University of Groningen

Fast mutual exclusion by the Triangle algorithm

Hesselink, Willem; Buhr, Peter; Dice, David

Published in:
Concurrency and Computation

DOI:
[10.1002/cpe.4183](https://doi.org/10.1002/cpe.4183)

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):

Hesselink, W., Buhr, P., & Dice, D. (2018). Fast mutual exclusion by the Triangle algorithm. *Concurrency and Computation*, 30(4), [e4183]. <https://doi.org/10.1002/cpe.4183>

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

Fast mutual exclusion by the Triangle algorithm

Wim H. Hesselink¹ | Peter A. Buhr² | David Dice³¹Department of Computer Science, University of Groningen, 9700 AK Groningen, The Netherlands²Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada³Oracle Labs, Burlington, MA, USA

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

Summary

This paper presents a new *starvation-free* software algorithm for the N -thread mutual-exclusion problem. In the absence of contention, the algorithm requires only eight write and four read operations to enter and leave the critical section; to the best of our knowledge, this is optimal. For algorithms *with starvation*, five write and two read operations are optimal. In the presence of contention, the algorithm has excellent performance comparable to the best-known software solutions using only atomic load and store and to a hardware-assisted lock (MCS) using stronger atomic primitives and used within the Linux kernel. It is rare for software-only algorithms for mutual exclusion to perform well for both minimal and maximal contention workloads, making the new algorithm largely self-tuning when exposed to swings in access patterns.

KEYWORDS

concurrency, mutual exclusion, performance experiment, software solutions

1 | INTRODUCTION

The problem of mutual exclusion was proposed in 1965 by Dijkstra¹ 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). 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 lock-out). Dijkstra's solution¹ has starvation. The first starvation-free solution is due to Knuth.² In contrast to software solutions are *hardware-assisted* solutions, which have additional atomicity, eg, test-and-set,³ and pure *hardware* solutions⁴ requiring little or no software.

In 1987, Lamport⁵ observed that, in practice, a thread often needs access to the CS when there are no competing threads. Therefore, it is important to optimize this case for good performance. This notion is formalized as follows. We define a mutual-exclusion algorithm to be *fast*, if in the absence of competing threads, a thread executes the entry and exit protocol for the CS in a number of memory accesses that is finite and independent of N . Lamport presented a fast-algorithm^{5, Fig. 2} for which the number of memory accesses is 7 (5 writes and 2 reads) and proved that this number is optimal. The paper states that the solution allows starvation, but alludes to

In private correspondence, Gary Peterson has described a modified version of Algorithm 2 that is starvation free. However, it requires one additional memory reference in the absence of contention.^{5, p. 5}

We cannot locate this algorithm and do not know whether Peterson published this modified version.* Alur and Taubenfeld⁶ present a simpler fast-algorithm with 8 accesses (5 writes and 3 reads), but it requires a minimum time-delay for waiting tasks, which can be difficult to accurately implement for short durations.

Over the years, many *software* mutual-exclusion algorithms have been proposed that require no atomic instructions other than loads and stores (or RW-safeness with no atomic requirement^{7, p. 836; 8, p. 337; 9, Figure 4}). Recently, we performed an investigation¹⁰ of 20 of these software solutions. The algorithms were implemented, and their performance compared, under both 0 and high contention. Among these algorithms, only Lamport's algorithm⁵ has the fast property. It allows, however, starvation of processes and therefore relies on the assumption that contention is rare.

* We asked Leslie Lamport if he could find the correspondence, but he was unable to locate it.

FIGURE 1 Triangle algorithm

The aim of this paper is to eliminate this assumption by providing an algorithm that combines the fast property with starvation freedom and strong performance even in the case of contention.

In 1992, Styer¹¹ published a supposedly starvation-free fast MX algorithm. However, all our attempts to implement Styer's algorithm¹¹, Figure 1, p. 162 led to MX violations. In our opinion, the algorithm is incorrect (see Section 3.3). Yet inspired by Styer's design, we developed a new starvation-free fast MX algorithm, called the *Triangle algorithm* because it has the shape shown in Figure 1. Independent of our recent extension to Styer's algorithm, Yang and Anderson¹², Fig. 3, p. 57 (YA95) in 1995 developed an algorithm based on the same structure as Figure 1 (see Yang and Anderson¹², Fig. 4, p. 57). Moreover, in 2001, Anderson and Kim¹³, Fig. 4, p. 22 (AK01) proposed an improved algorithm. (Neither of these papers nor a follow-on survey paper on software solutions for mutual exclusion¹⁴ cites Styer's work.) These papers are clearly based on the idea in Styer's algorithm. Yet our Triangle algorithm is unique for three reasons. First, its simplicity, as it has fewer variables and less code. Second, it is closer to Lamport's fast-algorithm than YA95 and AK01, and it modularizes the constituent algorithms more strongly. Third, our performance experiments show that the Triangle algorithm outperforms the other two, both under maximal and minimal contention (ie, only a single competing thread).

The goal behind all three algorithms is to combine a fast N -thread algorithm, with excellent minimal contention properties, with a slow N -thread algorithm, with excellent maximal contention properties, to obtain the best properties for these two important workloads (minimal and maximal contention). (Here, *fast* and *slow* refer to the number of memory accesses during no contention not an algorithm's performance.) In Figure 1, F (fast) stands for our variation of Lamport's fast-algorithm, S (slow) stands for an arbitrary starvation-free MX algorithm for N threads (eg, Hehner-Shyamasundar, Aravind, Taubenfeld, Peterson-Buhr¹⁰), and B (binary) stands for a starvation-free MX algorithm for two threads (eg, Dekker, Peterson, Kessels, Tsay⁹).

The intuition is as follows. All threads enter at F . When threads have traversed F far enough to determine that there are competing threads, all but one are redirected to S to compete again in a general N -thread algorithm. Finally, the winners of the F and S algorithms compete at B for access to the CS. Hence, the solution is parameterized by the algorithms F , S , and B . Good performance results are obtained by choosing the best solutions for S and B encountered in our earlier investigation.¹⁰

All three algorithms should be slightly slower than Lamport's fast-algorithm in the case of absence of competition because of more memory accesses (LamportFast has 7 accesses, Triangle 12, YA95 13 and AK01 19). In the presence of high competition, the Triangle algorithm is significantly faster than Lamport's fast-algorithm and most software solutions for mutual exclusion that we investigated and even faster than some solutions that use stronger hardware primitives than atomic load and store (eg, compare-and-assign). Note, the Triangle algorithm is not unequivocally better than every existing software solution, but it is unequivocally better than any existing software algorithm for both maximal and minimal contention.

1.1 | Overview

Motivations for using software solutions are given in Section 2. The new Triangle algorithm is presented in Section 3. The safety properties of the algorithm, in particular, mutual exclusion, are verified in Section 4. Section 5 proves informally that the algorithm is starvation free. A formal proof of starvation free is presented in Appendix A. The proofs in Sections 4 and Appendix A were developed and verified using the proof assistant PVS.¹⁵ Section 6 shows how the verified algorithm is safely implemented under a code-transforming, total store order (TSO) memory-model. Section 7 presents synthetic (micro) benchmarks to compare throughput performance at maximal and minimal contention for the variants of the Triangle algorithm with competing algorithms and evaluates the experimental results. Conclusions are drawn in Section 8.

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 M0-3¹⁶ and Microchip PIC24 8/16-bit¹⁷) 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 *have no atomic instructions*, possibly not even atomic reads and writes, even with multiple cores, because cache coherency is complex, expensive, and creates energy/heat issues. In these embedded systems, it is 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 WIFI/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 (Linux Light) 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.

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 algorithm = 2400 bytes, which is still small, as embedded processors often have 8-16 M bytes of memory on chip.

3 | THE TRIANGLE ALGORITHM

Figure 2 shows the Triangle algorithm. It protects the critical section by means of the functions `entryTriangle` and `Triangle`. The private variable `fm` is used to transfer whether a thread succeeded in `entryFast` from the entry to the exit protocol. The global structure of the algorithm is presented in Section 3.1. Section 3.2 contains our variation of Lamport's fast-algorithm, which is used as a subroutine in the Triangle algorithm. Styer's proposal is discussed in Section 3.3. YA95's and AK01's proposals are discussed in Section 3.4.

3.1 | Notations and global structure

The threads are numbered $0, \dots, N-1$. Variables p, q, r range over the thread identifiers. Typewriter font is used for shared variables, and slanted font for private variables. If v is a private variable, then $v.q$ means the private value of v for thread q . In particular, $pc.q$ is the program counter of thread q . The notation $q \text{ at } \ell$ is used to indicate that thread q is at line ℓ , ie, that $pc.q = \ell$. Similarly, if ℓ is a set of line numbers, $q \text{ in } L$ indicates that q is at one of the lines $\ell \in L$. In the pseudo code for Figures 3 and 5, consistent line numbering is used between the PVS proof and article (which is why numbering starts at 22). The PVS proof script is available for inspection and reuse.¹⁸ Finally, the command `await(B)` is defined and implemented by `while  $\neg B$  do skip endwhile`.

3.2 | Variation of Lamport's fast-algorithm

Figure 3 shows in pseudo code the variation of Lamport's functions `entryFast` and `exitFast` used in the Triangle algorithm, along with Lamport's fast-algorithm with trivial syntactic changes listed below. Triangle is a *variation* because it eliminates the unbounded waiting loop of Lamport's fast-algorithm, ie, where the algorithm restarts (`goto`) in multiple locations, which can result in starvation. The fast-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. If the fast track is empty, an arriving thread p finds $y = N$ at line 22, announces itself by writing $x \leftarrow p$ at line 24, passes line 25, and then sets the locks 22 and 25 by $y \leftarrow p$ at line 27. If thread p is fast enough, it then finds $x = p$ at line 28 and succeeds at line 33.

The function fails for threads that find $y \neq N$ at lines 22 or 25. If thread p reaches line 28 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 28 while $x = q$ and succeeds, or all threads q that reach line 28 find $x \neq q$. The latter case occurs if all contending threads delay before line 28, and a new thread changes x at line 24 but cannot join the contending threads because $y \neq N$ due to line 27.

```

function entryTriangle( $p$ ) : bool =
  var  $fa$  : bool  $\leftarrow$  entryFast( $p$ ) ;
  if  $\neg fa$  then entrySlow( $p$ ) endif ;
  entryBinary( $fa$ ) ;
  return  $fa$ 
end entryTriangle .

function exitTriangle( $p, fa$ ) =
  exitBinary( $fa$ ) ;
  if  $fa$  then exitFast( $p$ ) else exitSlow( $p$ ) endif
end exitTriangle .

var  $fm$  : bool ;                                // private variable of  $p$ 
 $fm \leftarrow$  entryTriangle( $p$ ) ;                    // entry protocol
CS ;                                              // critical section
exitTriangle( $p, fm$ ) ;                             // exit protocol

```

FIGURE 2 Triangle algorithm for thread p

```

var b : array[N] of volatile bool ← [N] false ;
var x : volatile N ;
var y : volatile N ← N ;

function entryFast(p) : bool =
22  if y ≠ N then return false endif ;
23  b[p] ← true ;
24  x ← p ;           // RACE
    Fence() ;
25  if y ≠ N then
26    b[p] ← false ;
    return false endif ;
27  y ← p ;           // RACE
    Fence() ;
28  if x ≠ p then
29    b[p] ← false ;
    Fence() ;           // OPTIONAL
30    for k ← 0 to N - 1 while y = p do
31      await y ≠ p ∨ ¬b[k] endfor ;
32    if y ≠ p then return false endif ;
    endif ;
33  return true
end entryFast .

function exitFast(p) =
36  y ← N ;
37  b[p] ← false
end exitFast .

```

(A) Variation

```

start :
23  b[p] ← true ;
24  x ← p ;
    Fence() ;
25  if y ≠ N then
26    b[p] ← false ; Fence() ;
    await y = N ; goto start endif ;
27  y ← p ;
    Fence() ;
28  if x ≠ p then
29    b[p] ← false ;
    Fence() ;
30    for k ← 0 to N - 1 do
31      await ¬b[k] endfor ;
32    if y ≠ p then
    await y = N ;2
    goto start
    endif
    endif
    CS ;
36  y ← N ;
37  b[p] ← false ;

```

(B) Lamport's fast

FIGURE 3 Variation F of Lamport's fast-algorithm

```

thread(p) =
1  start : NCS ;
2  x ← p ;
3  if y then goto aside endif ;
4  y ← true ;
5  if x ≠ p ∨ z then goto aside endif ;
6  CS ;
7  y ← false ; goto start ;
8  aside :
9  entrySlow(p) ;
10 z ← true ;
11 CheckVars( ... ) ;
12 CS ;
13 y ← false ; z ← false ;
14 exitSlow(p) ; goto start .

```

FIGURE 4 Abstract version of Styer's proposal (erroneous)

In the first case, thread q resets y in its exit function, and all other threads fail. In the second case, the contending threads enter regions 29 to 32, and y holds the identifier of one of these threads. This thread enters the CS and resets $y \leftarrow N$ in the exit protocol. All contending threads $q \neq y$ fail.[†]

When thread p finds $x \neq p$ at line 28, it has no means to decide which of the two alternatives applies. It therefore has to wait at the lines 30, 31, until any thread q with $x = q$ at line 28 has passed the CS. The booleans $b[q]$ serve to indicate this. They are set in line 23 and reset in lines 26, 29, 37. When thread p has waited long enough and then satisfies $y = p$, it can succeed.

Instead of the Triangle algorithm of Figure 3(A), entryFast can be encapsulated in an always succeeding entry function:

[†]A small modification to Lamport's fast-algorithm is removing the **await** associated with line 32 because transferring back to start immediately enters the same await associated with line 26, unless y has changed.^{10, § 15} This modification is used in all the experiments (see Section 7) because it generates the best results on the SPARC computer.

```

function entryFastSuc(p) =
    var fa: bool;
    do fa ← entryFast(p) while ¬fa
end entryFastSuc.

```

Together with exitFast, this gives a MX algorithm quite close to Lamport's fast-algorithm.⁵

We have made some changes, however. Firstly, a trivial syntactic change of Lamport's fast-algorithm to number threads from 0 to $N - 1$ instead of 1 to N and use the special value $y = N$ rather than $= 0$ as the nonthread value. Secondly, line 22 is absent in Lamport's fast-algorithm,^{5, Fig. 2} and the return statements in lines 26 and 32 replace waiting and then backward jumps to line 23. Thirdly, the conjunction $y = p$ is added to the **for** loop in line 30 so thread p is released earlier from the **for** loop and goes to S . Fourthly, the disjunction $y \neq p$ is added to the **await** statement of line 31. The first change is convenient for our work. The third change gives better performance. The second and fourth changes are essential to ensure that the algorithm of Figure 2 is starvation free. Specifically, in our version of F , a failing thread does not retry but goes to S , which occurs when function entryFast return *false* in lines 22, 26, and 32. Line 22 is needed for progress; without it, a thread can remain waiting in loops 30 to 31 because of another thread (say q) repeatedly enters the CS setting $b[q]$ to *true* and *false* again. Line 22, however, forces thread q to S ; in line 31, a thread $\neq y$ quits the loop and goes to S .

After the design of a correct concurrent algorithm, an implementation can be constructed (eg, in the C programming language) and run on current hardware. However, any *correct* concurrent algorithm based on data races can be invalidated by the language compiler and hardware performing legitimate sequential optimizations. Therefore, it is necessary to augment a correct algorithm with annotations restricting these optimizations. These secondary requirements do not make a concurrent algorithm correct, rather they prevent a correct algorithm from being made incorrect.

The pseudo code in Figure 3 for component F of the Triangle algorithm illustrates two essential additions to establish correctness for a C-language implementation (volatile / Fence()) for one of the most common memory models, TSO (see Section 6 for details). Different compilers and hardware may require slightly different additions. The key point is that compilers and hardware normally supply sufficient features to make any correct software algorithm execute correctly. The same correctness additions must be added to the S and B algorithms. The correctness additions for algorithms S and B used in Section 7 are discussed in Buhr et al.¹⁰

3.3 | Comparison with Styer

Styer¹¹ tries to combine the benefits of Lamport's fast-algorithm with a tournament algorithm[†] like Kessels.¹⁹ Figure 4 gives an abstract representation of Styer's proposal (see Styer^{11, Figure 1}), in which Styer's tournament code is replaced by an arbitrary mutual-exclusion algorithm with entry and exit protocols entrySlow and exitSlow, respectively. To ease the comparison with Figure 3, variable names are reused. Variable y becomes boolean, and a boolean companion z is added, both initially *false*. (The actual variable relationship between the two algorithms is $y = \text{Lock}$, $x = \text{Turn}$, and $z = \text{Block}$.) For conciseness, the code is flattened, where NCS denotes the noncritical section, and a thread is *idle* at NCS. The CS occurs twice to keep the fast and the slow tracks separate.

The variable z ensures a new fast thread does not enter the CS when occupied by a slow thread. Procedure CheckVars has the converse task of blocking the winner of the slow protocol when a fast thread is in the CS. The implementation of CheckVars in Styer¹¹ depends on the details of the tournament solution employed and contains an error that we are unable to repair. Ignoring this error, it is sufficient to show that, even at the abstract level, the algorithm is in error as it allows the following scenario.

Assume threads t_0 and t_1 start at NCS. Thread t_0 enters and sets $x \leftarrow t_0$, $y \leftarrow \text{true}$, and reaches the CS. Thread t_1 sees $y = \text{true}$ at line 3 and goes to aside. Thread t_0 continues, leaves the CS, and exits. Thread t_1 traverses entrySlow and goes into the CS. Thread t_0 enters again, sets $x \leftarrow t_0$ and $y \leftarrow \text{true}$, and stops at line 5. Thread t_1 completes the CS, sets $y \leftarrow \text{false}$, and exits. Thread t_0 at line 5 continues, sees $x = t_0$, and enters the CS. Thread t_1 enters again, sees $y = \text{false}$, and enters the CS along the fast track. Hence, MX is violated. Note that removal of $y \leftarrow \text{false}$ at line 13 does not help, because this allows y to remain *true* forever, which eliminates the fast track.

3.4 | Comparison with YA95 and AK01

The YA95 algorithm deviates from Lamport's fast-algorithm more than the Triangle. In fact, it replaces the waiting loop in lines 30 to 31 of Figure 3 by an order N loop performed by a slow thread in the exit protocol after the CS. It uses an additional shared-variable Z to do this safely and to redirect threads through the slow path. The YA95 paper has no proof of correctness for its Fig. 3 algorithm, but we have no reason to doubt it. The AK01 paper is a significant revision of YA95 to eliminate the order N loop at the cost of additional complexity.

Although it is theoretically attractive to replace the waiting loop by an ordinary loop, or to eliminate it altogether, there is a performance cost as shown in Section 7.2. For minimal contention, the Triangle performs better than YA95 and AK01 because of its greater simplicity, resulting in less code along the fast path (see the end of Section 7.1 for details). For maximal contention, it also performs better because of simplicity and better redirection. Our maximal contention experiments show both YA95 and AK01 redirect almost all threads from the fast to the slow path, whereas the Triangle algorithm has an equal number of threads traversing the fast and slow paths.

[†]A tournament algorithm uses divide-and-conquer, where only one of every D threads progresses to the next round, and the others busy wait. A tournament solution is based on an auxiliary solution (MX) for D threads, where D is a small number > 1 , usually $D = 2$. A D -ary tree is constructed with height $\lceil \log_D N \rceil$, and each thread is assigned a starting position at one of the N leaves of the tree.

```

thread( $p$ ) =
21  NCS ;
22  if  $y \neq N$  then goto 38 endif ;
23   $b[p] \leftarrow true$  ;
24   $x \leftarrow p$  ; // RACE
25  if  $y \neq N$  then
26     $b[p] \leftarrow false$  ; goto 38 endif ;
27   $y \leftarrow p$  ; // RACE
28  if  $x \neq p$  then
29     $b[p] \leftarrow false$  ;  $li \leftarrow \{0 \dots N - 1\}$  ;
30    if  $y = p \wedge \text{exists } k \in li$  then
31      await  $y \neq p \vee \neg b[k]$  ; remove  $k$  from  $li$  ; goto 30
    endif ;
32    if  $y \neq p$  then goto 38 endif ;
    endif ;
33  entryBinary( $true$ ) ;
34  CS ;
35  exitBinary( $true$ ) ;
36   $y \leftarrow N$  ;
37   $b[p] \leftarrow false$  ; goto 21 .

38  entrySlow( $p$ ) ;
39  entryBinary( $false$ ) ;
40  CS ;
41  exitBinary( $false$ ) ;
42  exitSlow( $p$ ) ; goto 21 .

```

FIGURE 5 Transition system of the Triangle algorithm

The reason for equal traversals is that the Triangle algorithm essentially alternates between the two paths. Under maximal contention, the Triangle F algorithm almost always generates a candidate (rather than redirecting to S), as the probability of executing the loop at lines 30 to 31 is low (less than 5% based on experiment results). Similarly, S has approximately $N - 1$ threads competing and can always generate a candidate. These two candidates arrive at B , which must have fairness to prevent starvation. In the experiments, Peterson's algorithm is used for B , which is first-come first-served (FCFS),⁸ resulting in alternation between F and S . This alternation implies that the probability of entering via F is about 0.5. As the probability of entering the CS via F or S is equal, the expected time for a complete traversal is the average of the times for traversing F and S . This result is better than for YA95 and AK01 because in these algorithms, under maximal contention, virtually all threads enter the CS via the longer S .

4 | CORRECTNESS OF THE TRIANGLE ALGORITHM

Correctness means there is never more than one thread in the CS and every competing thread eventually reaches the CS and returns to NCS. The first requirement is a safety property: no bad states are reachable, ie, multiple threads in the CS. The second requirement is a progress property: some states eventually lead to other states. To formally prove that the algorithm has these properties, a formal model is constructed in Section 4.1.

The formal model is a transition system with initial states and transitions between states. A state is *reachable* when it can be reached from an initial state by a sequence of transitions. A condition that holds for all reachable states is called an *invariant*. To prevent discussing individual states, the verification of safety is done in Section 4.2 completely in terms of invariants. The transition system is also used in the proof of progress in Section 5 and Appendix A.

4.1 | Transition system

The state of the system comprises the values of all shared and private variables, including program counters. The transitions are the steps of the threads, in which the state is modified by executing the command at the current line number. These transitions are given in Figure 5, which combines Figures 2 and 3 and eliminates the function calls and the private variables fa and fm . The line numbers are used extensively in the invariants and also in the proof of progress.

To analyse the atomic steps of the for-loop of line 30 of Figure 3, a private set-valued variable $li.p$ is introduced for the set of threads k that have yet to be treated in the loop of thread p . The for-loop is therefore transformed into a kind of while-loop in Figure 5, with a repeated nondeterministic choice of $k.p = -li.p$. Line numbers 21, 33, 34, 35, and 38 and beyond stand for abstract transitions. The other line numbers stand for atomic steps. The verification given below closely follows the formal verification of the algorithm by means of the proof assistant PVS.

Remark 1. In Figure 5, a nondeterministic choice for $k.p$ in $li.p$ is used to provide more generality so that the proof covers implementations where the range is traversed in any order. A nondeterministic choice also makes it somewhat easier in the correctness proof. For instance, the invariants $lq5$ and $Kq3$ below are more elegant than is possible with a loop index.

4.2 | Proof of mutual exclusion

The outline for the proof of MX starts back at the triangle in Figure 1, where it is assumed the selected S and B algorithms are correct. Mutual exclusion means at most one thread is in the CS. Algorithm B , as a correct binary MX algorithm, sends only one thread into the CS, provided it gets only one thread from each of its incoming arrows. Algorithm F sends only one successful thread to B because of the correctness of our version of Lamport's fast-algorithm. Algorithm S gets all threads that fail in F and, as a correct MX algorithm, sends only one thread to B . Therefore, the Triangle algorithm satisfies mutual exclusion. It follows that the MX proof for the Triangle algorithm must show first a reduction to the correctness of F , followed by a proof of F .

This statement is made precise in the following way. In Figure 5, CS is represented by the lines 34 and 40. Mutual exclusion means that, if a thread q is in the CS, then q is the only one. Therefore, mutual exclusion is formally specified as the invariant validity of the predicate MX:

$$\text{MX: } q \text{ in } \{34, 40\} \wedge r \text{ in } \{34, 40\} \Rightarrow q = r.$$

Here and henceforth, we use implicit universal quantification over all free variables, here q and r . The proof of such invariants must deal with all possible steps of the algorithm, performed by any of the threads. There are two ways to prove that a predicate is an invariant. Either by showing it is logically implied by some other invariants or by showing it holds initially and is preserved at every step for every thread. An invariant is *preserved by a step* if it is not invalidated by the step, ie, if the validity of the invariant in the state before the step implies its validity in the state resulting from the step.

Almost every invariant Iq_i , depends on one or two free variables of type thread, q and r . The validity of Iq_i is then claimed for all values of q and r , and it should be invariant under steps of all threads. For some invariant, it may be the case that only thread q 's steps threaten it; for another invariant it may be thread r . And there are invariants that are threatened by steps of all threads. Therefore, in the proof of every invariant, the steps of all threads must be considered. For the theorem prover, the acting thread is called p .

Algorithms S and B guarantee mutual exclusion between entry and exit, where S applies to N threads and B applies to two threads, with the thread identifiers false and true. This mutual exclusion is the invariant validity of the predicates:

$$\begin{aligned} \text{MxS: } & q \text{ in } \{39 \dots 41\} \wedge r \text{ in } \{39 \dots 41\} \Rightarrow q = r, \\ \text{MxB: } & q \text{ at } 34 \wedge r \text{ at } 40 \Rightarrow \text{false}. \end{aligned}$$

Note that the entry and exit code of the slow algorithm are not included in MxS , just as the entry and exit code of the binary algorithm are not included in MxB .

It therefore remains to prove

$$\text{MxF: } q \text{ in } \{33 \dots 36\} \wedge r \text{ in } \{33 \dots 36\} \Rightarrow q = r.$$

Indeed, MX easily follows from MxS , MxB , MxF , by distinguishing the four cases when q and r are each in one of the line 34 or 40, allowed in the left-hand side of MX. In this way, mutual exclusion for the Triangle algorithm is reduced to proving the mutual exclusion of algorithm F as expressed by predicate MxF . What follows is the proof of MxF , ie, mutual exclusion for our version of Lamport's fast-algorithm.

Predicate MxF can only be invalidated when some thread, say p , enters line 33. As thread p can enter line 33 only from the lines 28 or 32, MxF can only be invalidated by the step 28 or 32, which is expressed by saying that predicate MxF is *threatened* only by these steps. MxF is *preserved* by these steps because of the additional invariants

$$\begin{aligned} lq1: & q \text{ in } \{27, 28\} \wedge r \text{ in } \{33 \dots 36\} \Rightarrow x \neq q, \\ lq2: & q \text{ at } 32 \wedge r \text{ in } \{27, 28, 33 \dots 36\} \Rightarrow y \neq q. \end{aligned}$$

Indeed, MxF is preserved because, if r is in $\{33 \dots 36\}$, thread q does not go from 28 or 32 to 33 because of $lq1$ or $lq2$, respectively.

Predicate $lq1$ can only be invalidated when some thread, say p , goes from line 25 to line 27, or from one of the lines 28, 32 to line 33. The step from line 28 is harmless, because when q is and remains in $\{27, 28\}$ and r goes from 28 to 33, then $x = r \neq q$. Therefore, $lq1$ is threatened only by the steps 25 and 32. $lq1$ is preserved by step 25 because of the invariant

$$lq3: q \text{ in } \{33 \dots 36\} \Rightarrow y \neq N.$$

Indeed, if r is in $\{33 \dots 36\}$, then q cannot go from 25 to 27 because $lq3$ with $q \leftarrow r$ implies $y \neq N$. The proof assistant PVS verifies this fact easily. Predicate $lq1$ is preserved by step 32 because $lq2$ with q and r interchanged implies that, when q is in 27, 28, thread r cannot go from 32 to 33.

Treatment of line 32 and the associated invariant $lq2$ is postponed until later.

Predicate $lq3$ is preserved by the step of q from 32 to 33, because this step is done only when $y = q \neq N$. Therefore, $lq3$ is threatened only by the step from 28 to 33 and by step 36 that resets $y \leftarrow N$. Predicate $lq3$ is preserved by step 28 because of the additional invariant

$$lq4: \quad q \text{ at } 28 \wedge x = q \Rightarrow y \neq N.$$

Predicate $lq3$ is preserved by step 36 because of MxF : when thread q is in 33...36 and thread p resets $y \leftarrow N$ in step 36, then MxF implies $q = p$, so that q goes to line 37.

Predicate $lq4$ is preserved by step 27 because it sets $y \leftarrow p \neq N$. It is therefore threatened only by the modification of y in step 36. It is preserved because of $lq1$.

Remark 2. Note, $lq1$ is preserved because of $lq3$, $lq3$ is preserved because of lq , and $lq4$ is preserved because of $lq1$, which gives the circular dependency $lq1 \leftarrow lq3 \leftarrow lq4 \leftarrow lq1$. $lq1$ is preserved when $lq3$ holds in the precondition of every step. $lq3$ is preserved when $lq4$ holds in the precondition of every step. $lq4$ is preserved when $lq1$ holds in the precondition of every step. Therefore, if all three invariants hold in the precondition of every step, they also hold in the postcondition, which is sufficient for the proof. In the same way, the invariant MxF is used in the proof of $lq3$, while $lq1$ and $lq2$ are used in the proof of MxF .

Predicate $lq2$ is preserved by step 25 because thread r only goes from 25 to 27 when $y = N \neq q$. It is preserved by step 27 because, when q is at 32, any thread p that sets $y \leftarrow p$ in step 27 has $p \neq q$. For the treatment of $lq2$, two possibilities for the step at line 30 are distinguished. Step 30B is the *true* case and goes to line 31; step 30E is the *false* case and goes to line 32. Thread p can execute step 30E if and only if the set $li.p$ is empty. Therefore, $lq2$ is threatened only by step 30E. It is preserved by step 30E because of the new postulate

$$lq5: \quad q \text{ in } \{30, 31\} \wedge y = q \wedge r \text{ in } \{27, 28, 33 \dots 36\} \Rightarrow r \in li.q.$$

Predicate $lq5$ is preserved by step 25 because when $y = q$, thread r cannot go from 25 to 27. It is preserved by step 29 for q , because $li.q$ becomes the set of all threads, which includes r . Predicate $lq5$ is therefore threatened only by step 31 for thread q . Thread q only removes r from $li.q$ step 31 when $y \neq q$ or $b[r]$ is *false*. It follows that $lq5$ is preserved because of the new postulates

$$\begin{aligned} lq6: \quad & q \text{ in } \{24 \dots 29\} \Rightarrow b[q], \\ lq7: \quad & q \text{ in } \{28 \dots 31\} \wedge y = q \wedge r \text{ in } \{33 \dots 36\} \Rightarrow b[r]. \end{aligned}$$

Predicate $lq6$ is inductive: it holds initially and is preserved in every step, without needing other invariants in the precondition.

Predicate $lq7$ is threatened by the steps 27 and 28. It is preserved by step 28 because of $lq6$. It is preserved by step 27 because of the new postulate

$$lq8: \quad q \text{ at } 27 \wedge r \text{ in } \{33 \dots 36\} \Rightarrow b[r].$$

Predicate $lq8$ is threatened by the steps 25, 28, 32. Step 25 preserves $lq8$ because thread q can only go from 25 to 27 when $y = N$, so that $lq3$ implies that r is not in $\{33 \dots 36\}$. Step 28 preserves $lq8$ because $b[r]$ when thread r goes from 28 to 33. Step 32 preserves $lq8$ because thread r only goes from 32 to 33 when $y = r$, so that $lq2$ implies that q is not at 27.

This concludes the proof of mutual exclusion, ie, of the invariant MX .

5 | PROGRESS (INFORMAL)

The design of a MX algorithm is always a balancing act between the safety requirement of mutual exclusion and the progress requirement of absence of deadlock or livelock. The safety requirement is the more critical one, and it was dealt with above.

This section covers the progress requirement. The main result to be proved is starvation freedom: if thread p is anywhere in the entry protocol, it will eventually reach the critical section, and from the critical section it will eventually become idle again. The second assertion to be proved is that the algorithm is fast, ie, when some thread p is competing while there are (and remain) no other competing threads, thread p reaches the critical section in a bounded number of steps.

These assertions are proved formally in Appendix A by means of UNITY. Here, we argue informally for ease of reading.

5.1 | Starvation freedom

The proof of starvation freedom is based on the obvious invariant from Figure 5.

$$Kq0: \quad b[q] \Rightarrow q \text{ in } \{24 \dots 29\} \cup \{33 \dots 37\}.$$

Now, let p be an arbitrary thread somewhere in the entry protocol. A basic assumption, needed for almost every concurrent algorithm, is weak fairness, which means a thread that is continuously enabled, will eventually do a step. Referring to Figure 5, when thread p is in $\{23 \dots 29\}$, it eventually reaches the await at line 31, entryBinary at line 33, or entrySlow at line 38. If thread p remains waiting at line 31, then $y = p$ holds and remains valid. Hence, all new competing threads are redirected to the slow path. It follows that all threads in the region $\{23 \dots 29\} \cup \{33 \dots 37\}$ will eventually leave this region, and no new threads can enter it. When this region is empty, the invariant $q0$ implies that all variables $b[q]$ are *false*, and thread p will eventually loop. Now this thread p will eventually reach entryBinary at line 33 or entrySlow at line 38.

5.2 | The algorithm is fast

Here, we need the obvious invariants

$$\begin{aligned} Kq1: & \quad y = q \Rightarrow q \text{ in } \{ 28 \dots 36 \}, \\ Kq2: & \quad 0 \leq y \leq N \end{aligned}$$

Assume thread p starts competing with all other threads idle (NCS), and they remain idle. When thread p is at line 22, the invariants $Kq1$ and $Kq2$ imply that $y = N$. Therefore, thread p subsequently executes lines 22, 23, 24, 25, 27, 28 and reaches entryBinary at line 33. As there are no competing threads, it then enters CS. This shows that the algorithm is fast.

6 | IMPLEMENTATION CORRECTNESS

Concurrent software-algorithm correctness only relies on 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 effect 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 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-the-art automation often over-constrains an algorithm resulting in slower execution.

Figure 3 shows the necessary implementation additions for preventing incorrect optimizations by the compiler and hardware. The transition system of Figure 5 is a formal model and, hence, does not need fences because there are no implementation optimizations, but fences are needed in its implementation (called *Triangle*) in Section 7. The amount of fencing is related to the design and complexity of an algorithm and can result in reduced throughput in performance experiments. For example, the Triangle implementation needs 2 fences (1 fence is optional) for F , YA95 needs 3, and AK01 needs 4, not including fences needed for the S and B algorithms. Only the additions in Figure 3 are discussed in detail.

The **volatile** qualifier controls static compiler transformations, such as movement, elision, and caching of values into registers. For example, at line 31, if the compiler caches y or $b[k]$ into a register for the **await** loop, changes to these variables 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 three fences 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,²⁰ 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 4 and 5 and Appendix A.

In general, fences are inserted after every write to a shared variable that is followed by a read from a shared variable. 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 Section 4 and Appendix A, instead of multiple versions based on possible code movement. For the first two fences of Figure 3, 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 performance difference with and without the fence is undetectable because most threads are redirected to the slow path at the first two redirection points, so no more than 5% and mostly less than 1% of the threads get to the third redirection point and, hence, would be affected by the presence of a fence. The third fence was removed for the experiments to show the best possible performance.

The first fence prevents the read of y at line 25 from moving before the disjoint writes of $b[p]$ and x at lines 23 to 24 (or being elided and using the read of y at line 22), as in

```

22   if  $y \neq N$  then return false endif;
22(a) tempy  $\leftarrow y$ ;                // move read before disjoint writes
23    $b[p] \leftarrow true$ ;
24    $x \leftarrow p$ ;                    // RACE
25   if tempy  $\neq N$  then
26      $b[p] \leftarrow false$ ; return false endif;
27    $y \leftarrow p$ ;                    // RACE
28   if  $x \neq p$  then

```

This relaxation allows the scenario with initially $y = N$, where q_1, q_2 reach line 24 simultaneously. Now, q_1 executes lines 24 to 28 and enters the CS, and then q_2 executes lines 24 to 28 and also enters the CS, violating MX. If q_2 reads y at line 25, the value is unequal to N because of q_1 's assign at line 27, so q_2 executes line 26.

The second fence prevents the read of x at line 28 from moving before the disjoint writes of $b[p]$ and y at lines 26 to 27, as in

```

24       $x \leftarrow p$ ;                                // RACE
24(a)   $tempx \leftarrow x$ ;                            // move read before disjoint writes
25      if  $y \neq N$  then
26           $b[p] \leftarrow false$ ; return false endif;
27       $y \leftarrow p$ ;                                // RACE
28      if  $tempx \neq p$  then
29           $b[p] \leftarrow false$ ;

```

This relaxation allows the scenario with initially $y = N$, where thread q_1 executes lines 22 to 25 and then thread q_2 does the same. Hence, both threads at line 25 have $tempx = p$. The threads continue and enter the CS, violating MX. If q_1 and q_2 read x at line 28, x is equal to either q_1 or q_2 , so one thread executes line 29.

The third fence prevents a read of $b[k]$ on line 31 from moving before the write of $b[p]$ on line 29 for $k \neq p$:

```

29       $b[p] \leftarrow false$ ;
30      for  $k \leftarrow 0$  to  $N - 1$  while  $y = p$  do
31          await  $y \neq p \vee \neg b[k]$  endfor;
32      if  $y \neq p$  then return false endif;

```

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

Finally, all elements of all shared data-structures are cache-line aligned so there is no opportunity for false-sharing. While this approach wastes a small amount of memory, the total memory costs are very small, in the hundreds/thousands of bytes.

7 | PERFORMANCE EXPERIMENT[§] ¶

Two performance experiments, maximal and minimal contention, are used to compare the performance differences between the Triangle 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. 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,²¹ and the algorithm implementations are checked by the Relacy Race Detector.²²

7.1 | 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 CurrTid; // current thread in CS
    CurrTid = id;                        // RACE
    Fence( )§;                          // optional
    for ( int i = 1; i <= 100; i += 1 ) // delay
        if ( CurrTid != 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

[§]This section is based on Buhr et al,^{10,§ 21} as the same experimental setup is used.

[¶]See Buhr et al^{10,p.657} for an explanation of why this fence may be optional.

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 unblocks 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 seconds. The median value of the five results is plotted.

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, equals 64 cores, running Linux v3.13.0-49, compiling with gcc 4.9.2;
2. Supermicro SYS-8017R-TF+ with four sockets, each containing an Intel Xeon 8 core 2.6 GHz E5-4620V2, equals 32 cores, turbo-boost on, running Linux v3.13.0-49, compiling with gcc 4.9.2; 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 differences 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 multsocket 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 behaviour 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.

Two versions of the Triangle algorithm are considered: *TriangleMod* is the name for the modularized version shown in Figure 2. *Triangle* is the name for a completely flattened version of that in Figure 5 with all the function calls replaced by **gotos**.

In practice, with compiler optimization level `-O3`, all the function calls in *TriangleMod* are inlined generating a similar version to the hand-code inlined version. For the fast-algorithm *F*, the version in Figure 3 is used. For the slow algorithm *S*, the two best performing tournament algorithms are chosen from Buhr et al¹⁰: Taubenfeld-Buhr (TB) and Peterson-Buhr (PB). These algorithms ensure that the number of memory accesses and await statements for every thread in the slow path is logarithmic in N . Hence, the name *TriangleModTB* and *TriangleTB* are the names for the modularized/flattened Triangle algorithms using Taubenfeld-Buhr for algorithm *S*. For the binary algorithm *B*, Peterson's 2-thread algorithm^{23, Fig. 1} is chosen, because it gives good results having only a single fence to prevent hardware reordering and a single write-race to prevent livelock and starvation.

Given these choices for *F*, *S*, and *B*, the fast thread performs 8 write statements (**W**) and 4 read statements (**R**)

R, W, W, R, W, R, **W**, **W**, R, CS, **W**, W, W

traversing the *F* and *B* portions of the Triangle algorithm, and the 4 operations in bold are from *B*.

To contrast the Triangle algorithm with other software algorithms, the following algorithms were selected: PB, TB, and LamportFast. Comparing with PB and TB shows the effect of adding *F* and *B* to these algorithms. Comparing with LamportFast shows the effect of adding *S* and *B* to the new variant of this algorithm. To contrast the Triangle algorithm with two similar algorithms, YangAnderson¹² and AndersonKim¹³ were selected both using the same PB, TB for *S* and Peterson for *B*. Finally, one atomic hardware-assisted algorithm is included to give a sense of overall performance. The algorithm is Mellor-Crummey and Scott (MCS),²⁴ which augments a software solution with an atomic fetch-and-store and CAS instruction. MCS is *fast* (constant time for one thread) and FCFS (first-come first-served) and hence starvation free. Faster hardware-assisted locks exist, but only a handful are starvation free; most real-world implementations have starvation. MCS and its variants are used in the Linux kernel to obtain fast/fair locking.²⁵

7.2 | Experimental results

Figure 6(A-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 7(A-C) shows the relative standard deviation, $rc_v = \frac{\sigma}{\mu} \times 100$, where σ is the standard deviation and μ is the average, for the maximal contention experiment, which is a percentage of the coefficient of variation (c_v) representing a normalized measure of dispersion of fairness for each algorithm. If an algorithm is perfectly fair, then the count values for each thread are essentially equal (modulo small differences at start-up and close-down), resulting in an rc_v of essentially zero. The more entry counts differ, the higher the percentage of unfairness. The graph is in log scale to spread out the curves into ranges: 1% to 10%, 10% to 100%, and 100% to 1000%.

Figure 8(A-C) shows the entry counts from the minimal contention experiment, that is, an access with no contention, one thread, but $N = 1 \dots 32$. 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

FIGURE 6 Critical-Section Entry-Counts, Maximal Contention: $T = N = 1 \dots 32$, 60 second, algorithm performance, higher value is better

FIGURE 7 Relative standard deviation, maximal contention: $T = N = 1 \dots 32$, 20 s, fairness among threads, where 0% is perfect fairness. Algorithms not shown have less than 1% unfairness, eg, FCFS 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 the tournament algorithms take $O(\log N)$, resulting in steps at powers of 2.

7.3 | Experimental evaluation

Figure 6 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. The global observation across all architectures is the tight grouping of algorithms, which includes MCS, with only one outlier. For all architectures, the LamportFast algorithm has the slowest throughput for maximal contention because the algorithm is solely optimized for minimal contention. While the Triangle algorithms use a modification of LamportFast, any contending threads are redirected to the slow algorithm, which is optimized for high contention. Interestingly, LamportFast experienced significant jitter on the AMD architecture; we were unable to determine the reason, even after extensive adjustments to the algorithm and its environment. The AndersonKim and

FIGURE 8 Critical-Section Entry-Counts, Minimal Contention: $T = 1$, $N = 1 \dots 32$, 60 second, higher value is better

YangAnderson algorithms are always the slowest algorithms. At 32 threads/cores, the spread of the grouped algorithms (ignoring LamportFast) is 57%, 66%, and 21%, on the AMD, Intel, and SPARC, respectively. The spread is much closer ignoring the AndersonKim and YangAnderson algorithms: 18%, 28%, and 9%, respectively. The spread is fairly consistent across the entire range of threads, except at the start ($T = 1$), which is discussed in the minimal contention results. It is impressive that software algorithms can be competitive with the hardware-assisted MCS algorithm. In fact, some of the software algorithms are slightly faster than MCS for most or all of the thread/CPU range. The reason the software algorithms are competitive is because only the best algorithms were selected from prior work¹⁰ and extensive work has been done to provide extremely high-performance implementations for these algorithms. The key point is that all the Triangle algorithms are in the highest performant group showing the algorithm design allows any of the best high-contention algorithms to be used on the slow path, while the fast-path algorithm does not inhibit performance.

Figure 7 shows the maximal contention fairness for the AMD x86, Intel x86, and SPARC architectures. There are some effects at the powers-of-two boundaries because of the binary trees used in the tournament algorithms. Only the MCS algorithm has FCFS service, which implies perfect fairness, and as such, it does not appear on the graph because all values are 0. All software algorithms investigated have some level of unfairness. For tournament algorithms, the rc_i is zero for powers of 2 and rises and falls in between these values depending on the nonpower

of 2 mechanism used. Peterson-Buhr has a smooth, small amount of unfairness because of the minimal binary tree and hence is in the range 10% to 30%. Taubenfeld-Buhr has more unfairness right after a power of 2, which then diminishes as N is increased to the next power of 2, and hence is in the range 10% to 200%. LamportFast does not preclude starvation, and so has varying amounts of unfairness, depending on the nondeterminism during an experiment, and hence is in the range 10% to 300%. All of Triangle algorithms have two forms of unfairness: one thread in the fast path competes with multiple threads in the slow path, and the slow-path algorithms have unfairness as stated previously, and hence these algorithms are in the range 10% to 200%. The Triangle experiment was modified to count the number of times a thread makes consecutive fast-path entries into the CS. The results show the number of fast-path consecutive-entries is randomly distributed across the threads. Hence, the thread performing the exit protocol receives no long-term preferential treatment to the fast path because it last set variable y ; ie, there is no cache effect causing unfairness. In general, some unfairness is good for performance because it allows immediately available threads to barge ahead of waiting threads that must be restarted. However, too much unfairness leads to some significant level of starvation.

Figure 8 is the minimal ($T = 1, N = 1..32$) contention results for the 12 algorithms plus Burns2 on the 3 architectures. MCS and LamportFast should have the best performance because both have short fixed-length paths for zero contention, which creates a flat line across the entire thread range. Similarly, the Triangle algorithms have a fixed-length but slightly longer path for zero contention and also have flat lines across the entire thread range. The maximal difference among the algorithms on the three architectures is 66%, 34%, 22%, respectively. It is surprising to see a few algorithms are equal or slightly better than LamportFast on the AMD, and for TrianglePB, TriangleModTB and LamportFast to be faster than MCS on the SPARC, even with longer fixed-length instruction sequences. We conjecture the hardware pipeline manages to overlap some of the execution. On all the architectures, the Triangle algorithms perform equal or just below LamportFast, showing the fast component of the algorithm design optimizes minimal contention. The nonfast-path algorithms decrease across the thread range because the instruction path through these algorithms increases with the number of threads; tree-based algorithms have a series of steps in the curve at powers of two as the depth of the tree increases, while scanning algorithms have a steady decrease across the range. For interest, the Burns2^{26, Fig. 2, p. 170} algorithm tested in Buhr et al¹⁰ is shown because it equals or exceeds MCS up to 12 threads on the AMD and exceeds LamportFast up to 8 threads on the Intel. Given that the thread id is randomized for Fast experiments, there is no short path for Burns2, which must execute an $O(N)$ search. Currently, we have no explanation for this phenomenon and intend to investigate it in the future.

8 | CONCLUSION

Software algorithms for mutual exclusion rarely have good performance for both minimal and maximal contention workloads. LamportFast is the only complete algorithm with an *optimal* fast path for minimal contention, while tournament algorithms are very good for maximal contention without FCFS (ie, some unfairness). It is also rare to combine approaches and get the best qualities of both.

The algorithms Triangle, YA95, and AK01 all combine approaches with the same structure of F , S , and B , connected in the form of a triangle, where each algorithm has a unique F . This structure allows a programmer to select the binary and slow-path algorithms with the best properties needed for a specific application. And unlike LamportFast, these algorithms are starvation free if the S and B algorithms are also starvation free.

The Triangle is different from YA95 and AK01 as its *fast* path is based on Lamport's fast-algorithm, allowing it to be simpler having fewer variables and less code. In fact, LamportFast and Triangle have 7 memory accesses for the fast path, while YA95 has 9 and AK01 has 14 (plus 5 memory accesses to traverse the B algorithm when Peterson's 2-thread write-race is used for Triangle, YA95, and AK01.) As well, LamportFast and Triangle for the fast path have 2 fences (1 fence is optional), while YA95 has 3, and AK01 has 4 (there are also 1 fence for TB and 0 fences for PB used for S , and 1 fence for Peterson used for B).

The benefits of the Triangle simplicity are demonstrated in better performance. Triangle has higher throughput than YA95 and AK01 in all cases. Furthermore, one or more variants of the Triangle algorithm equals or exceeds the starvation-free MCS algorithm on three different architectures at maximal contention. Similarly, one or more variants of the Triangle algorithm are only slightly below the optimal LamportFast on three different architectures at minimal contention. Both maximal and minimal contention are two very important workloads for any lock to handle well. Essentially, the Triangle algorithm achieves its good overall performance by concurrent execution of the F and S algorithms, while YA95 and AK01 do not have this kind of concurrency under maximal contention.

As a result, we conjecture the Triangle algorithm will stand the test of time as one of the best *all-around* software solutions for mutual exclusion because it is so general with many excellent properties, making it highly configurable for low-cost embedded systems without atomic instructions.

Future work is to understand why a software lock requiring only atomic read/write can equal or outperform any hardware-assisted lock. As well, there are a number of performance cases among the software solutions that we do not understand. This work requires many experiments examining hardware performance counters to determine the particular part of the processor system that is causing the behaviour. Lamport's fast-algorithm and its application in the Triangle algorithm are *RW-unsafe*, meaning that reads and writes must be atomic. The algorithms do not work on hardware with flickering reads during a simultaneous write or that scrambles bits on simultaneous writes.⁹ In fact, we conjecture it is impossible to construct a fast-algorithm that is also *RW-safe*. An alternative approach is to design functions *entryFast* and *exitFast*, based on Burn's algorithm^{26, Fig. 2, p. 170} instead of Lamport's fast-algorithm, because of the observed high-performance of Burns' algorithm for minimal contention on the x86 architecture.

Finally, this paper is the result of a fruitful collaboration between theory and practice. Concurrent programming, software testing, formal correctness proofs, and hardware related issues require knowledge and capabilities rarely combined in a single person. All these aspects are essential to make this work sound and relevant.

ACKNOWLEDGMENTS

We would like to thank Aaron Moss for help with the proofs and Martin Karsten for help with the performance experiments. 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. *Commun ACM*. September 1965;8(9):569.
2. Knuth DE. Additional comments on a problem in concurrent programming control. *Commun ACM*. 1966;9(5):321-322. Letter to the Editor.
3. Anderson TE. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans Parallel Distrib Syst*. January 1990;1(1):6-16.
4. Aboulenein NM, Goodman JR, Gjessing S, Woest PJ. Hardware support for synchronization in the scalable coherent interface (sci). In: Proceedings., Eighth International Parallel Processing Symposium, 1994, Cancun, Mexico: IEEE; 1994:141-150.
5. Lamport L. A fast mutual exclusion algorithm. *ACM Trans Comp Syst*. 1987;5(1):1-11.
6. Alur Rajeev, Taubenfeld Gadi. Results about fast mutual exclusion. In: Real-Time Systems Symposium, Phoenix, Arizon, USA: IEEE; 1992:12-21.
7. Burns JE, Lynch NA. Mutual exclusion using indivisible reads and writes. In: Proceedings of the 18th Annual Allerton Conference on Communications, Control and Computing, Monticello, Illinois, USA; 1980:833-842. <http://groups.csail.mit.edu/tds/papers/Lynch/allertonconf.pdf> [Accessed on March 2014].
8. Lamport Leslie. The mutual exclusion problem: Part II—statement and solutions. *J ACM*. 1986;33(2):327-348.
9. Buhr PA, Dice D, Hesselink WH. Dekker's mutual exclusion algorithm made RW-safe. *Concurrency Comput: Pract Experience*. 2016;28(1):144-165.
10. Buhr PA, Dice David, Hesselink WH. High-performance N-thread software solutions for mutual exclusion. *Concurrency Comput: Pract Experience*. 2015;27(3):651-701.
11. Styer Eugene. Improving fast mutual exclusion. Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, PODC '92. New York, NY, USA: ACM; 1992:159-168.
12. Yang J-H, Anderson JH. A fast, scalable mutual exclusion algorithm. *Distrib Comput*. 1995;9(1):51-60.
13. Anderson JH, Kim Y-J. A new fast-path mechanism for mutual exclusion. *Distrib Comput*. 2001;14(1):17-29.
14. Anderson J, Kim Y, Herman T. Shared-memory mutual exclusion: major research trends since 1986. *Distr Comput*. 2003;16:75-110.
15. Owre S, Shankar N, Rushby J, Stringer-Calvert DWJ. PVS version 2.4, system guide, prover guide, PVS language reference. SRI International Computer Science Laboratory. <http://pvs.csl.sri.com> [Accessed March 2014]; 2001.
16. ARM. Cortex-m3 technical reference manual, arm ddi 0337e. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf [Accessed February 2017]; 2006.
17. Microchip. 16-bit mcu and dsc programmer's reference manual, ds70157f. <http://ww1.microchip.com/downloads/en/DeviceDoc/70157F.pdf> [Accessed February 2017]; 2011.
18. Hesselink WH. Mutual exclusion by the Triangle algorithm, PVS scripts. <http://wimhesselink.nl/mechver/triangle> [Accessed March 2014]; 2013.
19. Kessels JLW. Arbitration without common modifiable variables. *Acta Inf*. 1982;17(2):135-141.
20. Adve SV, Gharachorloo Kourosh. Shared memory consistency models: a tutorial. *Comput*. 1996;29(12):66-76.
21. Buhr PA, Dice D, Hesselink WH. Concurrent-locking. <https://github.com/pabuhr/concurrent-locking> [Accessed March 2014]; 2013.
22. Vyukov Dmitry. Relacy race detector. [Accessed March 2014]; 2009.
23. Peterson GL. Myths about the mutual exclusion problem. *Inf Process Lett*. 1981;12(3):115-116.
24. Mellor-Crummey JM, Scott ML. Algorithm for scalable synchronization on shared-memory multiprocessors. *ACM Trans Comp Syst*. 1991;9(1):21-65.
25. Linux 3.15, release notice. http://kernelnewbies.org/Linux_3.15 [accessed June, 2016]; 2014.
26. Burns JE. Symmetry in systems of asynchronous processes. *22nd Annual Symposium on Foundations of Computer Science, Nashville, TN, USA*. Los Angeles, CA, USA: IEEE; 1981:169-174.
27. Chandy KM, Misra J. *Parallel Program Design, A Foundation*. Boston, USA: Addison-Wesley; 1988.
28. Lehmann D, Pnueli A, Stavi J. Impartiality, justice and fairness: the ethics of concurrent termination. In: Even Shimon, Kariv Oded, eds. *Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 115. Berlin, Heidelberg: Springer; 1981:264-277.
29. Misra J. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. New York: Springer; 2001.
30. Hesselink WH. Correctness and concurrent complexity of the Black-White Bakery algorithm. *Formal Aspects Comput*. 2016;28:325-341.

APPENDIX A: PROGRESS (FORMAL)

The main result to be proved in this section is starvation freedom: if p is any thread at any line of the protocol, it will eventually return to the idle state, ie, to line 21. It follows that, if p is in the entry protocol, it will eventually reach the critical section, because there is no other way from the entry protocol to the idle state.

To prove both assertions, they are formalized by means of the *concept of execution*. Roughly speaking, an execution is an infinite sequence of states such that, for every state in the sequence, there is a step of the algorithm that goes to the next state in the sequence. The formal definition is given in Section A.3. The concept of execution is somewhat too liberal, because it does not guarantee a thread is scheduled, so a thread may fail to make further steps at any point. For example, there are executions in which a thread transfers to line 22 and then fails to do any further steps. Therefore, a *fairness assumption* is necessary to guarantee that every thread does “enough” steps. The main alternatives for fairness are discussed in Section A.2.

While executions are useful to formalize progress properties, they are inadequate to prove such properties in a systematic way. Section A.4 presents UNITY logic,²⁷ as a way to prove progress properties by only analysing states and step relations. This logic is used in Section A.5 to prove starvation freedom. The mechanical proof at Hesselink¹⁸ uses UNITY. Finally, Section A.6 proves that the door to the fast track cannot be closed forever.

A.1 Additional invariants

The following invariants are needed for the proof of progress:

$$\begin{aligned} Kq0: & \quad b[q] \Rightarrow q \text{ in } \{24 \dots 29\} \cup \{33 \dots 37\}, \\ Kq1: & \quad y = q \Rightarrow q \text{ in } \{28 \dots 36\}, \\ Kq2: & \quad 0 \leq y \leq N, \\ Kq3: & \quad q \text{ at } 31 \Rightarrow k.q \in li.q. \end{aligned}$$

It is easy to see that the predicates are preserved in every step. They hold initially because $pc.q = 21, y = N \neq q$, and $b[q] = \text{false}$ for all q .

A.2 Strong or weak fairness

If, in Figure 5, the disjunct $y \neq p$ is removed from the await statement in line 31, thread p can remain waiting indefinitely, because thread k can repeatedly pass through the CS along the fast track and set $b[k]$ repeatedly *true* and *false* again, while thread p only inspects $b[k]$ when it is *true*. Similarly, if line 22 is removed, threads k and p can reach the lines 25 and 28, respectively, with $x = k$ and $y = p$. Thread p proceeds to 31 because $x = k$, while k proceeds to the slow path because $y = p$. Now thread k can repeatedly pass through the CS along the slow track and set $b[k]$ repeatedly *true* and *false* again, while thread p only inspects $b[k]$ when it is *true*. With line 22, once thread k has reset $b[k]$ to *false* at line 26, while p remains waiting at line 31, thread k is unable to set $b[k]$ to *true* on its next entry.

For both scenarios, the modified algorithm is correct under the assumption of *strong fairness*, which means all executions are *strongly fair*, ie, executions in which every step that is enabled infinitely often is taken infinitely often (strong fairness was introduced in Lehmann et al²⁸ as fairness). Indeed, as $b[k]$ is infinitely often *false*, strong fairness implies that p eventually sees $b[k]$ as *false*.

A malicious scheduler might, however, repeatedly schedule thread p only when $b[k]$ is *true*. This would induce starvation of thread p . In practice, strong fairness does not exist, and a scheduler may repeatedly schedule thread p only when $b[k]$ is *true*, which induces starvation of thread p . We therefore only assume weak fairness (introduced as justice in Lehmann et al²⁸), which means all executions are *weakly fair*, ie, if a step is continuously enabled, it is executed infinitely often. Indeed, both scenarios fail under the assumption of *weak fairness*, because $b[k]$ does not remain *false* continuously.

It must be allowed, however, that a thread remains indefinitely at line 21, ie, never attempts entry or finishes before other threads. We shall therefore not impose weak fairness for step 21 or on the abstract steps 33, 38, and 39. All other steps are subject to weak fairness and are called the forward steps of the algorithm. A *forward step* is a transition that is eventually taken at some future time, if it is continuously enabled from this time onward.

A.3 Formalization

The *state* of the system is given by the values of all shared and private variables. Up to this point, we have kept the state implicit, but formally every invariant is a boolean function of the state (or equivalently a set of states). Let X be the set of all states, and X_0 the set of states that satisfy all invariants. If P is a predicate (on the state), it can also be regarded as the subset of X where predicate P holds. Therefore, $P \subseteq Q$ means that every state that satisfies P also satisfies Q (ie, that $P \Rightarrow Q$).

For thread p , let relation $step(p)$ be the set of the pairs (x, y) of states such that in state x thread p can do a step of the algorithm that results in state y . Relation $step$ is defined as the union of the relations $step(p)$, together with the identity relation of the state space. The identity relation is added in order to allow executions in which all threads remain idle from some time onward. An *execution* is defined to be an infinite sequence xs of states in X_0 such that $(xs_n, xs_{n+1}) \in step$ for all n .

The relation $\text{fwd}(p) \subseteq \text{step}(p)$ defines the set of forward steps of thread p . Because steps 21, 33, 38, and 39 are excluded, this leaves steps 22 to 32, 34 to 37, and 40 to 42. Thread p is said to be *enabled* in state x if there is a state y with $(x, y) \in \text{fwd}(p)$, and otherwise *disabled*. Relation $\text{ena}(p)$ defines the set of states in which p is enabled, giving

$$\text{ena}(p) \equiv p \text{ in } \{22 \dots 30, 32, 34 \dots 37, 40 \dots 42\} \vee (p \text{ at } 31 \wedge (y \neq p \vee \neg b[k.p])).$$

where $k.p$ stands for the value of the private variable k of thread p . Line 31 is a special case: it is the only **await** statement of the algorithm and $y \neq p \vee \neg b[k.p]$ is its enabling condition. Note that p is disabled at 21, 33, 38, 39, as the steps at these lines are not forward steps.

An *occurrence* of thread p in execution x is a number n with $(x_{s_n}, x_{s_{n+1}}) \in \text{fwd}(p)$ or $x_{s_n} \notin \text{ena}(p)$. As stated, weak fairness is assumed for every relation $\text{fwd}(p)$, and therefore, we only consider executions xs for which every thread p has infinitely many occurrences in xs . In other words, thread p is infinitely often executed or found to be disabled, eg, when thread p remains at line 21 forever.

While executions and weak fairness are introduced to establish a framework for progress assertions, it is possible and more convenient to prove progress assertions directly from the states and steps of the algorithm using a version of UNITY logic.²⁷

A.4 Leads-to and its proof rules

Before being able to claim and prove starvation freedom for the Triangle algorithm using UNITY, it is necessary to cover a number of UNITY's basic concepts.

UNITY logic^{27,29} focuses on the concept of "leads-to" ($\mapsto$). The informal meaning of $P \mapsto Q$ is "if P holds at any point in the computation, Q will hold eventually" (here "will" applies to the current point as well as the future). The formal definition of $P \mapsto Q$ is that, for every weakly fair execution xs and every index i with $x_{s_i} \in P$, there is an index $n \geq i$ with $x_{s_n} \in Q$.

Example 1. Starvation freedom of our MX algorithm means a thread, say p , cannot remain indefinitely in the *Entry* protocol, where *Entry* is the set of the line numbers 22 ... 33, 38, 39, but it always will return to the idle state at line 21. This requirement is expressed by $\text{pin Entry} \mapsto p \text{ at } 21$.

UNITY logic begins with defining the general relation $P \text{ co! } Q$ to mean that $x \in P$ implies $y \in Q$ for every step $(x, y) \in \text{step}$

$$P \text{ co } Q \equiv \forall (x, y) \in \text{step} : x \in P \Rightarrow y \in Q.$$

Next to relation **co**, there is the specific version $P \text{ co! } Q$ to express that there is a thread r , enabled throughout P , such that $x \in P$ implies $y \in Q$ for every pair $(x, y) \in \text{fwd}(r)$

$$P \text{ co! } Q \equiv \exists r : P \subseteq \text{ena}(r) \wedge (\forall (x, y) \in \text{fwd}(r) : x \in P \Rightarrow y \in Q).$$

According to **co!**, there is a specific thread r able to establish Q from P ; thread r is called the *responsible thread*. Summarizing, **co** means that if P holds at the beginning of a step, then Q will hold at the end; **co!** strengthens this condition by specifying that there is some enabled thread r that can always make the step. The operator **co** was introduced by Misra.²⁹ The formulation of UNITY used here is also used in Hesselink.³⁰

UNITY logic is based on the relations **unless** and **ensures** defined by:

$$\begin{aligned} P \text{ unless } Q &\equiv (P \wedge \neg Q \wedge X_0) \text{ co } (P \vee Q), \\ P \text{ ensures } Q &\equiv (P \text{ unless } Q) \wedge ((P \wedge \neg Q \wedge X_0) \text{ co! } Q). \end{aligned}$$

$P \text{ unless } Q$ means that if P ceases to be *true* over a step, then Q becomes *true*. $P \text{ ensures } Q$ is a stronger condition that guarantees Q will hold.

UNITY's *leads-to* relation $\mapsto$ is defined inductively by the three rules^{27, p. 52}:

- $P \text{ ensures } Q$ implies $P \mapsto Q$.
- Relation $\mapsto$ is transitive.
- If I is a set of indices and, for every $i \in I$, P_i is a predicate with $P_i \mapsto Q$, then $(\exists i \in I : P_i) \mapsto Q$.

The first rule is called the **ensures** rule. When we apply the **ensures** rule, we often indicate the thread r responsible for the corresponding relation **co!**. The second rule is called transitivity and it connects **ensures** rules. For example, consider the loop: **while** $t < 7$ **do** $t \leftarrow t + 1$ **endwhile**. Here $t = 1$ **ensures** $t = 2$ and $t = 2$ **ensures** $t = 3$. But $t = 1$ **ensures** $t = 3$ is invalid. Instead: $t = 1 \mapsto t = 3$ holds by transitivity. The third rule is called the disjunction rule.

While Chandy et al.^{27, p. 50–52} postulates the three UNITY rules for the *leads-to* relation, they are not proven. Here, we prove the **ensures** rule, and the proofs for transitivity and disjunction are similar. The **ensures** rule is justified by the following:

Lemma 1.^{27, p. 50} Assume that $P \text{ ensures } Q$. Then $P \mapsto Q$.

Proof. Let xs be an execution weakly fair for all threads, with $x_{s_i} \in P$. The aim is to show that $x_{s_n} \in Q$ holds for some $n \geq i$. This is done by contradiction.

Assume therefore that $x_{s_n} \notin Q$ holds for all $n \geq i$. As xs is an execution, $x_{s_n} \in X_0$ holds for all n . It follows that $x_{s_n} \in \neg Q \wedge X_0$ for all $n \geq i$. As $x_{s_i} \in P$, it follows from $P \text{ unless } Q$ by mathematical induction that $x_{s_n} \in P \wedge \neg Q \wedge X_0$ holds for all $n \geq i$. By the assumption $P \text{ ensures } Q$, there is a thread r enabled throughout $P \wedge \neg Q \wedge X_0$ with the property that $x \in P \wedge \neg Q \wedge X_0$ implies $y \in Q$ for every pair $(x, y) \in \text{fwd}(r)$. Thread r is therefore enabled in x_{s_n} for all $n \geq i$. As the execution is weakly fair for r , it follows that the step $\text{fwd}(r)$ is eventually taken, say $(x_{s_k}, x_{s_{k+1}}) \in \text{fwd}(r)$ for some $k \geq i$. Then, we have $x_{s_n} \in Q$ for $n = k + 1$. Contradiction. $\square$

We need two more rules of UNITY.^{27, p. 65} The PSP (progress-safety-progress) rule combines a leads-to rule with a **unless** rule:

$$\text{PSP: } (P \mapsto Q) \wedge (A \text{ unless } M) \Rightarrow P \wedge A \mapsto (Q \wedge A) \vee M.$$

Example 2. Starvation freedom of our algorithm is formalized in $\text{true} \mapsto p \text{ at } 21$. We can use the PSP rule to derive from this the conclusion

$$p \text{ in Entry} \mapsto p \text{ in CS}$$

First observe that the code of Figure 5 implies that $p \text{ in Entry unless } p \text{ in CS}$ holds. Second, application of the PSP rule gives the conclusion because $\text{true} \wedge p \text{ in Entry}$ is the same as $p \text{ in Entry}$, and $p \text{ at } 21 \wedge p \text{ in Entry}$ equals *false*.

UNITY also has a completeness rule which states that, for predicates P and R , and a finite family of predicates $(Q_i)_{i \in I}$, we have

$$\text{CMP: } (\forall i \in I: (P \mapsto Q_i) \wedge (Q_i \text{ unless } R)) \Rightarrow P \mapsto (\bigcap_{i \in I} Q_i) \vee R.$$

The completeness rule allows combining a set of conditions Q_i , saying that if P leads-to each of the Q_i , and $Q_i \text{ unless } R$ holds for all of them, then P leads-to either all of the Q_i or R .

For our algorithm, we formalize the requirements that entrySlow at line 38 and entryBinary at lines 33 and 39 are both starvation free, by imposing the axioms:

$$\text{Amx: } p \text{ at } \ell \mapsto p \text{ at } \ell + 1 \text{ for each } \ell \in \{33, 38, 39\}.$$

By postulating these axioms, we restrict the attention to the weakly fair executions that also satisfy these axioms: an execution xs satisfies the axiom $P \mapsto Q$ if, for every index i with $xs_i \in P$, there is an index $n \geq i$ with $xs_n \in Q$.

A.5 Progress of the algorithm

After these preparations, we are ready to claim and prove starvation freedom of the algorithm in the following form:

Theorem 1. *The protocol always terminates in the sense that $\text{true} \mapsto p \text{ at } 21$.*

Proof. Note that

$$p \text{ at } \ell \mapsto p \text{ at } \ell + 1 \text{ for } \ell \in \{23, 24, 27, 29, \mathbf{33}, 34, 35, 36, \mathbf{38}, \mathbf{39}, 40, 41\}.$$

Locations 23, 24, 27, 29, 34, 35, 36, 33, 40, 41 are sequential steps $(\ell, \ell + 1)$ and follow from the **ensures** rule with responsible thread p . Locations **33, 38, 39** are from S and B and are assumed starvation free, as expressed in the axioms Amx . The remaining locations have branching steps or jumps. Here, the ensures rule implies

$$\begin{aligned} p \text{ at } 22 &\mapsto p \text{ in } \{23, 38\}, \\ p \text{ at } 25 &\mapsto p \text{ in } \{26, 27\}, \\ p \text{ at } 26 &\mapsto p \text{ at } 38, \\ p \text{ at } 28 &\mapsto p \text{ in } \{29, 33\}, \\ p \text{ at } 32 &\mapsto p \text{ in } \{33, 38\}, \\ p \text{ at } 37 &\mapsto p \text{ at } 21, \\ p \text{ at } 42 &\mapsto p \text{ at } 21. \end{aligned}$$

By the UNITY rules for transitivity and disjunction, it follows that

$$(0) \quad p \text{ in } \{32 \dots 42\} \mapsto p \text{ at } 21.$$

That is, 33 to 37 go to 21, 38 to 42 go to 21, and 32 goes to 38, which goes to 21. For the fast track, the branching at the lines 22, 26, and 28 gives the more complicated relation

$$(1) \quad p \text{ in } \{22 \dots 29\} \mapsto p \text{ in } \{30, 33, 38\}.$$

That is, 22 and 26 go to 38, and 28 goes to 30 or 33 depending on the value of the condition.

The main problem with progress is at the loop of the lines 30, 31. In order to treat this loop, we first show, for every thread q , that $b[q]$ always becomes *false* again. At line 23, however, thread q cannot be stopped from making $b[q]$ *true*. We therefore define predicate

$$B(q) \equiv q \text{ at } 23 \vee b[q],$$

and the aim is to prove that $B(q)$ always becomes *false* again. The invariant $Kq0$ implies that $B(q)$ only holds in the regions 23 to 29 and 33 to 37. Formulas (0) and (1) easily show that

$$(2) \quad \text{true} \mapsto \neg B(q).$$

That is, eventually q must set its $B(q)$ to *false* at lines 26, 29, or 37. As $B(q)$ only becomes *true* when thread q passes line 22, we have

$$(3) \quad \neg B(q) \text{ unless } y = N.$$

By the *CMP* rule, formulas (2) and (3) combine to

$$(4) \quad \text{true} \mapsto (\forall q : \neg B(q)) \vee y = N.$$

That is, eventually either all threads have *false* $B(q)$, or $y = N$. Formula (4) implies

$$(5) \quad \text{true} \mapsto (\forall q : \neg B(q)) \vee y \neq p.$$

We now focus on the lines 30, 31. It is easy to see that

$$(6) \quad p \text{ in } \{30, 31\} \text{ unless } p \text{ at } 32.$$

The *PSP* rule applied to formulas (5) and (6) gives

$$(7) \quad \begin{aligned} p \text{ in } \{30, 31\} &\mapsto P_{loop}(p), \text{ where} \\ P_{loop}(p) &= ((\forall q : \neg B(q)) \vee y \neq p) \wedge p \text{ in } \{30, 31\} \vee (p \text{ at } 32). \end{aligned}$$

Termination of the loop of thread p at the lines 30, 31 is proved by showing that the set $li.p$ becomes smaller. More precisely, we use the variant function

$$vf(p) = \begin{cases} 2 \cdot \#li.p + 1 & p \text{ in } 30 \\ 2 \cdot \#li.p & p \text{ in } 31 \\ 0 & \text{otherwise} \end{cases}$$

where $\#li.p$ is the number of elements of $li.p$. To use **ensures**, which requires atomic steps, vf needs to decrease both in steps 30 and 31; otherwise, these lines must be treated separately, which is more complex. Therefore, vf is not based directly on the reduction of $li.p$ at line 31. It counts the number of atomic steps of the loop, which is $2 \cdot N + 1$ because initially $li.p$ has N elements, every loop body takes two steps, and there is the terminating step from 30 to 32.

Now the invariant $Kq3$ is used. Predicate $P_{loop}(p)$ implies that step 31 is enabled when p is at line 31. Moreover $P_{loop}(p)$ remains valid unless p leaves line 32. We therefore have (for all $k \geq 0$):

$$vf(p) \leq k + 1 \wedge P_{loop}(p) \text{ ensures } vf(p) \leq k \wedge P_{loop}(p).$$

with responsible thread p .

Using induction and the *UNITY* rules, we obtain

$$(8) \quad vf(p) \leq k \wedge P_{loop}(p) \mapsto p \text{ at } 32.$$

because $vf(p) \leq 0 \wedge P_{loop}(p)$ implies p at 32.

Formula (8) can be combined with formula (7) to give

$$(9) \quad p \text{ in } \{30, 31\} \mapsto p \text{ at } 32.$$

By combining formulas (9), (0), and (1), we finally obtain termination:

$$\text{true} \mapsto p \text{ at } 21.$$

This concludes the proof of the theorem. □

A. 6 The fast track survives

There is one remaining proof obligation, viz., that the door to the fast track does not get closed forever. More concretely, we want to prove that $y = N$ holds infinitely often.

Theorem 2. $\text{true} \mapsto y = N$.

Proof. The idea is to prove that while $y \neq N$ holds, the set of threads in the fast track F

$$\text{SinF} = \{q \mid q \text{ in } \{27 \dots 36\}\}$$

does not grow and eventually becomes empty, unless $y = N$ has been made *true*. If the set SinF is empty, the invariant $Kq1$ implies $y = N$.

We first verify for the induction variable $k \in \mathbb{N}$, that

$$p \in \text{SinF} \wedge \#\text{SinF} \leq k + 1 \text{ unless } \#\text{SinF} \leq k \vee y = N.$$

While $y \neq N$ holds, no thread can enter the set $SinF$ from line 22. Therefore, if thread p leaves the set, the set becomes smaller. Theorem 1 implies that $true \mapsto p \notin SinF$. The PSP rule can therefore be applied to the left-hand side

$$(true \mapsto p \notin SinF) \wedge (p \in SinF \wedge \#SinF \leq k + 1) \text{ unless } (\#SinF \leq k \vee y = N)$$

where the parenthesized expressions correspond to P , Q , A and M in the PSP rule. The PSP rule applied to the left-hand side above gives the following right-hand side

$$(true) \wedge (p \in SinF \wedge \#SinF \leq k + 1) \mapsto (p \notin SinF) \wedge (p \in SinF \wedge \#SinF \leq k + 1) \vee (\#SinF \leq k \vee y = N)$$

and the expression $(p \notin SinF) \wedge (p \in SinF)$ is false, leaving

$$(true \wedge A \mapsto false \vee M) \equiv (A \mapsto M)$$

and re-substituting gives

$$p \in SinF \wedge \#SinF \leq k + 1 \mapsto \#SinF \leq k \vee y = N.$$

This holds for every thread p . Therefore, the disjunction rule gives

$$(10) \quad (\exists p : p \in SinF \wedge \#SinF \leq k + 1) \mapsto \#SinF \leq k \vee y = N.$$

If $\#SinF = k + 1$, there exists $p \in SinF$; otherwise, we have $\#SinF \leq k$, giving

$$(11) \quad \#SinF \leq k + 1 \equiv (\exists p : p \in SinF \wedge \#SinF \leq k + 1) \vee (\#SinF \leq k).$$

Trivially,

$$(12) \quad \#SinF \leq k \mapsto \#SinF \leq k \vee y = N,$$

thus another application of the disjunction rule on Formulas (10) and (12) using Formula 11 gives

$$\#SinF \leq k + 1 \mapsto \#SinF \leq k \vee y = N.$$

Using the transitivity rule, induction over k , and the observation that emptiness of $SinF$ implies $y = N$ by Kq1 and Kq2, we obtain

$$\#SinF \leq k \mapsto y = N.$$

Finally, $\#SinF \leq N$ holds, because every set of threads has at most N elements. Taking $k := N$, this proves the theorem. $\square$