trylock*, line 3 of Figure 1 can be implemented by

```

if trylock(p) then
  await first = p  $\vee$  first = N;
  first  $\leftarrow$  p;
  unlock(p);
else
  await first = p;
endif

```

If *trylock*(p) returns *false*, the lock is taken or is being taken by another thread, and thread *p* waits for the condition *first* = *p*. Three different applicable versions of *trylock* are presented, one using hardware-assist and two using software-only approaches, and each is verified separately. The hardware-assist uses the CAS instruction (see Formula (1)) and the software-only approaches only require atomic read and write instructions.

In general, an implementation of *trylock* is called strong, if a call of *trylock* only responses *false* when the lock is held during the request-response interval, ie, no false negatives. The two software implementations of *trylock* in Sections 3.1.2 and 3.1.3 are not strong.

3.1.1 | Implementing *trylock* by CAS

The CAS instruction can directly implement *trylock* on a shared Boolean variable, say *fast*, which is initially *false*

```

function trylockCAS(p): bool =
  return CAS(fast, false, true);
end trylockCAS.

procedure unlock(p) =
  fast  $\leftarrow$  false;
end unlock.

```

The lock is closed if and only if *fast* holds. The CAS instruction closes the lock if it is open. Unlocking uses a single atomic assignment. It is only allowed for the thread holding the lock, but this is not verified.

3.1.2 | Burns-Lamport

The first software-only approach is based on the Burns-Lamport algorithm^{12(p.836)} (see fig. 1 in the work of Lamport¹³) (see fig. 3 in the work of Lycklama and Hadzilacos¹⁴) for MX, where a naïve implementation of *trylock* and *unlock* is

```

var b: array[N] of bool  $\leftarrow$  [N] false;

function trylockBLO(p): bool =
  b[p]  $\leftarrow$  true;
  forall thr < p do
    if b[thr] then
      b[p]  $\leftarrow$  false; return false; endif
  endfor;
  forall thr > p do
    await  $\neg$  b[thr] endfor;

```

```

    return true
end trylockBLO.

procedure unlockBLO(p) =
    b[p] ← false;
end unlock.

```

The algorithm prioritizes threads based on the thread identifier, where lower identifiers have higher priority. A thread first searches in the high-priority direction for another thread with its intent set; if one is found already competing for `fast`, the searching thread retracts its intent and returns `false` because `trylock` must return `false` when the lock is occupied. Otherwise, the thread searches in the low-priority direction for a thread with its intent set; for each thread found, the searching thread waits for it to retract its intent after exiting the `cs` or finding a higher-priority thread with its intent set. A thread searching in the low-priority direction does not retract its intent. After completing the second search, the searching thread has seen a moment in time when no other thread is attempting entry.

This version of `trylock`, however, is not applicable to implement `CAS` because it has the following deadlock scenario. Assume three threads: p_0, p_1, p_2 (ordered $p_0 < p_1 < p_2$). Thread p_0 is in the `CS` (Figure 1, line 5). Threads p_1 and p_2 now set their `apply` to `true` and call `trylockBLO`. p_2 sets $b[p_2] \leftarrow true$, goes through both loops of `trylockBLO` and waits at Figure 1, line 3. Then, p_1 goes through the first loop of `trylockBLO` and waits in the second loop for $\neg b[p_2]$. However, $b[p_2]$ remains true because p_2 has not called `unlock`. Then, p_0 exits `CS`, chooses `first` ← p_1 , goes through `NCS`, and calls `trylockBLO` again. It now enters the first loop and waits for $\neg b[p_1]$. As a result, p_0 and p_1 are waiting inside `trylockBLO`, and p_2 is waiting in Figure 1.

The point is that the calls of `trylockBLO` by threads p_0 and p_1 do not terminate. To guarantee that every call of `trylock` terminates independently of the status of the lock, `trylockBLO` is extended (Figure 2A, lines 32–33) with the `CAS` simulation from Equation 1 at the point where local mutual-exclusion is acquired within `trylockBLO`. The private variable `leader` for thread p expresses that thread p is the owner of the lock associated with `fast`. (This technique is also used in the next `CAS` simulation.) The corresponding `unlock` procedure is the same as used in Section 3.1.1 (`fast` ← `false`).

There is a bias for high-priority threads to reach line 32 of Figure 2A, but experiments show that, even for $N = 32$, almost all threads entering function `BL` reach line 32. Therefore, the bias is negligible; furthermore, the bias has no effect on the individual progress of the threads.

3.1.3 | Lamport's fast algorithm

A second software-only approach is based on Lamport's fast algorithm (see fig. 2 in the work of Lamport⁷). Figure 2B gives an implementation of `trylock` based on Lamport's algorithm by replacing its backward jumps by return statements of `false` because `trylock` must return `false` when the lock

<pre> var b : array[N] of bool ← [N] false ; function trylockBL(p : 0 ... N - 1) : bool = 23 b[p] ← true ; // entry protocol Fence() ; 24 forall thr < p do 25 if b[thr] then 35a b[p] ← false ; return false ; endif endfor 26 forall thr > p do 27 await ¬b[thr] endfor ; var leader : bool ← false ; 32 if ¬fast then // cs 33 fast ← leader ← true endif ; 35b b[p] ← false ; // exit protocol return leader ; end trylockBL. </pre>	<pre> var b : array[N] of volatile bool ← [N] false ; var x : volatile 0 ... N - 1 ; var y : volatile 0 ... N ← N ; function trylockLF(p : 0 ... N - 1) : bool = 23 b[p] ← true ; // entry protocol 24 x ← p ; // race Fence() ; 25 if y ≠ N then 35a b[p] ← false ; return false endif ; 26 y ← p ; // race Fence() ; 27 if x ≠ p then 28 b[p] ← false ; Fence() ; // optional 29 forall thr ∈ 0 ... N - 1 do 30 await ¬b[thr] endfor ; 31 if y ≠ p then return false endif ; endif ; var leader : bool ← false ; 32 if ¬fast then // cs 33 fast ← leader ← true endif ; 34 y ← N ; // exit protocol 35b b[p] ← false ; return leader ; end trylockLF. </pre>
(A)	(B)

FIGURE 2 Two software-only implementations of `trylock`. A, Based on Burns-Lamport; B, Based on Lamport's fast algorithm

is occupied. The algorithm uses three shared variables x , y , b . The variables x and y are used for thread identifiers. The value $y = N$ is special and indicates that the fast track is empty, where fast track is an execution sequence of `trylockLF` with a finite number of memory accesses that are independent of N . If the fast track is empty, ie, $y = N$, an arriving thread declares its intent and starts the fast track by writing $x \leftarrow p$ at line 24, passes line 25, and then sets $y \leftarrow p$ at line 26. If thread p is fast enough, it then finds $x = p$ at line 27, and proceeds to line 32.

The function returns *false* for threads that find $y \neq N$ at 25. If thread p reaches line 27 but is not fast enough, other threads may have modified x at line 24. Now there are two alternatives. Either some other thread q reaches line 27 while $x = q$ and succeeds, or all threads q that reach line 27 find $x \neq q$. The latter case occurs if all contending threads delay before line 27, and a new thread changes x at line 24 but cannot join the contending threads because $y \neq N$ at line 25.

In the first case, thread q resets y at line 34, and all other threads fail. In the second case, the contending threads enter the region 28–31 and y holds the identifier of one of these threads. This thread resets $y \leftarrow N$ at line 34, and all contending threads $q \neq y$ fail at line 31.

When thread p finds $x \neq p$ at line 27, it has no means to decide which of the two alternatives applies. It therefore has to wait at line 29, until any thread q with $x = q$ at line 27 has retracted its intent at line 35b.[†] The Booleans $b[q]$ serve to indicate this. They are set in line 23 and reset in lines 35a, 28, and 35b. When thread p has waited long enough to satisfy $y = p$ at line 31, it can proceed; otherwise, it returns *false*.

Finally, just as in the case of Burns-Lamport, the variable *fast* is used to ensure that `trylockLF` always terminates, independently of the status of the lock. Again, the *unlock* function of Section 3.1.1 is used.

The following scenarios show our *trylocks* are not strong. Both scenarios start with thread $p0$ holding the lock, so $\neg b[p0]$ and *fast*.

- In Figure 2A, thread $p1$ calls *trylock* and stops at line 35b before the assignment. Thread $p0$ unlocks. Thread $p2$ calls *trylock*, while the lock is free, and fails because $b[p1]$ holds.
- In Figure 2B, thread $p1$ calls *trylock*, sets y to $p1$ and stops at line 34 before the assignment. Thread $p0$ unlocks. Thread $p2$ calls *trylock*, while the lock is free, and fails because $y \neq N$ holds.

3.2 | Remote memory references

A next step in the implementation of Figure 1 is to observe that the occurrences of *first* in lines 3, 7, and 8 are remote memory references because all threads execute these lines concurrently. There is, however, only one thread (viz, p) interested in the truth of $\text{first} = p$. At any time, there is at most one thread (viz, the thread that holds the lock) interested in the truth of $\text{first} = N$. It may therefore be advantageous to split the variable *first* into $N + 1$ Boolean flags such that $\text{flag}[k]$ means the same as $\text{first} = k$. Then, line 3 is implemented by

```

if trylock(p) then
  await flag[p] ∨ flag[N];
  flag[N] ← false;
  unlock(p);
else
  await flag[p];
endif
flag[p] ← false;

```

where lines 7 and 8 become $\text{flag}[k] \leftarrow \text{true}$ with $k = \text{thr}$ or $k = N$, respectively. Hence, global cache-invalidation is reduced over threads accessing *first*. The experiments show local references are advantageous to performance.

Interestingly, while variables $\text{flag}[p]$ and $\text{flag}[N]$ are never simultaneously *true*, we discovered it is simplest to reset both unconditionally, rather than conditionally resetting the one that became *true*. Hence, a remote write to $\text{flag}[N]$ occurs often as threads enter both parts of the *if* statement, without a performance slowdown on the AMD, Intel, and SPARC processors used in the experiments. We are unable to explain how the cache managed to prevent a slowdown, as for *first*, even though there are still remote reads/writes to $\text{flag}[N]$.

3.3 | The search for a successor

In line 6 of Figure 1, thread p searches for a successor to elevate into the CS. Two implementations of this idea are proposed.

3.3.1 | Linear search

A simple approach is a linear search for a thread q that has indicated intent to enter (by $\text{apply}[q] \leftarrow \text{true}$) and then assign $\text{first} \leftarrow q$.

```

k ← N − 1;
while ¬ apply[(p + k) mod N] do k ← k − 1 endwhile;
apply[p] ← false;
first ← (k ≠ 0 ? (p + k) mod N : N);

```

(2)

[†] See the *goto* target in Figure 9 at line 35.

If the search starts at `apply[0]` instead of relative to thread p , then starvation is possible. For instance, threads 0 and 1 could awaken each other indefinitely, while thread 2 remains waiting. The relative search results in non-FIFO execution of waiting threads.

The thread exiting the CS leaves its intent set during the search as a marker; thus, the search is guaranteed to stop. `apply[p]` must be reset before setting `first`; otherwise, there is a race. Resetting `apply[p]` after setting `first` allows a selected (waiting) thread to make progress to the exit protocol, and see the exiting p still with its intent set, and hence, select it to enter next. However, the prior exiting thread may terminate, and hence, no thread can make progress as the selected thread is gone. It is safe to reset `apply[p]` before setting `first` because p has fully retracted its intent so no waiting thread can select it, unless it attempts entry again.

This search gives the delay bound $N - 1$; when thread p sets `apply[p] ← true` and some other thread q is entering or in the CS, only threads $q, q + 1, \dots, p - 1$ (in cyclic order) can enter the CS before p . The worst case is $q = p + 1$; then $N - 1$ threads go before p .

For maximal contention, the linear search is very short, as there are many waiting threads. For minimal contention, the linear search is always N probes to determine there are no waiting threads. If there is one waiting thread, the search takes $N/2$ probes on average. In our experiments, N is small, below 32.

3.3.2 | Tree search

The algorithm of Attiya et al.⁶ (see Figure 3) uses the ingenious idea to let waiting threads announce themselves in a binary tree with $2N - 1$ nodes. Each thread registers itself on entry along its entire root path. After the CS, the thread holding `MX` promotes waiting threads along its root path from the tree to a queue but does not remove the threads from the tree. Forming a queue of waiting threads is related to the Mellor-Crummey and Scott (MCS)¹⁵ algorithm. The queue actions in the exit protocol are performed under the CS mutual exclusion, and therefore need no locking, but examining and updating the tree is done without locking and requires atomic read/write operations.

The exit loop of the RMRs algorithm of Figure 3 unnecessarily inspects three nodes $n1, n2$, and $n3$ per element $n1$ of the root path. We therefore transform the RMRs algorithm into the tree flag elevator of Figure 6B where the inspection is reduced to a single node, the sibling of $n1$.

We use the following definitions for the search tree (see Figure 4). The nodes of the tree are numbered $1 \dots 2N - 1$. The root is 1. Every node $n > 1$ has parent $n \text{ div } 2$. Every node $n < N$ has the children $\text{left}(n) = 2n$ and $\text{right}(n) = 2n + 1$. The nodes $n \geq N$ are the leaves. Thread $p < N$ owns $\text{leaf}(p) = N + p$. The number $\text{depth}(p)$ is defined to be the length of the path of $\text{leaf}(p)$ to the root. The nodes on this root path are enumerated

```

1  var first: volatile 0... N ← N;
2  var val: array[node] of volatile 0... N ← [node] N;
3  var apply, inPromQ, flag: array[N] of volatile bool ← [N] false;
4  var queue: FIFO queue of volatile 0... N - 1 ← ∅;
5  var exits: volatile N ← 0;
6
7  thread( p: 0... N - 1 ) =
8    loop
9      NCS;
10     flag[p] ← false;
11     apply[p] ← true;           // intent to enter
12     node ← leaf(p)
13     while node ≠ null do      // null is the parent of the root
14       val[node] ← p; node ← parent(node);
15     endwhile
16     if ¬CAS(first, N, p) then
17       var e: N ← exits;
18       await exits - e ≥ 2 or first = p or first = N;
19       if ¬CAS(first, N, p) then
20         await flag[p]; endif
21     endif
22     CS;
23     apply[p] ← false;
24     exits ← exits + 1;
25     node ← root;
26     while node ≠ leaf(p) do
27       n1 ← node; n2 ← left(node); n3 ← right(node);
28       for each n in {n1, n2, n3} do
29         if apply[ val[n] ] and ¬inPromQ[ val[n] ] then
30           queue.add( val[n] ); inPromQ[ val[n] ] ← true; endif
31       endfor;
32       node ← n2 or n3 if it is an ancestor of leaf(p)
33     endwhile
34     if queue = ∅ then
35       first ← N;
36     else
37       thr ← drop(queue);
38       inPromQ[thr] ← false;
39       first ← thr;
40       flag[thr] ← true;
41     endif
42   endloop

```

FIGURE 3 RMRs Algorithm

FIGURE 4 Tree Definitions: $N = 4$

FIGURE 5 Search Tree

$path(p, i)$, from the root upward, so that $path(p, 0) = 1$ and $path(p, depth(p)) = leaf(p)$. For every node n with $1 < n < 2N$, the sibling is the other child of its parent, given by

$$sib(n) = (n \bmod 2 = 0 ? n + 1 : n - 1).$$

In comparison with Figure 3, the tree handling in Figure 6B is further simplified by removing Boolean array `inPromQ` and encoding `apply[q]` in the condition `val[leaf(q)] ≠ N`. In detail, the $2N - 1$ tree-nodes are represented by array `val`, where a node index is its array subscript (`val[0]` is unused).

To outline the workings of the tree, consider the scenario in Figure 5. Threads p_0, p_1, p_3 , and p_2 arrive in this order, and p_0 has entered the CS and is selecting the next thread to enter. In our version, the thread leaving the CS (p_0) traverses the tree along its reverse path adding any waiting threads on an opposite branch to the queue. At level 1, the opposite branch holds p_2 , and at level 2, the opposite branch holds p_1 , while waiting thread p_3 is missed. The leaving thread then wakes the thread at the head of the queue (p_2) to enter the CS. Thread p_2 eventually finds thread p_3 in its exit loop as it inspects the siblings of each node on its reverse root path. In Section 8.2, it is proved that every thread waiting in the tree is eventually found by some thread in its exit loop, and moved to the queue. As the queue is FIFO, it follows that there is no starvation.

Remark 1. Even though the binary tree has no node $2N$, our implementations of tree elevators use an additional element `val[2N]` with the constant value N . Then the test $k < N$ in the second for-loop of Figure 6B can be omitted.

An important role in the proof is played by the concept of *meet*. For threads q and r , $meet(q, r)$ is the level of the node in the two paths from leaf to root where the paths join. It is defined as the greatest integer i with $i \leq depth(q)$, $i \leq depth(r)$, and $path(q, i) = path(r, i)$. It is clear that $meet(q, r) = meet(r, q)$. As the threads are located at the leaves of the tree, it implies

$$q \neq r \equiv meet(q, r) < depth(q). \quad (3)$$

If $q \neq r$ and $i = meet(q, r) + 1$, then $path(q, i)$ and $path(r, i)$ are siblings: $path(q, i) = sib(path(r, i))$.

4 | TWELVE ELEVATOR ALGORITHMS

For implementing the algorithm of Figure 1, we offer three orthogonal choices.

- A: Implement line 3 with one of the *trylock* functions of Section 3.1.
- B: Eliminate the remote memory references to the shared variable `first` by means of Boolean flags private to each thread, as proposed in Section 3.2.
- C: Implement line 6 by a linear search or by a *tree+queue* data-structure, as discussed in Section 3.3.

As the choices are independent, this results in $3 \text{ (CAS/Soft1/Soft2)} \times 2 \text{ (shared/local flags)} \times 2 \text{ (data-structures)} = 12$ different algorithms, which are tested on three different computer architectures.

The algorithms are partitioned into six algorithms using a linear search, as described in Section 3.3.1, called Linear Elevator, and six using a tree search, as described in Section 3.3.2, called Tree Elevator. Within these two searching partitions, the algorithms are partitioned by synchronization


```

var fast : volatile bool  $\leftarrow$  false ;
var first : 0 ... N  $\leftarrow$  N ;
var apply : array[N] of volatile
    bool  $\leftarrow$  [N] false ;

thread(p : 0 ... N - 1) =
  loop
    NCS ;
    apply[p]  $\leftarrow$  true ;
    if trylock(p) then
      Fence() ; // only for BL and LF
      await first = p  $\vee$  first = N ;
      first  $\leftarrow$  p ;
      fast  $\leftarrow$  false ;
    else
      await first = p ;
    endif

    CS ;
    k  $\leftarrow$  N - 1 ;
    while  $\neg$  apply[(p + k) mod N] do
      k  $\leftarrow$  k - 1 endwhile ;

    apply[p]  $\leftarrow$  false ;
    first  $\leftarrow$  (k  $\neq$  0 ? (p + k) mod N : N) ;
  endloop .

```

(A)

```

var fast : volatile bool  $\leftarrow$  false ;
var val : array[2N] of volatile N  $\leftarrow$  [2N] N ;
var queue : FIFO queue of volatile 0 ... N - 1  $\leftarrow$   $\emptyset$  ;
var flag : array[0 ... N] of bool  $\leftarrow$  [N] false, true ;

thread(p : 0 ... N - 1) =
  loop
    NCS ;
    for j  $\leftarrow$  depth(p) downto 1 do
      val[path(p, j)]  $\leftarrow$  p endfor ;
    if trylock(p) then
      Fence() ; // only for BL and LF
      await flag[p]  $\vee$  flag[N] ;
      fast  $\leftarrow$  flag[N]  $\leftarrow$  false ;
    else
      await flag[p] ;
    endif
    val[N + p]  $\leftarrow$  N ;
    flag[p]  $\leftarrow$  false ;
    CS ;
    for j  $\leftarrow$  1 to depth(p) do
      k  $\leftarrow$  val[sib(path(p, j))] ;
      if k < N  $\wedge$  val[N + k] < N then
        val[N + k]  $\leftarrow$  N ; add(queue, k) ; endif ;
      endfor ;
      if queue  $\neq$   $\emptyset$  then flag[drop(queue)]  $\leftarrow$  true ;
      else flag[N]  $\leftarrow$  true endif ;
    endloop .

```

(B)

FIGURE 6 Two main Elevator Algorithms. A, Linear Elevator; B, Tree Flag Elevator

from choice A, where the suffixes CAS, BL, and LF refer to choices in Sections 3.1.1, 3.1.2, and 3.1.3, respectively. Finally, within the three synchronization partitions, the algorithms are partitioned from choice B with the suffix Flag if the shared variable `first` is eliminated by means of a Boolean array `flag`, as described in Section 3.2. Figure 6A and 6B shows the Linear Elevator and Tree Flag Elevator, respectively, where `trylock` stands for one of the three `trylock` versions of Section 3.1.

As we are unable to verify the twelve variations of the Elevator algorithm in a modular fashion, three of them are selected for verification. For consistent verification, different code variations have the same line numbers, if possible. Therefore, the simpler variations have gaps in the numbering.

The linear CAS elevator without flags is verified in Section 5 for safety and Section 6 for progress. The latter section also contains a brief introduction to Bounded Unity. The Tree CAS Flag elevator is verified in Section 7 for safety and in Section 8 for progress. The Tree LF Flag elevator is verified in Section 9.

As the algorithms differ mainly in how progress is made, the proofs of MX are almost identical. The proofs of absence of deadlock states prepare the way for the proofs of progress. The three proofs of progress all have the same structure: first general progress by means of a variant function, then individual progress beginning with bounded overtaking, which is the main work, and concluded by the treatment of the doorway.

5 | SAFETY OF THE LINEAR CAS ELEVATOR

This section verifies the safety of the linear CAS elevator without flags, ie, MX and absence of deadlock states.

5.1 | The transition system

For the purpose of the verification, the algorithm of the linear CAS elevator of Figure 6A is transformed into the transition system given in Figure 7. It uses the shared variables `fast`, `first`, `apply`. In the transition system, history variables, `mu`: 0 ... N - 1 and `inc`: N, are added. The CAS implementation of `trylock` is replaced by its meaning in Formula 1.

Line numbers are introduced to distinguish the atomic steps of the algorithms. Consistent line numbering is used between the PVS proof and paper (which is why numbering starts at 21). The PVS proof script is available for inspection and reuse in the work of Hesselink.^{16,†} Every line number

[†]When there are more than 4 threads, the model checkers we have worked with are unable to say anything sensible about the elevator algorithms. Perhaps, newer model checkers can do symbolic model-checking that may work.

```

var  $\mu$  : 0 ...  $N - 1$  ;

thread( $p$  : 0 ...  $N - 1$ ) =
  var  $k$  : 0 ...  $N - 1$  ;
  var  $pc, inc$  :  $\mathbb{N}$  ;
21  NCS ;
22  apply[ $p$ ]  $\leftarrow$  true ;
36  if  $\neg$  fast then
    fast  $\leftarrow$  true ;  $\mu \leftarrow p$  ;
37    await first =  $p \vee$  first =  $N$  ;
38    first  $\leftarrow p$  ;
39    fast  $\leftarrow$  false ;
    else
40    await first =  $p$  ;
    endif ;
43  CS ;
     $k \leftarrow N - 1$  ;
44  while  $\neg$  apply[( $p + k$ ) mod  $N$ ] do  $k \leftarrow k - 1$  endwhile ;
45  apply[ $p$ ]  $\leftarrow$  false ;
46  first  $\leftarrow$  ( $k \neq 0 ? (p + k)$  mod  $N$  :  $N$ ) ;
    inc  $\leftarrow$  inc + 1 ; goto 21 .

Initial condition:
 $\neg$  fast  $\wedge$  first =  $N \wedge (\forall q : pc.q = 21 \wedge \neg$  apply[ $q$ ]) .

```

FIGURE 7 The transition system for the linear CAS elevator

corresponds to an atomic step of the transition system. According to the principle of single critical reference, an atomic step has not more than one read or write action of a shared variable, except when the step is provided by special hardware. For instance, step 36 of Figure 7 reads and (possibly) writes the shared variable `fast`, which is allowed by the CAS operation. Step 36 also writes the shared variable `$\mu$` , which is allowed because `$\mu$` is a history variable that does not occur in the program but is only used in the proof. It is introduced for the thread that did the latest successful CAS operation. The atomic step corresponding to line 44 is the combination of an inspection of the guard with a single execution of the loop body. It would be represented more faithfully by the line

```
44  if  $\neg$  apply[( $p + k$ ) mod  $N$ ] then  $k \leftarrow k - 1$  ; goto 44 endif.
```

In line 46, a private history variable `inc` is introduced to count the number of times thread p returns to the NCS.

5.2 | Mutual exclusion

Mutual exclusion means there is never more than one thread in the CS. In other words, if thread q is in the CS, it is the only one: every thread in the CS is equal to q . Therefore, mutual exclusion for the algorithm is formally specified as invariant validity of the predicate

$$MX : q \text{ at } 43 \wedge r \text{ at } 43 \Rightarrow q = r.$$

Here and henceforth, we use implicit universal quantification over all free variables, here, q and r . In order to prove MX , we postulate the simpler and stronger invariant

$$lq0 : q \text{ in } \{39\} \cup \{41 \dots 46\} \Rightarrow \text{first} = q.$$

Indeed, predicate MX follows by applying $lq0$ to both threads q and r . Note that $lq0$ implies that mutual exclusion holds for the whole stretch $\{39\} \cup \{41 \dots 46\}$. The locations 41 and 42 are used in the other algorithms.

The proof of $lq0$ is typical. It is proved first that $lq0$ holds initially. Subsequently, it is proved that $lq0$ is preserved under all possible steps of the algorithm by any of the threads. For some steps of the algorithm, it is not easy to see that the precondition $lq0$ implies the postcondition $lq0$, which is expressed by writing that $lq0$ is threatened by the step. When a predicate is threatened by some step, usually another invariant is needed in the precondition. A predicate is called *inductive* iff it holds initially and is preserved by every step, without relying on other invariants in the precondition.

Predicate $lq0$ holds initially because of $pc.q = 21$. It is threatened by the assignments to `first` in steps 38 and 46. It is preserved by step 46 of thread p , because $lq0$ implies that thread p is the only thread with `first =  $p$` . It is preserved by step 38 because of the invariant

$$lq1 : q \text{ at } 38 \Rightarrow \text{first} = q \vee \text{first} = N.$$

From now onward, the verification that the invariant holds initially is no longer mentioned in the text. It has been done in the mechanical verification, and readers can do it themselves.

Predicate $lq1$ is threatened only by the steps 38 and 46. It is preserved by step 46 because of $lq0$. It is preserved by step 38 because of the invariant

$$lq2 : q \text{ in } \{37 \dots 39\} \Rightarrow \mu = q.$$

Predicate $lq2$ is threatened only by step 36. It is preserved because of the invariant

$$lq3 : q \text{ in } \{37 \dots 39\} \Rightarrow \text{fast}.$$

Predicate $lq3$ is threatened only by step 39. It is preserved because of $lq2$. This concludes the proof of MX .

Remark 2. The region $\{39\} \cup \{41 \dots 46\}$ for mutual exclusion is somewhat inconvenient. The line numbers of the branches of the conditional statement of line 36 can be swapped to obtain a coherent region. This however would make it less clear that the assignments to fast and μ are done in the same atomic command 36 as the reading of fast .

The invariant $lq2$ implies that the region 37–39 also satisfies mutual exclusion. These regions of mutual exclusion cannot be combined. Indeed, it is possible that there is a thread at 37, while another thread traverses 43–46.

5.3 | Absence of deadlock states

A thread is said to be idle when it is at the NCS , ie, at line 21. A thread is competing iff it is not idle (but possibly in the CS). A deadlock state is a state in which there are competing threads and none of them can do a step. Proving the absence of deadlock states is often a first step for the proof of progress. For the elevator algorithms, it is the most difficult step. Absence of deadlock states is a safety property that may require several invariants.

Every thread is always in one of the locations of the transition system, which is expressed by the invariant

$$Jq0a : q \text{ in } \{21, 22\} \cup \{36 \dots 40\} \cup \{43 \dots 46\}.$$

The suffix a of $Jq0a$ is introduced to distinguish $Jq0a$ from invariants for other variations of the algorithm.

When a thread is idle, ie, at line 21, it may stay there forever. A thread that is elsewhere will eventually be scheduled and then does a step if enabled. To distinguish them, step 21 is called an environment step, and all other steps are called forward steps. A thread is said to be enabled if it can do a forward step. Using $Jq0a$ and the transition system, it is easy to see that enabledness of thread q is equivalent to

$$\begin{aligned} \text{ena}(q) = & (q \text{ not-at } 21) \\ & \wedge (q \text{ at } 37 \Rightarrow \text{first} = q \vee \text{first} = N) \\ & \wedge (q \text{ at } 40 \Rightarrow \text{first} = q). \end{aligned}$$

A state is a deadlock state iff all threads are disabled, and not all threads are idle.

A MX algorithm needs to preserve a balance between MX and deadlock freedom. Therefore, some implications needed for MX must be inverted for deadlock freedom. In particular, the implication of $lq3$ needs to be inverted: when fast holds, there is a thread in 37–39. As an existential quantification in an invariant complicates the proof, the history variable μ is introduced with the invariant

$$Jq1 : \text{fast} \Rightarrow \mu \text{ in } \{37 \dots 39\}.$$

This predicate is inductive.

It is clearly relevant to know if first is a thread identifier and, if so, where the thread can be, which is postulated in the invariant

$$Jq2a : \text{first} = q \Rightarrow q \text{ in } \{23 \dots 46\}.$$

Predicate $Jq2a$ holds initially because of $\text{first} = N$, and N is not a thread. It is threatened only by step 46. It is preserved by step 46 because of the invariants

$$\begin{aligned} Jq3a : & \text{apply}[q] \Rightarrow q \text{ in } \{23 \dots 45\}, \\ Jq4a : & q \text{ in } \{45, 46\} \wedge k.q \neq 0 \Rightarrow \text{apply}[(q + k.q) \bmod N]. \end{aligned}$$

Predicate $Jq3a$ is inductive. Predicate $Jq4a$ is threatened by step 45. It is preserved because of $lq0$.

A rather unexpected invariant is

$$Jq5 : q \text{ at } 40 \wedge \text{first} = N \Rightarrow \text{fast}.$$

This predicate is threatened by the steps 39 and 46. It is preserved by step 39 because of $lq0$. It is preserved by step 46 because of the invariant

$$Jq6a : q \text{ at } 40 \wedge r \text{ in } \{44 \dots 46\} \wedge \neg \text{fast} \Rightarrow (q - r) \bmod N \leq k.r.$$

Predicate $Jq6a$ is threatened only by the steps 39 and 44. It is preserved by step 39 because of $lq0$. It is preserved by the decrement of k in step 44 because of the converse of $Jq3a$:

$$Jq7a : q \text{ in } \{23 \dots 45\} \Rightarrow \text{apply}[q].$$

Predicate $Jq7a$ is inductive. At this point, everything is prepared for the proof of absence of deadlock states:

Theorem 1. Assume all threads are disabled. Then all threads are idle.

Proof As all threads are disabled, the definition of enabledness together with $Jq0a$ implies that all threads are in the locations 21, 37, and 40. If $\text{first} < N$, there is a thread q with $q = \text{first}$. Then, $Jq2a$ implies that q is not at 21, and is therefore at 37 or 40, and is therefore enabled. This proves that $\text{first} = N$.
 If fast holds, then $Jq1$ implies that thread μ is in 37–39, and therefore at 37. Then, thread μ is enabled because $\text{first} = N$.
 It follows that fast is false. Then, there are no threads at 37 because of $Jq3$ and no threads at 40 because of $Jq5$. Therefore, all threads are at 21, that is, are idle. $\square$

6 | PROGRESS FOR THE LINEAR CAS ELEVATOR

In general, progress of an algorithm means that its aims are established when it has done sufficiently many steps. Progress can be expressed and proved in various forms of temporal logic. Our experience is that admitting operational arguments for a specific concurrent algorithm leads to confusing and unreliable proofs. It is therefore better to use a theory that treats the operational arguments at a general level and allows only assertional reasoning for the specific algorithm.

Progress for the elevator algorithms is proved with Bounded Unity,^{17–19} which is a method to prove and quantify progress properties by only analyzing states and step relations. It is a quantitative version of UNITY.^{20,21} To distinguish the two versions, the acronym UNITY is used only for the original version.

Section 6.1 gives the operational background that is used only to interpret the questions and results, and as a foundation for the mechanical proofs. Section 6.2 presents the basic proof rules of Bounded Unity.

6.1 | The operational foundation

The state of a system is given by the values of all shared and private variables. Let X be the set of all states. If P is a predicate on the state, P is also regarded as a subset of X , viz, as the set of the states that satisfy P . In particular, the invariants of Sections 5.2 and 5.3 are subsets of X . Let INV be the intersection of all invariants obtained. Although predicates on X are now identified with subsets of X , we mostly keep to predicate notation: $P \wedge Q$ and $P \vee Q$ for conjunction and disjunction, $\neg P$ for negation. The main exception is that, if P and Q are predicates, the set inclusion $P \subseteq Q$ is used to express that P implies Q .

Let step be the binary relation on X such that $(x, y) \in \text{step}$ iff state x has some transition to state y or $x = y$. An execution fragment of length $n \geq 0$ is a nonempty finite sequence $(x_0, \dots, x_n)$ in the set INV such that $(x_i, x_{i+1}) \in \text{step}$ for all $0 \leq i < n$. Two execution fragments can be concatenated iff the final state of the first fragment is equal the initial state of the second fragment.

Recall from Section 5.3 that the forward steps of thread p are those that do not start at line 21, and that $\text{ena}(p)$ is the set of the states x where thread p can do a forward step. Let $\text{fwd}(p)$ be the set of pairs (x, y) such that in state x thread p can do a forward step toward y .

Thread p is defined to occur in an execution fragment $(x_0, \dots, x_n)$ iff there is an index i with $0 \leq i < n$, and $(x_i, x_{i+1}) \in \text{fwd}(p)$ or $x_i \notin \text{ena}(p)$. An execution fragment is called a round iff it contains an occurrence for every thread. Informally speaking, in a round, every thread can be regarded as scheduled at least once, and to be either executed or found to be disabled.

The key concept of Bounded Unity is the quantified leads-to relation: predicate P leads to predicate Q within n rounds, notation $P \text{ LT } \langle n \rangle Q$. This relation means that every execution fragment that contains a concatenation of n rounds and has its initial state in P , contains a state in Q . The number n is called the concurrent complexity of reaching Q from P .

6.2 | Bounded Unity

The logic of Bounded Unity begins, just as UNITY logic, with the definitions of a relation **co** and a judgment **trans** (for **transient**) for predicates

$$\begin{aligned} P \text{ co } Q &\equiv \forall (x, y) \in \text{step} : x \in P \wedge INV \Rightarrow y \in Q, \\ \text{trans}(P) &\equiv \exists r : (P \wedge INV) \subseteq \text{ena}(r) \wedge (\forall (x, y) \in \text{fwd}(r) : x \in P \wedge INV \Rightarrow y \notin P). \end{aligned}$$

$P \text{ co } Q$ means that every step that starts in P ends in Q . The judgement $\text{trans}(P)$ expresses there is a specific thread r that, from every state of P , establishes $\neg P$. For UNITY, the operators **co** and **trans** are introduced by Misra²¹ with the same informal meanings. The definitions differ, however, because UNITY does not have step , INV , ena , and fwd .

The operators **co** and **trans** are combined in the relations **unless** and **ensures** defined by

$$\begin{aligned} P \text{ unless } Q &\equiv (P \wedge \neg Q) \text{ co } (P \vee Q), \\ P \text{ ensures } Q &\equiv (P \text{ unless } Q) \wedge \text{trans}(P \wedge \neg Q). \end{aligned}$$

At this point, Bounded Unity deviates more strongly from UNITY by defining the leads-to operator **LT** with a numerical parameter. The basic proof rules for **LT** are

- If P ensures Q , then $P \text{ LT } \langle 1 \rangle Q$.
- If $(P \wedge \text{INV}) \subseteq Q$, then $P \text{ LT } \langle n \rangle Q$ for every $n \geq 0$.
- If $P \text{ LT } \langle k \rangle Q$ and $Q \text{ LT } \langle m \rangle R$, then $P \text{ LT } \langle k + m \rangle R$.
- If $P_i \text{ LT } \langle n \rangle Q$ holds for all $i \in \mathbb{I}$, then $(\exists i \in \mathbb{I}: P_i) \text{ LT } \langle n \rangle Q$.

The first rule is called the **ensures** rule, the second one is the subset rule, the third one is called transitivity, and the last one is the disjunction rule.

There is also the Progress-Safety-Progress Rule²⁰:

$$\text{PSP} : (P \text{ LT } \langle n \rangle Q) \wedge (A \text{ unless } M) \Rightarrow (P \wedge A) \text{ LT } \langle n \rangle ((Q \wedge A) \vee M).$$

The soundness of these proof rules for the operational semantics of Section 6.1 has been proved mechanically with PVS.¹⁶ For later use, the transitivity rule for **unless** is mentioned

$$(P_1 \text{ unless } (P_2 \vee Q)) \wedge (P_2 \text{ unless } Q) \Rightarrow ((P_1 \vee P_2) \text{ unless } Q). \quad (4)$$

6.3 | Throughput

In the transition system of Figure 7, the history variable $\text{inc}.q$ counts how many times thread q has returned to NCS, indicating the progress of thread q . The total progress of the system is the sum $\text{sinc} = \sum_q \text{inc}.q$.

The progress of thread q is related to the number of steps of q . To estimate this number, we define

$$\text{lvf}(q) = (36 \leq \text{pc}.q ? \text{pc}.q - 35 : 0) + (40 \leq \text{pc}.q ? -3 : 0) + (44 \leq \text{pc}.q ? N - 1 - k.q : 0).$$

By Jq2a, $\text{lvf}(q)$ is bounded by

$$0 \leq \text{lvf}(q) < A_1 \text{ where } A_1 = N + 8. \quad (5)$$

Moreover, $\text{lvf}0$, $\text{lvf}1$, and $\text{lvf}2$ collectively imply mutual exclusion for $\text{lvf} \geq 3$:

$$\text{lvf}(q) \geq 3 \wedge \text{lvf}(r) \geq 3 \Rightarrow q = r. \quad (6)$$

Using Jq7a at line 44, it is possible to show that the value of $\text{lvf}(q)$ increases with every forward step of q , except for step 46. It follows that the function $\text{avf}(q) = \text{lvf}(q) + A_1 \cdot \text{inc}.q$ increases in every forward step of q . On the other hand, $\text{avf}(q)$ remains constant under the environment step of thread q , and under all steps of threads $\neq q$. It follows that the sum $\text{savf} = \sum_q \text{avf}(q)$ increases under every forward step of any thread and remains constant under the environment steps.

Let AI be the predicate that expresses all threads are idle. If AI is false, Theorem 1 implies that there are enabled threads. Once an enabled thread does a forward step, it increases savf . In this way, it is proved that savf increases with the number of rounds unless AI becomes valid, as expressed in the formula

$$(m \leq \text{savf}) \text{ LT } \langle i \rangle ((m + i \leq \text{savf}) \vee AI).$$

The Formulas (5) and (6) imply that the function sinc is related to savf by

$$A_1 \cdot \text{sinc} \leq \text{savf} \leq A_1 \cdot \text{sinc} + 3 \cdot N + 5.$$

The last two formulas combine to the Throughput Theorem.

Theorem 2. $(k \leq \text{sinc}) \text{ LT } \langle A_1 \cdot j + C_1 \rangle ((k + j \leq \text{sinc}) \vee AI)$ where $A_1 = N + 8$ and $C_1 = 2 \cdot N - 2$.

This result implies that, for a large number r of rounds, the system reaches the all-idle state AI or function sinc increases with at least r/A_1 . Therefore A_1 is called the throughput factor. The additional term C_1 can be regarded as initial delay.

6.4 | Individual progress for the linear CAS elevator

Individual progress (also called lockout freedom²²) is the property of the algorithm that every competing thread eventually reaches the CS and returns to the NCS.

The stages for thread q in the algorithm are

$$\begin{aligned} \text{Doorway}(q) &= (q \text{ at } 22), \\ \text{Inside}(q) &= (q \text{ in } \{36 \dots 46\}), \\ \text{Idle}(q) &= (q \text{ at } 21). \end{aligned}$$

Theorem 3. $(\text{Inside}(q) \wedge \text{sinc} = m) \text{ LT } \langle dr \rangle (\text{Idle}(q) \wedge \text{sinc} \leq m + N + 1)$, where $dr = N^2 + 11 \cdot N + 6$.

This result implies bounded overtaking. In fact, the result implies that, once a thread q has passed the Doorway, the number of times a different thread can enter the CS is bounded by N , because it exits and increments $sinc$ before another thread can enter the CS. There is indeed a scenario in which q passes the doorway, after which $N - 1$ threads $\neq q$ exit before q exits. The number dr is an upper bound of the number of rounds.

The proof of the theorem is based on Theorem 2 in combination with the fact that $sinc$ cannot grow too much while thread q remains waiting. The same idea is used below in the proofs of individual progress for the other elevator variations.

Proof. The aim is to prove that the system proceeds within dr rounds from the precondition $pre0(q, m)$ to some postcondition $goal(q, k)$ where

$$\begin{aligned} pre0(q, m) &= (Inside(q) \wedge sinc = m), \\ goal(q, k) &= (Idle(q) \wedge sinc \leq k). \end{aligned}$$

Consider a thread q with $Inside(q)$. Thread q can only become idle from line 46, when it is `first` because of $lq0$. This observation suggests the predicate

$$pre3(q, m) = (q = \text{first} \wedge sinc \leq m).$$

When $q = \text{first}$, it becomes idle in step 46, where it increments $sinc$. In the mean time, no other thread can modify $sinc$. This implies that

$$pre3(q, m) \text{ unless } goal(q, m + 1). \quad (7)$$

This question now is how can `first` reach q ? The value of `first` is modified in every step 46, and it usually jumps to the left toward q . The number of jumps to the left is bounded by the distance

$$dist(q) = (\text{first} < N ? (\text{first} - q) \bmod N : N - 1).$$

In this way, $0 \leq dist(q) \leq N - 1$ always holds. A complication, however, is that thread `first` may have passed q in its loop 44 before `apply[q]` is set to true. To exclude this possibility for the moment, we introduce the predicate

$$Run(q) = (Inside(q) \wedge (\text{first} = N \vee pc.\text{first} \leq 43 \vee N \leq k.\text{first} + dist(q))).$$

When $Run(q)$ holds, the condition $\text{first} = N$ can only be invalidated by step 38, which makes $pc.\text{first} \leq 43$ true. The condition $pc.\text{first} \leq 43$ is only invalidated by step 43, which makes $N \leq k.\text{first} + dist(q)$ true because $dist(q) > 0$ and $k.\text{first}$ becomes $N - 1$. The condition $N \leq k.\text{first} + dist(q)$ is not invalidated by step 44 because when $N = k.\text{first} + dist(q)$ then $(\text{first} + k.\text{first}) \bmod N = q$ and `apply[q]` holds because of $Jq7a$ and $lq0$. It follows that $Run(q)$ can only be invalidated when `first` does step 46 and increments $sinc$. This decrements $dist(q)$, and it may keep $Run(q)$ valid. Otherwise, `first` becomes q . In this way, it is proved that

$$\begin{aligned} pre2(q, m) &\text{ unless } pre3(q, m), \\ \text{where } pre2(q, m) &= (Run(q) \wedge sinc + dist(q) \leq m). \end{aligned}$$

Combining this formula and Formula (8) using the rule (4), results in

$$\begin{aligned} pre23(q, m) &\text{ unless } goal(q, m + 1), \\ \text{where } pre23(q, m) &= (pre2(q, m) \vee pre3(q, m)). \end{aligned} \quad (8)$$

Now, work backward to the final case, where no more is known about $dist(q)$ than $dist(q) \leq N - 1$, giving

$$\begin{aligned} pre1(q, m) &\text{ unless } pre23(q, m + N) \vee goal(m + N + 1), \\ \text{where } pre1(q, m) &= (Inside(q) \wedge sinc \leq m + 1 \wedge (sinc \leq m \vee \text{first} = N)). \end{aligned} \quad (9)$$

Combining the formulas (8) and (9) with formula (4) yields

$$\begin{aligned} pre123(q, m) &\text{ unless } goal(m + N + 1), \\ \text{where } pre123(q, m) &= (pre1(q, m) \vee pre23(q, m + N)). \end{aligned} \quad (10)$$

This **unless** relation is combined with Theorem 2 with $k \leftarrow m$ and $j \leftarrow N + 1$ via the PSP rule, giving

$$\begin{aligned} (m \leq sinc \wedge pre123(q, m)) &\text{ LT } \langle A_1 \cdot (N + 1) + C_1 \rangle \\ ((m + N + 1 \leq sinc \vee AI) \wedge pre123(q, m)) &\vee goal(q, m + N + 1). \end{aligned}$$

In this formula, the precondition can be replaced by $pre0(q, m)$, because this is a subset of $pre123(q, m)$ and of $(m \leq sinc)$. The postcondition can be replaced by $goal(q, m + N + 1)$, because $AI \wedge pre123(q, m)$ is contained in $goal(q, m + N + 1)$. The resulting upper bound for the number of rounds is $(N + 8) \cdot (N + 1) + 2 \cdot N - 2 = N^2 + 11 \cdot N + 6$. $\square$

Using the disjunction rule, Theorem 3 implies

$$Inside(q) \text{ LT } \langle N^2 + 11 \cdot N + 6 \rangle Idle(q).$$

As every thread passes the Doorway within a single round, it follows

Theorem 4. $\text{true } \text{LT}(F_1) \text{Idle}(q)$ where $F_1 = N^2 + 11 \cdot N + 7$.

which means, wherever thread q may be now, it will be idle within $F_1 = N^2 + 11 \cdot N + 7$ rounds. Therefore F_1 is called the individual delay.

7 | SAFETY OF THE TREE CAS FLAG ELEVATOR

The tree CAS flag elevator differs from the linear CAS elevator in that the code is extended with flags in the way described in Section 3.2, and the final loop of order N over the array `apply` is replaced by two loops of order $\log(N)$ along the root path of a binary tree, see Section 3.3.2 and Figure 6B.

In this section, the safety properties of the Tree CAS Flag elevator are verified. Several of the invariants of the linear CAS flag elevator are reused. The transition system is developed in Section 7.1. The first steps of the verification are taken in Section 7.2. The proof of MX of the linear CAS flag elevator is reused completely, but the queue and the two loops over the root path need new invariants. The main work is in the proof of absence of deadlock states in Section 7.3.

7.1 | The transition system of the Tree CAS Flag elevator

Just as above, the verification begins with transformation of the code of Figure 6B into the transition system of Figure 8. The first thing to note is that the private variable j of thread p is made persistent with initial value $\text{depth}(p) + 1$. The history variable `mu` is used in the same way as in Figure 7. A history variable `first` is used in the same role as the actual variable `first` in Figure 7.

The values at the nodes of the tree are contained in array `val`. This array has a hybrid role in the algorithm because the leaves and the internal nodes are treated in different ways. In the loop of line 22, thread p writes its thread identifier at all nodes of its root path. It follows that, at the internal nodes, this identifier can be overwritten by other threads. The value at leaf $N + p$ is not overwritten in this loop. This value is therefore inspected at line 45. It is reset by thread p in line 41 and possibly by a different thread in line 45. In the loop of lines 44–45, thread p inspects the siblings of its root path and puts the values read at the end of the queue if $\text{val}[N + k] \neq N$.

The queue is represented as a finite sequence `queue` of thread identifiers that is initially equal to the empty sequence ϵ . The function `add` used in line 45 adds the thread k at the end of the finite sequence. The function `head` yields the first element of the sequence. The function `tail` removes the first element from the sequence.

```

var val : array[node] of 0 ... N;
var queue : finite sequence of 0 ... N - 1;
var mu, first : 0 ... N; // history variables

thread(p : 0 ... N - 1) =
  var k : 0 ... N;
  var j, pc, inc : N;
21  NCS;
22  while j > 1 do j ← j - 1; val[path(p, j)] ← p; endwhile;
36  if ¬fast then
      fast ← true; mu ← p;
37  await flag[N] ∨ flag[p];
38  flag[N] ← false; first ← p;
39  fast ← false;
  else
40  await flag[p];
  endif
41  val[N + p] ← N;
42  flag[p] ← false;
43  CS;
44  while j ≤ depth(p) do
      k ← val[sib(path(p, j))];
45  if k < N ∧ val[N + k] ≠ N then // enqueue
      queue ← add(queue, k);
      val[N + k] ← N; endif;
      j ← j + 1;
  endwhile;
46  if queue ≠ ε then // dequeue
      k ← head(queue); queue ← tail(queue);
  else k ← N endif;
  flag[k] ← true; first ← k; inc ← inc + 1; goto 21.

Initial condition:
¬fast ∧ first = N ∧ flag[N] ∧ queue = ε ∧ (∀ n ∈ node: val[n] = N)
  ∧ (∀ q: pc.q = 21 ∧ j.q = depth(q) + 1 ∧ ¬flag[q]).

```

FIGURE 8 The transition system for the tree CAS flag elevator

Warning. In view of the loops, it is assumed that $\text{depth}(p) \geq 1$ for all threads p . This assumption holds if $N > 1$. The algorithm is also correct for $N = 1$, but the proof does not apply.

7.2 | The first steps for verification of safety

The invariants $lq0$ up to $lq3$ of the Section 5.2 are retained without change. Just as in Section 5.2, mutual exclusion MX is implied by $lq0$. Note that the locations 41 and 42 in the invariant $lq0$ now make sense. Preservation of $lq0$ by step 40 now, however, requires the new invariant

$$lq4 : \text{flag}[t] \Rightarrow \text{first} = t,$$

where t ranges over $0 \dots N$. Predicate $lq4$ is threatened only by the steps 38 and 46. It is preserved by step 38 because of $lq1$. It is preserved by step 46 because of $lq0$ and the new invariant

$$lq5 : q \text{ in } \{43 \dots 46\} \Rightarrow \neg \text{flag}[q].$$

Predicate $lq5$ is threatened only by step 46. It is preserved because of $lq0$. This concludes the proof of MX .

7.3 | Absence of deadlock states

The proof of absence of deadlock states is much more difficult for the tree elevators than for the linear elevators. Predicate $Jq0a$ needs to be updated to

$$Jq0b : q \text{ in } \{21, 22\} \cup \{36 \dots 46\}.$$

Predicate $Jq1$ is retained, and it is still inductive. Predicate $Jq2a$ has to be changed to:

$$Jq2b : \text{first} = q \Rightarrow q \text{ in } \{22 \dots 46\}.$$

In order to prove that it is preserved by step 46, an invariant about the queue is needed

$$Kq0 : r \in \text{qset} \Rightarrow r \text{ in } \{22 \dots 37\} \cup \{40\}.$$

Here qset is the set of the elements of the queue

$$\text{qset} = \{r \mid \exists i : i < \text{length}(\text{queue}) \wedge \text{queue}[i] = r\}.$$

Predicate $Kq0$ is threatened only by the steps 37, 40, and 45. It is preserved by the steps 37 and 40 because of $lq4$ and the new invariant

$$Kq1 : r \in \text{qset} \Rightarrow \text{first} \neq r \wedge \text{first} \neq N.$$

Predicate $Kq0$ is preserved by step 45 because of $lq0$, $lq1$, and the new invariant

$$Jq3b : q \in \text{apply} \Rightarrow q \text{ in } \{22 \dots 41\},$$

where apply is the set given by

$$\text{apply} = \{k \in 0 \dots N - 1 \mid \text{val}[N + k] \neq N\}.$$

This set plays the same role as the Boolean array apply in the linear elevator.

Predicate $Kq1$ is threatened only by the steps 38, 45, and 46. It is preserved by step 38 because of $lq1$, and by step 45 because of $lq0$ and $Jq3b$. It is preserved by step 46 because the queue contains no duplicates

$$Kq2 : \forall i, j < \text{length}(\text{queue}) : \text{queue}[i] = \text{queue}[j] \Rightarrow i = j.$$

Predicate $Jq3b$ is inductive. Indeed, it is preserved by step 22B (the body of the loop) because node $N + q$ is a leaf of the tree.

Predicate $Kq2$ is threatened only by step 45, which adds a new element to the queue. It is preserved because of the new invariant

$$Kq3 : \text{apply} \cap \text{qset} = \emptyset.$$

Predicate $Kq3$ is threatened only by step 22 when $\text{val}[N + p]$ is set. It is preserved because of the new invariant

$$Kq4 : q \in \text{qset} \Rightarrow j.q \leq \text{depth}(q).$$

Predicate $Kq4$ is threatened only by step 45. It is preserved because of $Kq0$, $Jq3b$, and the new postulate

$$Jq4b : q \in \text{apply} \Rightarrow j.q \leq \text{depth}(q).$$

Predicate $Jq4b$ is threatened only by the steps 22B and 45. It is preserved at 45 because of $Jq3b$. It is preserved at 22B because of the new invariant

$$Kq5 : j.q \leq \text{depth}(q) + 1.$$

Predicate $Kq5$ is threatened only by step 45. It is preserved because of

$$Kq6 : \quad q \text{ at } 45 \Rightarrow j.q \leq \text{depth}(q).$$

Concerning the loop variable j , we also postulate

$$Kq7 : \quad q \text{ in } \{23 \dots 43\} \Rightarrow j.q = 1,$$

$$Kq8 : \quad q \text{ in } \{21, 46\} \Rightarrow j.q = \text{depth}(q) + 1.$$

Predicates $Kq6$ and $Kq7$ are inductive. Predicate $Kq8$ is threatened only by step 44. It is preserved because of $Kq5$.

For the flags, the following inductive invariants are needed:

$$Jq8 : \quad q \text{ in } \{\dots 38\} \cup \{40\} \wedge q = \text{first} \Rightarrow \text{flag}[q],$$

$$Jq9 : \quad \text{first} = N \Rightarrow \text{flag}[N].$$

For the absence of deadlock states, note that enabledness is still equivalent to predicate ena given in Section 5.3. It turns out that the proof of Theorem 1 can easily be adapted if the invariants $lq3$, $Jq2a$, $Jq1$, $Jq5$, $Jq8$, and $Jq9$ are available.

It remains to prove validity of $Jq5$. As $\text{first} = N$ implies that the queue is empty by $Kq1$, predicate $Jq5$ is implied by $Kq1$ together with the new invariant

$$Lq0b : \quad q \text{ at } 40 \wedge \neg \text{fast} \wedge \text{queue} = \varepsilon \Rightarrow \text{first} \neq N \wedge \text{first in } \{22 \dots 45\}.$$

The idea of $Lq0b$ is that thread first will make queue nonempty in the loop 44–45, because of the location of q . Predicate $Lq0b$ is threatened only by the steps 39, 44, and 46. It is preserved by step 39 because of $lq0$. It is preserved by step 46 because of $lq0$ and $Kq0$. It is preserved by step 44 because of $lq0$ and the postulate

$$Lq1b : \quad q \text{ at } 40 \wedge r \text{ in } \{44, 45\} \wedge \neg \text{fast} \wedge \text{queue} = \varepsilon \Rightarrow j.r \leq \text{meet}(q, r) + 1.$$

Here, recall from Section 3.3.2 that $\text{meet}(q, r)$ is the level where the root paths of q and r meet, ie, the greatest integer i with $i \leq \text{depth}(q)$ and $i \leq \text{depth}(r)$ and $\text{path}(q, i) = \text{path}(r, i)$. Therefore, if $q \neq r$, then $\text{meet}(q, r) < \text{depth}(r)$, which implies that thread q cannot terminate its loop 44–45 while the antecedent of $Lq1b$ holds.

Predicate $Lq1b$ is threatened only by the steps 39, 43, 45, and 46. It is preserved by steps 39 and 46 because of $lq0$. It is preserved by step 43 because of $Kq7$. It is preserved by step 45 because of

$$Lq2b : \quad q \text{ at } 40 \wedge r \text{ at } 45 \wedge \neg \text{fast} \wedge \text{queue} = \varepsilon \wedge j.r = \text{meet}(q, r) + 1 \Rightarrow k.r \in \text{apply}.$$

Predicate $Lq2b$ is threatened only by the steps 39, 41, 44, 45, and 46. The steps 39, 41, 45, and 46 preserve $Lq2b$ because of $lq0$. It is preserved by step 44 because of the new postulate

$$SibAp : \quad q \text{ in } \{23 \dots 40\} \wedge r \text{ at } 44 \wedge \text{queue} = \varepsilon \Rightarrow \text{val}[\text{sib}(\text{path}(r, \text{meet}(q, r) + 1))] \in \text{apply}.$$

Here, the range for q is extended to also serve in the next variation of the algorithm.

At this point, it is useful to postulate

$$Mq0 : \quad j.q \leq \text{depth}(q) \Rightarrow q \in \text{apply} \vee q \in \text{qset} \vee q = \text{first}.$$

Predicate $Mq0$ is threatened only by the steps 38, 41, and 46. It is preserved because of $lq0$, $lq1$, and $Kq8$.

In preparation of the proof of $SibAp$, some innocent invariants about the values at the nodes are established first.

$$Mq1 : \quad \text{val}[N + q] = N \vee \text{val}[N + q] = q,$$

$$Mq2 : \quad j.q \leq i < \text{depth}(q) \Rightarrow \text{val}[\text{path}(q, i)] < N.$$

Both predicates are inductive.

If the root path of thread q above level $j.q$ holds a node with the value of thread r , the node is also in the root path of thread r , which is expressed by the invariant

$$Mq3 : \quad j.q \leq i \leq \text{depth}(q) \wedge n = \text{path}(q, i) \wedge r = \text{val}[n] \Rightarrow i \leq \text{depth}(r) \wedge n = \text{path}(r, i).$$

Predicate $Mq3$ is threatened only by step 22B. It is preserved because of $Kq5$.

In order to prove $SibAp$, we observe that the loop at 22 tries to fill the root path of thread p with values in apply . This suggests the invariant

$$Nq1 : \quad q \in \text{apply} \wedge j.q \leq i < \text{depth}(q) \wedge \text{level}(q) \leq i \Rightarrow \text{val}[\text{path}(q, i)] \in \text{apply}.$$

for some function level , which is yet to be determined.

This predicate is threatened by the steps 41 and 45 that remove elements from apply . It is also threatened by step 22 of thread p when $p \notin \text{apply}$. In these cases, the value of $\text{level}(q)$ needs to be adapted because some thread, $r \notin \text{apply}$, appears on the root path of q . It turns out that this thread is a candidate for promoting thread q from the tree to the queue, called a friend of q .

This suggests an invariant that asserts, under certain conditions, the existence of friend of q . As stated in Section 5.3, invariants with an existential quantifier are usually difficult to prove but can often be simplified by introducing a history variable that serves as a witness of the existential quantification. This approach is used again.

For every thread q , a shared history variable $\text{friend}[q]$ of type $0 \dots N$ is introduced. The value $\text{friend}[q] = N$ means that q has no friend yet. The level of q is defined by

$$\text{level}(q) = (\text{friend}[q] = N ? 0 : \text{meet}(\text{friend}[q], q)) + 1.$$

The value of $\text{friend}[q]$ is only relevant when $q \in \text{apply}$. In step 22, when $q \in \text{apply}$ is made true, $\text{friend}[p]$ is set to N . Furthermore, array friend is modified by procedure

```
modfriend( $r : 0..N - 1$ ) =
  foreach  $q$  with  $\max(j.q - 1, \text{level}(q)) \leq \text{meet}(r, q)$  do
    friend[ $q$ ]  $\leftarrow r$ 
  endfor
end modfriend.
```

The steps 22, 41, and 45 are therefore modified as follows

```
22 body:  if  $j = \text{depth} + 1$  then friend[ $p$ ]  $\leftarrow N$ 
          else if val[ $N + p$ ] =  $N$  then modfriend( $p$ ) endif
           $j \leftarrow j - 1$ ; val[ $\text{path}(p, j)$ ]  $\leftarrow p$ ;
41 :      val[ $N + p$ ]  $\leftarrow N$ ; modfriend( $p$ );
45 body :  queue  $\leftarrow \text{add}(\text{queue}, k)$ ;
          val[ $k$ ]  $\leftarrow N$ ; modfriend( $k$ );
```

Function level can now be eliminated from $Nq1$ to give the equivalent version

$$Nq1 : \quad q \in \text{apply} \wedge j.q \leq i < \text{depth}(q) \wedge \text{val}[\text{path}(q, i)] \notin \text{apply} \Rightarrow \text{friend}[q] < N \wedge i \leq \text{meet}(\text{friend}[q], q).$$

Predicate $Nq1$ is threatened by the steps 22B, 41, and 45. It is preserved by step 22 because of $Kq5$. It is preserved by the steps 41 and 45 because of $Mq3$.

The usefulness of friends lies in the invariant

$$Nq2 : \quad q \in \text{apply}[q] \wedge \text{friend}[q] < N \Rightarrow \text{friend}[q] \in q\text{set} \vee \text{friend}[q] = \text{first}.$$

Before proving that $Nq2$ is an invariant, it is shown why $Nq2$ is useful.

Lemma 1. *Predicate SibAp follows from the invariants $Nq1$, $Nq2$, $Mq0$, $Mq1$, $lq0$, and $Kq7$.*

Proof. Assume that thread q is in 23–40, thread r is at 44, and the queue is empty. Put $i = \text{meet}(q, r) + 1$ and $n = \text{sib}(\text{path}(r, i))$. Then, it is to be proved that $\text{val}[n] \in \text{apply}$. Therefore, assume that $\text{val}[n] \notin \text{apply}$.

As $i = \text{meet}(q, r) + 1$, the node $\text{path}(q, i)$ is the sibling of $\text{path}(r, i)$. This implies that $n = \text{path}(q, i)$. By $lq0$, we have $r = \text{first}$. As threads q and r are at different location, it follows that $q \neq \text{first}$. As the queue is empty, $Mq0$ implies that $q \in \text{apply}$. The invariant $Kq7$ implies that $j.q = 1$, so that $j.q \leq i$. As q and r have different leaves of the tree, it holds that $\text{meet}(q, r) < \text{depth}(q)$. Therefore, $i \leq \text{depth}(q)$.

If $i = \text{depth}(q)$, then n is the leaf $N + q$ and $\text{val}[n] \neq N$ because $q \in \text{apply}$, and hence, $\text{val}[n] = q \in \text{apply}$ because of $Mq1$, which is a contradiction.

Otherwise, $i < \text{depth}(q)$. Now, the antecedent of $Nq1$ holds. This implies that q has a $\text{friend}[q] = f < N$ with $i \leq \text{meet}(f, q)$. By $Nq2$, it follows that f is in the queue or $f = r$. As the queue is empty, this implies $f = r$, and hence, $i \leq \text{meet}(r, q) = \text{meet}(q, r)$, a contradiction. $\square$

Predicate $Nq2$ is threatened only by the steps 22B, 38, 41, and 46. It is preserved by step 22B because of $Mq0$, by step 38 because of $lq1$, by step 41 because of $lq0$. It is preserved by step 46 because of $lq0$, $Jq3b$, $Kq8$, and the new postulate

$$Nq3 : \quad q \in \text{apply} \wedge r = \text{friend}[q] \wedge r \text{ in } \{23 \dots 46\} \Rightarrow j.r \leq \text{meet}(r, q) + 1.$$

Note that q and r always range over the threads. Therefore, $r \neq N$.

Predicate $Nq3$ is threatened only by the steps 41 and 45. It is preserved by step 41 because of $Kq7$. For the preservation of $Nq3$ by step 45, two invariants are postulated:

$$Nq4 : \quad q \in \text{apply} \wedge r = \text{friend}[q] \Rightarrow j.q \leq \text{meet}(r, q) + 1,$$

$$Nq5 : \quad q \in \text{apply} \wedge r = \text{friend}[q] \wedge \text{rat } 45 \wedge j.r = \text{meet}(r, q) + 1 \Rightarrow k.r \in \text{apply} \wedge j.r \leq \text{meet}(k.r, q).$$

Preservation of $Nq3$ by step 45 is shown as follows. Assume that step 45 of thread p invalidates $Nq3$. In the postcondition of the step, thread $r = \text{friend}[q]$ is in 23–46 and $\text{meet}(r, q) + 1 < j.r$. This implies that $r = p$ by $Kq7$ and $lq0$. By $Jq3b$, $p \notin \text{apply}$ in the precondition of the step. Therefore, $k.p \neq p$ and the step does not modify array $\text{friend}[q]$. Therefore, $\text{friend}[q] = p$ holds in the precondition. As $Nq3$ holds in the precondition, it

follows that $j.p = \text{meet}(p, q) + 1$ in the precondition. It follows that the antecedents of $Nq4$ and $Nq5$ hold. Therefore, the consequents hold. These imply that the call of $\text{modfriend}(k.r)$ modifies $\text{friend}[q] \leftarrow k.r$, a contradiction.

Predicate $Nq4$ is threatened only by the incrementation of $j.q$ in step 45. It is preserved because of $Jq3b$.

Predicate $Nq5$ is threatened only by the steps 41, 44, and 45. It is preserved by the steps 41 and 45 because of $lq0$.

Preservation of $Nq5$ by step 44 is shown as follows. Assume that step 44 of thread p invalidates $Nq5$. Step 44 of p does not modify apply , friend , or any private variable j , it only modifies $k.p$ and $pc.p$. As $Nq5$ holds in the precondition, it follows that $p = r$ and that the precondition satisfies

$$q \in \text{apply} \wedge r = \text{friend}[q] \wedge r \text{ at } 44 \wedge j.r = \text{meet}(r, q) + 1 \wedge \neg(x \in \text{apply} \wedge j.r \leq \text{meet}(x, q))$$

$$\text{where } x = \text{val}[\text{sib}(\text{path}(r, j.r))].$$

The threads q and r are different because of $Jq3b$. Therefore, $j.r \leq \text{depth}(q)$. As above, it follows that $\text{sib}(\text{path}(r, j.r)) = \text{path}(q, j.r)$. Predicate $Nq4$ gives $j.q \leq j.r$. Finally, $Nq1$ gives $x \in \text{apply}$, and $Mq3$ gives $\text{path}(x, j.r) = \text{path}(q, j.r)$, proving that $j.r \leq \text{meet}(x, q)$, which is a contradiction.

This concludes the proof of the invariants Nq^* , and thus of SibAp , and hence also of the invariants Lq^* , and therefore also of $Jq5$, and thus of the absence of deadlock states.

8 | PROGRESS FOR THE TREE CAS FLAG ELEVATOR

General progress and the estimate of throughput for the Tree CAS Flag elevator are treated in Section 8.1 in the same way as for the linear CAS Flag elevator. Individual progress and the estimate of individual delay are treated in Section 8.2. The proofs are more difficult than for the linear elevator because the arguments for absence of deadlock states in Section 7.3 need to be extended. On the other hand, this section carries over completely to the Tree LF Flag elevator that is treated in the next section.

8.1 | Throughput

The proof of throughput is similar to the treatment in Section 6.3, but now uses the function

$$\text{lvf}(q) = pc.q - 21 + (pc.q = 22 ? \text{depth}(q) + 1 - j.q : 0) + (pc.q > 22 ? \text{depth}(q) - 13 : 0)$$

$$+ (44 \leq pc.q \leq 45 ? 2 \cdot j.q - 2 : 0) + (pc.q = 46 ? 2 \cdot \text{depth}(q) - 1 : 0).$$

Using $Jq0b$, $Kq5$, and $Kq6$, one easily verifies that

$$0 \leq \text{lvf}(q) < A_2 \text{ where } A_2 = 3 \cdot D + 12 \text{ and } D = \max_q \text{depth}(q).$$

The remainder of Section 6.3 is used directly. The result is the Throughput Theorem for the tree CAS flag elevator:

Theorem 5. $(k \leq \text{sinc}) \text{ LT } \langle A_2 \cdot j + C_2 \rangle ((k + j \leq \text{sinc}) \vee A1)$ where $A_2 = 3 \cdot D + 12$ and $C_2 = A_2 \cdot N - A_2 + 1$.

It follows that, if the tree is balanced, the throughput factor A_2 is logarithmic in N .

8.2 | Individual progress for the Tree CAS Flag elevator

Let q be any thread. The aim is to show that q becomes idle. The main stages in the algorithm are

$$\text{Doorway}(q) = (q \text{ at } 22),$$

$$\text{Inside}(q) = (q \text{ in } \{23 \dots 46\}),$$

$$\text{Idle}(q) = (q \text{ at } 21).$$

The first aim is the following leads-to property:

Theorem 6. $(\text{Inside}(q) \wedge \text{sinc} = m) \text{ LT } \langle dr(q) \rangle (\text{Idle}(q) \wedge \text{sinc} \leq m + dn(q))$, where $dn(q) = (N - 1) \cdot \text{depth}(q) + N + 2$ and $dr(q) = A_2 \cdot dn(q) + C_2$.

Just as in Section 6.4, this result implies bounded overtaking. A similar result is contained in lemma 8 in the work of Attiya et al.⁶

Proof. The aim is to prove that the system proceeds within $dr(q)$ rounds from the precondition $\text{preO}(q, m)$ to some postcondition $\text{goal}(q, k)$:

$$\text{preO}(q, m) = (\text{Inside}(q) \wedge \text{sinc} = m),$$

$$\text{goal}(q, k) = (\text{Idle}(q) \wedge \text{sinc} \leq k).$$

The condition $Inside(q)$ is partitioned into three parts:

$$\begin{aligned} QorF(q) &= (Inside(q) \wedge (q \in qset \vee q = first)), \\ Run(q) &= (Inside(q) \wedge \neg QorF(q) \wedge friend[q] < N), \\ Margin(q) &= (Inside(q) \wedge \neg QorF(q) \wedge friend[q] = N). \end{aligned}$$

For the first two cases, we need to know how a thread traverses the queue. For this purpose, function $qval$ is defined by

$$\begin{aligned} qval(r) &= 0 \text{ if } r \notin qset, \\ qval(r) &= i + 1 \text{ if } queue[i] = r. \end{aligned}$$

It follows from $Kq1$ that $0 \leq qval(r) \leq N - 1$.

While thread r is in the queue, $qval(r)$ never grows, and it decreases whenever some thread increments $sinc$ and hence executes step 46. This implies that

$$\begin{aligned} pre3(q, m) &\text{ unless } goal(q, m + 1), \text{ where} \\ pre3(q, m) &= (QorF(q) \wedge sinc + qval(q) \leq m). \end{aligned} \quad (11)$$

Now, assume that $Run(q)$ holds. $Run(q)$ is only invalidated when $QorF(q)$ becomes true. The invariant $Mq0$ implies $q \in apply$. Therefore, $Nq2$ implies that $friend[q]$ is a thread that differs from q . It follows that $meet(friend[q], q) < depth(q)$. Therefore, $1 \leq level(q) \leq depth(q)$. For q with $Run(q)$, define the function

$$dist(q) = qval(friend[q]) + (N - 1) \cdot (depth(q) - level(q)).$$

It holds that $0 \leq dist(q) \leq (N - 1) \cdot depth(q)$.

When step 46 is executed, $level(q)$ and $friend[q]$ do not change, and $qval(friend[q])$ decreases with 1, because $friend[q]$ is never at 46 (by $Nq3$ and $Kq8$). When $friend[q]$ changes, $level(q)$ increases and $qval(friend[q])$ can increase at most with $N - 1$. Therefore, $dist(q)$ never increases, and it decreases with 1 whenever step 46 is executed. In this way, it is proved that

$$\begin{aligned} pre2(q, m) &\text{ unless } pre3(q, m + N), \text{ where} \\ pre2(q, m) &= (Run(q) \wedge sinc + dist(q) \leq m). \end{aligned} \quad (12)$$

It follows from (11) and (12) that

$$\begin{aligned} pre23(q, m) &\text{ unless } goal(q, m + N + 1), \text{ where} \\ pre23(q, m) &= (pre2(q, m) \vee pre3(q, m + N)). \end{aligned} \quad (13)$$

Assume that $Margin(q)$ holds. $Margin(q)$ can be invalidated in two ways. Either q becomes $first$ in step 38, or q gets a friend in one of the steps 22, 41, or 45. $Margin(q)$ implies $level(q) = 0$. As above, $Mq0$ implies $q \in apply$. Thread q is in 23–40 because of $lq0$. Therefore $j.q = 1$ by $Kq7$. Therefore, the first execution of $modfriend$ gives q a friend. The only unconditional call of $modfriend$ is in step 41. Step 41 is done each time a thread traverses the CS, but when $Margin(q)$ is made true, the thread $first$ may have passed line 41, in which case q has to wait for the next traversal:

$$pre1(q, m) = (Margin(q) \wedge sinc \leq m + 1 \wedge (sinc \leq m \vee first = N \vee pc.first \leq 41)).$$

In view of the upper bounds of $dist$ and $qval$, it holds that

$$pre1(q, m) \text{ unless } pre2(q, m + (N - 1) \cdot depth(q) + 1) \vee pre3(q, m + N + 1).$$

As **unless** allows weakening of the postcondition, this implies

$$pre1(q, m) \text{ unless } pre23(q, m + (N - 1) \cdot depth(q) + 1). \quad (14)$$

The formulas (13) and (14) together imply

$$\begin{aligned} pre123(q, m) &\text{ unless } goal(q, m + dn(q)), \text{ where} \\ pre123(q, m) &= (pre1(q, m) \vee pre23(q, m + (N - 1) \cdot depth(q) + 1)). \end{aligned} \quad (15)$$

because $dn(q) = (N - 1) \cdot depth(q) + N + 2$.

Theorem 5 with $k \leftarrow m$ and $j \leftarrow dn(q)$ gives

$$(m \leq sinc) \text{ } \mathbf{LT}\langle dr(q) \rangle \quad (m + dn(q) \leq sinc \vee Al). \quad (16)$$

The PSP-rule is applied to combine the Formulas (15) and (16). After this, the precondition $pre0(q, m)$ is replaced by noting that $pre0(q, m)$ is a subset of the intersection of $pre123(q, m)$ with $(m \leq sinc)$. By setting $k = m + dn(q)$, the postcondition can be replaced by $goal(q, k)$, because $pre123(q, m) \wedge (k \leq sinc \vee Al)$ is a subset of $goal(q, k)$. $\square$

By the disjunction rule, Theorem 6 easily implies

$$Inside(q) \text{ } \mathbf{LT}\langle dr(q) \rangle \quad Idle(q).$$

so it straightforward to prove that the Doorway is passed in $depth(q) + 1$ rounds:

$$Doorway(q) \text{ } \mathbf{LT}\langle depth(q) + 1 \rangle \quad Inside(q).$$

By transitivity,

Theorem 7. $\text{true } \mathcal{L}\mathcal{T}(dt(q)) \text{Idle}(q)$, where $dt(q) = dr(q) + \text{depth}(q) + 1$.

Note that both $dr(q)$ and $dt(q)$ are equal to $3 \cdot N \cdot \text{depth}(q)^2$ plus some lower order terms. In particular, if the tree is balanced, the individual delay is $\mathcal{O}(N(\log N)^2)$.

9 | VERIFICATION OF THE TREE $\mathcal{L}\mathcal{F}$ FLAG ELEVATOR

The verification of the Tree $\mathcal{L}\mathcal{F}$ Flag elevator has the same structure as the one of the Tree $\mathcal{C}\mathcal{A}\mathcal{S}$ Flag elevator in Sections 7 and 8. The transition system is built in Section 9.1. Section 9.2 proves MX . The most difficult part is the proof of absence of deadlock states in Section 9.3. The proof of progress, both throughput and individual progress, is given in Section 9.4.

9.1 | The transition system of the Tree $\mathcal{L}\mathcal{F}$ Flag elevator

The transition system for the Tree $\mathcal{L}\mathcal{F}$ Flag elevator is obtained from Figure 8 by replacing lines 23–40 by Figure 9. Lines 23–35 of this figure represent the function `trylock $\mathcal{L}\mathcal{F}$` of Figure 2B. Lines 36–40 are analogous to lines 36–40 of Figure 8. In lines 28–30, a private set `li` is used to hold the threads for which the loop body, line 30, has yet to be executed. On the one hand, this allows the implementer of the algorithm to choose any order for the loop. On the other hand, it is convenient for the proof, see the invariant $Hq4$, and the function lvf in Section 9.4.

For convenience, the private variable `leader` is made persistent, and initially false. This makes it easier to mention `leader` in the invariant $lq7$ of the algorithm. The shared history variable `mu` has the same role as for the Tree $\mathcal{C}\mathcal{A}\mathcal{S}$ elevator.

9.2 | Mutual exclusion

In the proof of MX , the invariants $lq0, \dots, lq5$ are retained. The invariant $lq2$, however, is now also threatened by steps 33 and 36. It is preserved because of $lq3$ and the new invariants

$$\begin{aligned} lq6 : & \quad q \text{ at } 33 \Rightarrow \neg \text{fast}, \\ lq7 : & \quad \text{leader}.q \Rightarrow q \text{ in } \{34 \dots 39\} \wedge \text{fast} \wedge \text{mu} = q. \end{aligned}$$

```

var x : 0 .. N - 1 ;
var y : 0 ... N ← N ;
var b : array[0 .. N - 1] of bool ← [N] : false ;

thread(p : 0 ... N - 1) =
  var li, pc, i : ℕ ;
  var leader : bool ← false
23  b[p] ← true ;
24  x ← p ;
25  if y ≠ N then goto 35 endif ;
26  y ← p ;
27  if x ≠ p then
28    b[p] ← false ; li ← thread \ {p} ;
29    while exists i ∈ li do
30      await ¬b[i] ; remove i from li ;
    endwhile ;
31    if y ≠ p then goto 37 endif ;
  endif ;
32  if ¬fast then
33    fast ← leader ← true ; mu ← p endif ;
34  y ← N ;
35  b[p] ← false ;
36  if leader then
37    await flag[N] ∨ flag[p] ;
38    flag[N] ← false ; first ← p ;
39    fast ← leader ← false ;
  else
40    await flag[p] ;
  endif .

```

FIGURE 9 The Tree $\mathcal{L}\mathcal{F}$ flag elevator: replacement for lines 23–40 in Figure 8

Predicate $Iq6$ is threatened only by step 33. It is preserved because this variation of Lamport's fast algorithm establishes MX for the region 32–34:

$$Hq0 : q \text{ in } \{32 \dots 34\} \wedge r \text{ in } \{32 \dots 34\} \Rightarrow q = r.$$

Predicate $Iq7$ is threatened only by the steps 33 and 39. It is preserved because of $Iq6$ and $Iq2$, respectively.

Predicate $Hq0$ is threatened only by the steps 27 and 31. It is preserved because of the new invariants

$$Hq1 : q \text{ in } \{26 \dots 27\} \wedge r \text{ in } \{32 \dots 34\} \Rightarrow q \neq x,$$

$$Hq2 : q \text{ at } 31 \wedge r \text{ in } \{26 \dots 27\} \cup \{32 \dots 34\} \Rightarrow q \neq y.$$

Predicate $Hq1$ is threatened only by the steps 25 and 31. It is preserved by step 31 because of $Hq2$. It is preserved by step 25 because of the new invariant

$$Hq3 : q \text{ in } \{32 \dots 34\} \Rightarrow y \neq N.$$

Predicate $Hq2$ is threatened only by step 29E. It is preserved because of

$$Hq4 : q \text{ in } \{26 \dots 27\} \cup \{32 \dots 34\} \wedge y \neq N \wedge y \text{ in } \{29, 30\} \Rightarrow q \in \text{li.y}.$$

Predicate $Hq3$ is threatened only by the steps 27 and 34. It is preserved by step 34 because of $Hq0$. It is preserved by step 27 because of

$$Hq5 : x \text{ at } 27 \Rightarrow y \neq N.$$

Predicate $Hq4$ is threatened only by step 30. It is preserved because of

$$Hq6 : q \text{ in } \{24 \dots 28\} \Rightarrow b[q],$$

$$Hq7 : q \text{ in } \{32 \dots 34\} \wedge y \neq N \wedge y \text{ in } \{27 \dots 30\} \Rightarrow b[q].$$

Predicate $Hq5$ is threatened only by step 34. It is preserved because of $Hq1$. Predicate $Hq6$ is inductive. Predicate $Hq7$ is threatened only by the steps 26 and 27. It is preserved by step 27 because of $Hq6$. It is preserved by step 26 because of

$$Hq8 : q \text{ in } \{32 \dots 34\} \wedge r \text{ at } 26 \Rightarrow b[q].$$

Predicate $Hq8$ is threatened only by steps 25, 27, and 31. It is preserved because of $Hq3$, $Hq6$, and $Hq2$, respectively.

9.3 | Absence of deadlock states

Predicate $Jq0b$ is now replaced by the inductive predicate

$$Jq0c : q \text{ in } \{21 \dots 46\}.$$

The predicates $Jq2a$, $Jq3b$, $Jq8$, $Jq9$, and $Kq0, \dots, Kq8$ are retained.

Predicate $Jq1$ is replaced by the inductive predicate

$$Jq1c : \text{fast} \Rightarrow \mu \text{ in } \{34 \dots 39\} \wedge \text{leader}.\mu.$$

Two other invariants are needed, both of which are inductive:

$$Jq6c : b[q] \Rightarrow q \text{ in } \{24 \dots 28\} \cup \{32 \dots 34\},$$

$$Jq7c : y \neq N \Rightarrow y \text{ in } \{27 \dots 34\}.$$

Predicate $Jq5$, however, needs to be weakened to

$$Jq5c : q \text{ at } 40 \wedge \text{first} = N \wedge y = N \Rightarrow \text{fast}$$

because it is possible that y holds some thread that is hidden at line 34. In that case, other threads can move from line 25 and 31 to 35 and 36.

Enabledness of thread q is expressed by

$$\begin{aligned} \text{ena}(q) = & (q \text{ not-at } 21) \\ & \wedge (q \text{ at } 30 \Rightarrow \neg b[q]) \\ & \wedge (q \text{ at } 37 \Rightarrow \text{flag}[q] \vee \text{flag}[N]) \\ & \wedge (q \text{ at } 40 \Rightarrow \text{flag}[q]). \end{aligned}$$

Under the assumption of $Jq5c$, absence of deadlock states is proved as follows. Assume that no threads are enabled. By $Jq0c$ and predicate ena , all threads are at the locations 21, 30, 37, or 40. By $Jq6c$, it follows that $b[q]$ is false for all threads q . Therefore, all threads at line 30 are enabled. This result implies there are no threads at 30, and all threads are at 21, 37, or 40. If $\text{first} \neq N$, then thread first is enabled because of $Jq2a$ and $Jq8$. If $y \neq N$, then thread y is enabled because of $Jq7c$. Therefore, $\text{first} = y = N$. If fast holds, then thread μ is enabled because of $Jq1$ and $Jq9$. Therefore, $Jq5c$ implies there are no threads at 40. Any thread at 37 is enabled because of $Jq9$. This proves that all threads are at line 21, ie, idle.

It therefore remains to prove $Jq5c$. Predicate $Jq5c$ is implied by predicate $Kq1$ together with the new invariant

$$Lq0c : q \text{ in } \{27 \dots 40\} \wedge \text{queue} = \varepsilon \wedge \neg \text{fast} \wedge y = N \Rightarrow \text{first} \neq N \wedge \text{first} \text{ in } \{22 \dots 45\}.$$

Predicate $Lq0c$ is threatened only by the steps 34, 39, 44, and 46. It is preserved by step 39 because of $lq0$, and by step 46 because of $lq0$, and $Kq0$. It is preserved by step 34 because of

$$Lq1c : q \text{ at } 34 \wedge \text{queue} = \varepsilon \wedge \neg \text{fast} \Rightarrow \text{first} \neq N \wedge \text{first} \text{ in } \{22 \dots 45\}.$$

It is preserved by step 44 because of

$$Lq2cA : q \text{ in } \{27 \dots 40\} \wedge r \text{ at } 44 \wedge \text{queue} = \varepsilon \wedge \neg \text{fast} \wedge y = N \Rightarrow j.r \leq \text{depth}(r).$$

Predicate $Lq1c$ is threatened only by the steps 39, 44, and 46. It is preserved by 39 and 46 because of $lq0$ and $Kq0$. It is preserved by step 44 because of

$$Lq3cA : q \text{ at } 34 \wedge r \text{ at } 44 \wedge \text{queue} = \varepsilon \wedge \neg \text{fast} \Rightarrow j.r \leq \text{depth}(r).$$

In order to prove $Lq2cA$ and $Lq3cA$, a new shared history variable `cand` (for candidate) is introduced, which gets a new value when a thread executes line 42. In fact, command 42 is extended in a nondeterministic way to

```

42   flag[p] ← false;
      if exists q in {27 ... 40} then cand ← q
      else cand ← N endif;

```

At line 42, `cand` gets the number of some thread that approaches a waiting location, if possible. Otherwise, `cand` becomes N (not a thread). There are two invariants that imply that `cand` holds a thread, and two invariants that apply when `cand` holds a thread:

$$\begin{aligned}
Lq2c : & q \text{ in } \{27 \dots 40\} \wedge r \text{ in } \{43 \dots 46\} \wedge \neg \text{fast} \wedge y = N \Rightarrow \text{cand} < N, \\
Lq3c : & q \text{ at } 34 \wedge r \text{ in } \{43 \dots 46\} \wedge \neg \text{fast} \Rightarrow \text{cand} < N, \\
Lq4c : & r \text{ in } \{43 \dots 46\} \wedge \text{cand} < N \Rightarrow \text{cand} \text{ in } \{27 \dots 40\}, \\
Lq5c : & r \text{ in } \{44, 45\} \wedge \text{cand} < N \wedge \text{queue} = \varepsilon \wedge \neg \text{fast} \Rightarrow j.r \leq \text{meet}(r, \text{cand}) + 1.
\end{aligned}$$

The predicate $Lq2cA$ is implied by $Lq2c$, $Lq4c$, and $Lq5c$. Similarly, predicate $Lq3cA$ is implied by $Lq3c$, $Lq4c$, and $Lq5c$.

Predicate $Lq2c$ is threatened only by the steps 34, 39, and 42. It is preserved by step 34 because of $Lq3c$, by step 39 because of $lq0$, by step 42 because of $lq0$ and $lq1$.

Predicate $Lq3c$ is threatened only by steps 39 and 42. In either case, it is preserved because of $lq0$.

Predicate $Lq4c$ is threatened only by the steps 37, 39, and 40. It is preserved by steps 37 and 40 because of $lq0$ and $lq4$, and by step 39 because of $lq0$.

Predicate $Lq5c$ is threatened only by the steps 39, 42, 43, 45, and 46. It is preserved by steps 39, 42, and 46 because of $lq0$, by step 43 because of $Kq7$, and by step 45 because of the new invariant

$$Lq6c : r \text{ at } 45 \wedge \text{cand} < N \wedge \text{queue} = \varepsilon \wedge \neg \text{fast} \wedge j.r = \text{meet}(r, \text{cand}) + 1 \Rightarrow \text{apply}[k.r].$$

Predicate $Lq6c$ is threatened only by the steps 39, 41, 42, 44, and 46. It is preserved by steps 39, 41, 42, and 46 because of $lq0$. It is preserved by step 44 because of $Lq4c$ and the postulate $SibAp$.

It turns out that the invariants Mq^* and Nq^* of Section 7.3 can be retained without modification. Therefore, $SibAp$ follows from Lemma 1 as before.

This concludes the proof of absence of deadlock states for the Tree LF Flag elevator.

9.4 | Progress for the Tree LF Flag elevator

Having done the above verifications with a mechanical theorem prover PVS, the transition system for the Tree LF Flag elevator was submitted to the PVS-proof of progress of the Tree CAS Flag elevator. PVS immediately signaled that the function lvf needed modification. It was therefore changed to

$$\begin{aligned}
lvf(q) = & pc.q - 21 + (pc.q = 22 \ ? \ \text{depth}(q) + 1 - j.q : 0) + (pc.q \geq 23 \ ? \ \text{depth}(q) : 0) \\
& + (pc.q \geq 29 \ ? \ 2 \cdot N - 2 \cdot \#1.i.q : 0) + (44 \leq pc.q \leq 45 \ ? \ 2 \cdot j.q - 2 : 0) + (pc.q = 46 \ ? \ 2 \cdot \text{depth}(q) - 1 : 0).
\end{aligned}$$

Using $Jq0b$, $Kq5$, and $Kq6$, it is now possible to verify that

$$0 \leq lvf(q) < A_3 \text{ where } A_3 = 2 \cdot N + 3 \cdot D + 25 \text{ and } D = \max_q \text{depth}(q).$$

After this modification, and with A_2 replaced by A_3 , all arguments and results of Section 8 remain valid. For instance, the new version of $lvf(q)$ increases with every step of q , which was easily verified with PVS. The remainder of the PVS verification of progress was identical, which results in the Throughput Theorem:

Theorem 8. $(k \leq \text{sinc}) \ \text{LT}(A_3 \cdot j + C_3)((k + j \leq \text{sinc}) \vee A1)$ where $A_3 = 2 \cdot N + 3 \cdot D + 25$ and $C_3 = A_3 \cdot N - A_3 + 1$.

The throughput factor A_3 is linear in N because of the loop at line 30. Therefore, C_3 is quadratic in N . The result for individual progress is

Theorem 9. *true* $\text{LT}(F_3)\text{Idle}(q)$, where $F_3 = A_3 \cdot ((N - 1) \cdot D + 2N + 1) + D + 2$.

In particular, the individual delay F_3 is of order $\mathcal{O}(N^2 \log N)$.

10 | IMPLEMENTATION CORRECTNESS[§]

Concurrent software-algorithm correctness only relies on the sequential-consistency (SC) memory-model, which allows reading stale information. By relying on equality/inequality testing for a unique value when threads (busy) wait, delays in propagating values do not affect the algorithm as the old value is correct and the new value is correct when it eventually arrives. In general, software algorithms are proved correct and tested under an SC assumption.

However, to aggressively optimize sequential programs, memory models additively allow: reads to be moved before disjoint (different variables) writes (total store order, TSO), writes to be moved before disjoint reads (partial store order, PSO), and writes to be moved before disjoint writes (weak order, WO). In general, all parallel architectures provide hardware directives, called *fences*, to disable one or more of these optimizations and claw-back SC. Fencing is independent of a software algorithm; therefore, all software algorithms that work under SC can be made to work on any modern architecture. Correctly inserting minimal fencing for each architecture is performed via an ad-hoc procedure done by experienced programmers, or automated by the compiler using atomic directives. However, the state-of-art automation often over-protects an algorithm resulting in slower execution.

Figures 2 and 6 show the necessary implementation additions for preventing incorrect optimizations by the compiler and hardware. The transition systems of Figures 7, 8, and 9 are formal models, and hence, do not need fences because there are no implementation optimizations, but fences are needed in their implementations in Section 11. The amount of fencing is related to the design and complexity of an algorithm and can result in reduced throughput in performance experiments. Only the additions in Figure 2B are discussed in detail.

All implementation work is done in C to allow direct access to the underlying hardware and avoiding any performance effects from managed runtimes. Furthermore, we believe that most embedded environments provide better support for C than Java. Hence, the **volatile** in the paper refers to C.

The **volatile** qualifiers in Figures 2 and 6 control static compiler transformations, such as movement, elision, and caching of values into registers. For example, at any of the **await** statements, eg, **await** $\text{first} = p \vee \text{first} = N$, if the compiler caches first into a register, changes to it in memory are not seen, resulting in livelock due to spinning. The **volatile** qualifier prevents the caching, forcing a variable to be loaded on each reference. Because the **volatile** qualifier is associated with a variable rather than its specific usages, the compiler may restrict some valid optimizations, resulting in reduced performance.

The fences in Figures 2 and 6 control dynamic hardware-transformations. The implementations are run on x86 and SPARC processors, both using a TSO memory model, allowing reads to be moved before writes for disjoint (different) variables,²³ which is a legitimate optimization for sequential programs. For concurrent programs, this optimization violates program order, increasing potential interference beyond what is considered in the verification discussed in Sections 5, 6, 7, and 8.

In general, fences are inserted after every write to a shared variable that is followed by a read from a shared variable. Java **volatile** and C/C++ **atomic** provide implicit fencing, which we do manually by inspection for each C program to prevent over-protection. (The same technique is used in the Linux kernel, for instance). In cases where we can reason about safety, some fences are removed to boost performance by allowing the hardware to perform its optimizations. For the purposes of verification, all fences are retained so there is only a single algorithm to verify in Sections 5, 6, 7, and 8, instead of multiple versions based on possible code movement. For the first two fences of Figure 2B, a scenario is presented to show that removal of the fence can lead to violation of MX. For the third fence, an explanation is given for why it can be removed and maintain safety.

The first fence prevents the read of y at line 26 from moving before the disjoint writes of $b[p]$ and x at lines 24-25, as in

```

23(a)      tempy ← y;                      // move read before disjoint writes
24         b[p] ← true;
25         x ← p;                          // race
26         if tempy ≠ N then
27           b[p] ← false; return false endif;
28         y ← p;                          // race
29         if x ≠ p then

```

This relaxation allows the scenario with initially $y = N$, where q_1, q_2 reach line 25 simultaneously. Now, q_1 executes lines 25-29 and enters the CS, and then q_2 executes lines 25-29 and also enters the CS, violating MX. If q_2 reads y at line 26, the value is unequal to N because of q_1 's assign at line 28, so q_2 executes line 27.

[§]This section is based on²⁴[Section 8.2], as the general discussion is similar.

The second fence prevents the read of x at line 29 from moving before the disjoint writes of $b[p]$ and y at lines 27-28, as in

```

25       $x \leftarrow p$ ;                // race
25(a)   temp $x \leftarrow x$ ;          // move read before disjoint writes
26      if  $y \neq N$  then
27         $b[p] \leftarrow \text{false}$ ; return false endif;
28       $y \leftarrow p$ ;                // race
29      if temp $x \neq p$  then
30         $b[p] \leftarrow \text{false}$ ;

```

This relaxation allows the scenario with initially $y = N$, where thread q_1 executes lines 24-26 and then thread q_2 does the same. Hence, both threads at line 26 have $\text{temp}x = p$. The threads continue and enter the CS, violating MX. If q_1 and q_2 read x at line 29, x is equal to either q_1 or q_2 ; thus, one thread executes line 30.

The third fence prevents a read of $b[\text{thr}]$ on line 31 from moving before the write of $b[p]$ on line 30 for $\text{thr} \neq p$:

```

30      b $[p] \leftarrow \text{false}$ ;
31      forall  $\text{thr} \in 0 \dots N - 1$  do await  $\neg b[\text{thr}]$  endfor;
32      if  $y \neq p$  then return false endif;

```

This fence is optional because postponing $b[p] \leftarrow \text{false}$ only keeps other threads waiting longer, but eventually, the store occurs. Hence, removing the third fence does not endanger the algorithm safety; however, modeling this movement is difficult in the theorem prover, so the proof uses the simpler approach of assuming the existence of the third fence.

11 | PERFORMANCE EXPERIMENT[†]

Two performance experiments, maximal and minimal contention, are used to compare the performance differences between the Elevator and the other relevant algorithms. The maximal-contention experiment tests the algorithms in the worst-case scenario, where all (but one) threads are spinning in the lock. The minimal-contention experiment tests the algorithms in the best-case scenario, where the lock has been provisioned for N threads, but only one thread is active, that is, the thread does not have to wait in the lock. These extremes give a general sense of how a locking algorithm performs. Often an algorithm is designed to optimize only one of these scenarios. Furthermore, while embedded systems usually have a small number of cores, we wanted to examine if the algorithms scaled, so tests are performed up to 32-cores. Finally, a standard atomic hardware-assisted algorithm is presented to contrast performance with software solutions based solely on atomic read/write. The code to reproduce the experiments is publicly available,²⁵ and the algorithm implementations are checked by the Relacy Race Detector.²⁶

11.1 | Experimental setup

The performance experiments have a test harness that creates T pthread worker-threads and uses the algorithms constructed for N threads, where $1 \leq T \leq N$ and $N \leq 32$ for N -thread solutions. The maximal contention experiment uses $T = N$; the minimal contention experiment uses $T = 1$. After thread creation, the harness blocks for a fixed period, t , and then sets a global stop flag to indicate the experiment is over. The T threads repeatedly attempt entry into the following self-checking *CriticalSection* until the stop flag is set:

```

inline void CriticalSection( const unsigned int  $id$  ) {
  static volatile unsigned int  $\text{CurrTid}$ ;    // current thread in CS
   $\text{CurrTid} = id$ ;                          // race
  Fence()# ;                             // optional
  for ( int  $i = 1$ ;  $i \leq 100$ ;  $i++$  )        // delay
    if (  $\text{CurrTid} \neq id$  ) abort();         // MX?
}

```

The shared variable CurrTid holds the id of the thread currently in the CS, and it is initialized at the beginning of *CriticalSection* to the calling thread's id. A thread then loops 100 times pretending to perform the CS, but at the end of each iteration, the thread compares its id with the one in CurrTid for any change. If there is a change, MX has been violated, and the program is stopped. (This check helped significantly during algorithm testing.)

During the t seconds of the experiment, each thread counts the number of times it enters the CS. The higher the aggregate count, the better an algorithm, as it is able to process more requests for the CS per unit time (throughput). When the stop flag is set, a worker thread stops entering the CS and atomically adds its subtotal entry-counter to a global total entry-counter. When the harness wakes after t seconds, it busy waits until all threads have noticed the stop flag and added their subtotal to the global counter, which is then stored. Five identical experiments are performed, each lasting 60 s. The median value of the five results is plotted.

[†]This section is based on²⁴[Section 8.1], as the same experimental setup is used.

[#] See the work of Buhr et al.^{3(p.657)} for an explanation of why this fence may be optional.

The minimal contention experiment measures the cost of fast access within an algorithm as N increases. To ensure the single thread exercises all aspects of an algorithm, it is assigned different start-points on each access to the CS by randomly changing its thread id. The randomness is accomplished using approximately 64 pseudo random thread ids, where 64 is divided by N to obtain R repetitions, for example, for $N = 5$, $R = 64/5 = 12$. Each of the 12 repetitions is filled with five random values in the range, $0..N-1$, without replacement, for example, 0 3 4 1 2. There are no consecutive thread ids within a repetition, but there may be between repetitions. The thread cycles through this array of ids during an experiment.

The performance experiments were run on three different multicore hardware systems to determine if there is consistency across platforms:

1. Supermicro AS-1042G-TF with four sockets, each containing an AMD Abu Dhabi 6380 16 core 2.5 GHz, is equal to 64 cores, running Linux v3.13.0-49, compiling with gcc 6.2.0;
2. Supermicro SYS-8017R-TF+ with four sockets, each containing an Intel Xeon 8 core 2.6 GHz E5-4620V2, is equal to 32 cores, turbo-boost off, running Linux v3.13.0-49, compiling with gcc 6.2.0; and
3. Oracle single-socket SPARC T2+ with eight cores, each core has two pipelines, and each pipeline supports (multiplexes) four logical processors, running Solaris 10, compiling with gcc 5.2.

Only two sockets are used on the AMD, and all four on the Intel for the 32 core experiments; hence, NUMA effects with respect to accessing shared data occur. The SPARC T2+ is a single-node UMA-architecture.

The major architecture difference between the two x86 (Intel/AMD) servers are as follows: interconnect, QPI versus HyperTransport, and cache-coherence protocol (MESI/MESIF versus MOESI). As well, Intel processors have hyper-threading, but it is not used.

The major difference between the x86 and SPARC architectures is communication cost, because of the single versus multi-socket architectures. This difference in communication cost has a significant effect on software solutions for MX because the algorithms communicate heavily via shared memory. Therefore, when communication cost is high, placement of threads is crucial to reduce the cost of shared access, that is, pack cores then sockets. However, the default scheduling policies for both Linux and Solaris assume low sharing among threads (which is normally the correct assumption) and hence, scatter threads across sockets and cores to prevent inter-core resource competition, such as increased cache pressure, which slows execution. Because the single-socket SPARC has very low communication costs, thread placement is largely irrelevant; hence, Solaris placement policy does not affect testing the software algorithms on the SPARC T2+ (but would on other SPARC processors). Unfortunately, given the high communication costs on the x86 machines, the thread placement policy of Linux is the worst possible approach to demonstrate differences among the software algorithms and results in unusual behavior and significant jitter. Therefore, for the x86 experiments, threads are explicitly pinned (using affinity) to cores and then sockets, which clears up irregularities in the data and more closely represents single-socket processors on embedded systems, where these algorithm might be used.

Twelve elevator algorithms are tested, linear and tree, each with 6 variants CAS, CAS flag, BL, BL flag, LF, LF flag. For comparison, two other atomic hardware-assisted algorithms are included to give a sense of overall performance. One algorithm is Mellor-Crummey and Scott (MCS),¹⁵ which augments a software solution with an atomic fetch-and-store and CAS instruction, and the RMRs algorithm, with two CAS instructions. All tested algorithms are starvation free, while only MCS is fast (constant time for one thread). Faster hardware-assisted locks exist but most have starvation. MCS and its variants are used in the Linux kernel to obtain fast/fair locking.²⁷

11.2 | Experimental results

Figure 10A-C shows the entry counts from the maximal contention experiment for each algorithm on the AMD x86, Intel x86 and SUN SPARC T2+ architectures, respectively. (Throughput graphs where higher is better.) The graph for x86 uses a log scale because the results range by an order of magnitude, which compresses the results, making it difficult to see differences among them.

Figure 11A-C shows the entry counts from the minimal contention experiment, that is, an access with no contention, one thread, but $N = 1..32$. Since many applications have locks with low contention, getting through a non-contented access as quickly as possible is important. For $T = 1$, all fast algorithms have a finite number of memory accesses independent of N , and hence generate flat performance as N varies. The non-fast-algorithms experience some drop as N increases because of increased data structure size resulting in more scanning. As N increases, more steps are required by most of the algorithm even when only one thread is accessing the CS. For example, MCS is $O(1)$ with six accesses (three reads, three writes), but searching algorithms (linear/tree and BL/LF) take $O(N)$ and $O(\log N)$, resulting in linear or stepped (powers of 2) decreases.

11.3 | Experimental evaluation

Figure 10 is the maximal ($T = N$) contention results for the 12 algorithms on the 3 architectures. There is little or no effect at the 16-core NUMA boundary for the AMD experiments; there is an effect at the 8-core NUMA boundary for the Intel experiments. Nothing special is performed in the algorithms to be NUMA aware.

AMD x86

The performance results stratify into three groups. First are ElevatorLinearCASFlag, ElevatorLinearBLFlag, and ElevatorLinearLFFlag. Second are MCS, ElevatorTreeCASFlag, ElevatorTreeBLFlag, ElevatorTreeLFFlag. Third is ElevatorLinearCAS, ElevatorLinearBL, ElevatorLinearLF, ElevatorTreeCAS, ElevatorTreeBL, ElevatorTreeLF, and RMRs. Flag is faster

FIGURE 10 Critical-Section Entry-Counts, Maximal Contention: $T = N = 1..32$, 60 second, algorithm performance, higher value is better. A, AMD x86: Threads-CPU; B, Intel x86: Threads-CPU; C, SUN SPARC T2+: Threads-CPU

than non-flag, and within Flag, Linear is faster than Tree. The use of one or more atomic instructions does not provide a benefit.

Intel x86

The performance results stratify into three groups. First are ElevatorLinearCASFlag, ElevatorLinearBLFlag, and ElevatorLinearLFFlag. Second are MCS, ElevatorTreeCASFlag, ElevatorTreeBLFlag, ElevatorTreeLFFlag. Third is ElevatorLinearCAS, ElevatorLinearBL, ElevatorLinearLF, ElevatorTreeCAS, ElevatorTreeBL, ElevatorTreeLF, and RMRS. Flag is faster than non-flag, and within Flag, Linear is faster than Tree. The use of one or more atomic instructions does not provide a benefit.

SUN SPARC T2+

The performance results stratify into three groups. First are MCS, ElevatorLinearCASFlag, ElevatorLinearBLFlag, ElevatorLinearLFFlag, ElevatorLinearCAS, ElevatorLinearBL, and ElevatorLinearLF. Second are ElevatorTreeCASFlag, ElevatorTreeBLFlag, ElevatorTreeLFFlag, ElevatorTreeCAS, ElevatorTreeBL, and ElevatorTreeLF. Third is RMRS. Flag and Linear is faster than non-flag and Tree. The use of one or more atomic instructions does not provide a benefit.

BL and LF results are the same because both have a similar execution sequence at maximal contention, lines 24-26 and 25-27, respectively; thus, each returns *false* almost all the time. For the same reason, CAS and simulated CAS results are similar because of the extremely short search for simulated CAS. Flag is better than non-flag because it removes remote memory references that result in cache bouncing of the shared variable *first*. Linear is better than tree because the linear search always finds the next array element wanting for entry at maximal contention $O(1)$, while the tree approach must walk the tree $O(\log N)$.

Figure 11 is the minimal ($T = 1, N = 1..32$) contention results for the 12 algorithms on the 3 architectures. The graphs all have the following pattern: MCS is first and flat because its fastpath is seven accesses regardless of N . None of the other algorithms are FAST, as there are linear/tree searches in the entry protocol, which increase as N increases; hence, these algorithms all have a smooth/stepped linear decrease. In general, CAS

FIGURE 11 Critical-Section Entry-Counts, Minimal Contention: $T = 1$, $N = 1..32$, 60 second, higher value is better. A, AMD x86: N; B, Intel x86: N; C, SUN SPARC T2+: N

is better than simulated CAS, tree is better than linear (shorter search, $\log N$ versus N), LF is better than BL (fixed versus variable number of memory accesses) Flag has no affect because there is only one thread accessing the shared variable `first`. Essentially, the shorter the path to the CS, the better the performance. There are anomalies: ElevatorTreeCASFLAG is lower than expected on the AMD, and ElevatorTreeCAS is lower than expected on the SPARC.

12 | CONCLUSIONS

The tree CAS flag elevator algorithm performs better than the RMRs algorithm of Attiya et al⁶ (see Figure 3) from which it is derived. The main differences between the two algorithms are

1. In the exit-loop of RMRs (lines 26-33), the thread inspects three times as many `val[node]` values as the elevator. Also, the code of the exit-loop is more complicated, with its two loops.
2. RMRs has two waiting loops (lines 18 and 20) and the first one waits on the disjunction of three tests.
3. Two CAS instructions have more overhead than one.
4. In RMRs, the variable `first` is subject to two CAS instructions, some read instructions, and one write instruction.

Further experiments show that, for maximal contention, point 1 hardly counts for the AMD and Intel but makes a significant difference for the SPARC. For minimal contention, point 1 makes a difference on all three architectures. In this case, lines 17-20 of RMRs are never executed, which may explain why RMRs performs better than the elevator for minimal contention on AMD.

All flag elevators perform better than the corresponding non-flag elevators. For maximal contention, the linear elevators perform better than the tree elevators. For minimal contention, however, the tree elevators are to be preferred. Roughly speaking, the LF elevators perform equally well as the CAS elevators; both perform better than the BL elevators for larger N , say $N > 10$. The BL elevators perform surprisingly well for small N .

The verification of the tree elevators (CAS and LF) was extremely complicated until we found the simplifications of introducing the history variables `friend` in Section 7.3 and `cand` of Section 9.3.

12.1 | Complexity measures

For any MX algorithm, let A be the throughput factor and F be the individual delay. The linear CAS elevator has $A = \mathcal{O}(N)$ and $F = \mathcal{O}(N^2)$ by Theorems 2 and 4. The tree CAS elevator has $A = \mathcal{O}(\log N)$ and $F = \mathcal{O}(N \cdot (\log N)^2)$ by Theorems 5 and 7. The tree LF elevator has $A = \mathcal{O}(N)$ and $F = \mathcal{O}(N^2 \cdot \log N)$ by Theorems 8 and 9.

The different throughput factors for the CAS elevator and the LF elevator are a matter of worst-case performance. They are due to the loop at lines 29-30 of Figure 2B. Experiments show that most threads avoid this loop, either by going from 27 to 32, or by leaving from 25. Consequently, the average performance of LF elevator is comparable to the CAS elevator.

The order of magnitude of the constants A and F is unknown for most other MX algorithms. In the work of Hesselink,¹⁸ it is shown that the algorithm of Lycklama-Hadzilacos-Aravind has $A = \mathcal{O}(N^2)$ and $F = \mathcal{O}(N^3)$. It is proved in the work of Hesselink¹⁹ that the tournament algorithms considered in our other work,³ if based on Peterson's binary MX algorithm, reach $A = \mathcal{O}(\log N)$ and $F = \mathcal{O}(N)$.

For every algorithm, the individual delay F satisfies $F \geq N$. Assume an execution fragment with any kind of scheduling starting in a state with all threads in their entry protocols. Whenever a thread enters the CS, a round begins in which no other thread enters the CS. Therefore, it takes at least N rounds before all threads have entered the CS.

ACKNOWLEDGMENT

Peter Buhr was funded by the Natural Sciences and Engineering Research Council of Canada.

ORCID

Peter A. Buhr <http://orcid.org/0000-0003-3747-9281>

REFERENCES

1. Dijkstra EW. Solution of a problem in concurrent programming control. *Communications of the ACM*. 1965;8:569.
2. Dijkstra EW. Co-operating sequential processes. In: Genuys F, ed. *Programming Languages*. New York, NY: Academic Press; 1968:43-112. EWD123. <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html>; 1965.
3. Buhr PA, Dice D, Hesselink WH. High-performance N -thread software solutions for mutual exclusion. *Concurr Computat Pract Exp*. 2015;27(3):651-701.
4. Anderson TE. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans Parallel Distrib Syst*. 1990;1(1):6-16.
5. Aboulenein NM, Goodman JR, Gjessing S, Woest PJ. Hardware support for synchronization in the Scalable Coherent Interface (SCI). Paper presented at: Proceedings of 8th International Parallel Processing Symposium. 1994; New York, NY.
6. Attiya H, Hendler D, Levy S. An $\mathcal{O}(1)$ -barriers optimal RMRs mutual exclusion algorithm. Paper presented at: PODC'13: Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing. 2013; New York, NY.
7. Lamport L. Fast mutual exclusion algorithm. *ACM Trans Comp Syst*. 1987;5(1):1-11.
8. ARM. Cortex-M3 Technical Reference Manual, ARM DDI 0337E. 2006. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf
9. Microchip. 16-Bit MCU and DSC programmer's reference manual, DS70157f. 2017. <http://ww1.microchip.com/downloads/en/DeviceDoc/70157F.pdf>
10. Andrews GR. A method for solving synronization problems. *Sci Comput Prog*. 1989;13(4):1-21.
11. Hesselink WH, Buhr PA, Dice D. Fast mutual exclusion by the Triangle algorithm. *Concurr Computat Pract Exp*. 2018;30(4).
12. Burns JE, Lynch NA. *Mutual exclusion using indivisible reads and writes*. Paper presented at: Proceedings Eighteenth Annual Allerton Conference on Communication, Control, and Computing. 1980; Urbana, IL.
13. Lamport L. The mutual exclusion problem - Part II: statement and solutions. *Journal of the ACM*. 1986;33(2):327-348.
14. Lycklama EA, Hadzilacos V. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Trans Prog Lang Syst*. 1991;13(4):558-576.
15. Mellor-Crummey JM, Scott ML. Algorithm for scalable synchronization on shared-memory multiprocessors. *ACM Trans Comp Syst*. 1991;9(1):21-65.
16. Hesselink WH. PVS Proof scripts for elevator algorithms for mx. 2017. <http://wimhesselink.nl/mechver/elevators>
17. Hesselink WH. Progress under bounded fairness. *Distr Comput*. 1998;12:197-207.
18. Hesselink WH. Mutual exclusion by four shared bits with not more than quadratic complexity. *Sci Comput Program*. 2015;102(0):57-75. <http://www.sciencedirect.com/science/article/pii/S016764231500009X>
19. Hesselink WH. Tournaments for mutual exclusion: verification and concurrent complexity. *Form Asp Comput*. 2017;29(5):833-852.

20. Chandy KM, Misra J. *Parallel Program Design: A Foundation*. Boston, MA: Addison-Wesley; 1988.
21. Misra J. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. New York, NY: Springer-Verlag New York; 2001.
22. Lamport L. The mutual exclusion problem - Part I: a theory of interprocess communication. *Journal of the ACM*. 1986;33(2):313-348.
23. Adve SV, Gharachorloo K. Shared memory consistency models: a tutorial. *Computer*. 1996;29(12):66-76.
24. Buhr PA, Dice D, Hesselink WH. Dekker's mutual exclusion algorithm made RW-safe. *Concurr Computat Prac Exp*. 2016;28(1):144-165.
25. Buhr PA, Dice D, Hesselink WH. Concurrent-locking. 2017. <https://github.com/pabuhr/concurrent-locking>
26. Vyukov D. Relacy race detector. 2014. <http://www.1024cores.net/home/relacy-race-detector>
27. Linux 3.15. Release notice. 2014. http://kernelnewbies.org/Linux_3.15

How to cite this article: Buhr PA, Dice D, Hesselink WH. High-contention mutual exclusion by elevator algorithms. *Concurrency Computat Pract Exper*. 2018;30:e4475. <https://doi.org/10.1002/cpe.4475>