MCSH, a lock with the standard interface

Hesselink, W H; Buhr, Peter

Published in:
ACM Transactions on Parallel Computing

DOI:
10.1145/3584696

MCSH, a Lock with the Standard Interface

WIM H. HESSELINK, University of Groningen, The Netherlands
PETER A. BUHR, University of Waterloo, Canada

The MCS lock of Mellor-Crummey and Scott (1991) is a very efficient first-come first-served mutual-exclusion algorithm that uses the atomic hardware primitives fetch-and-store and compare-and-swap. However, it has the disadvantage that the calling thread must provide a pointer to an allocated record. This additional parameter violates the standard locking interface, which has only the lock as a parameter. Hence, it is impossible to switch to MCS without editing and recompiling an application that uses locks.

This article provides a variation of MCS with the standard interface, which remains FCFS, called MCSH. One key ingredient is to *stack* allocate the necessary record in the *acquire* procedure of the lock, so its life-time only spans the delay to enter a critical section. A second key ingredient is communicating the allocated record between the *acquire* and *release* procedures through the lock to maintain the standard locking interface. Both of these practices are known to practitioners, but our solution combines them in a unique way. Furthermore, when these practices are used in prior papers, their correctness is often argued informally. The correctness of MCSH is verified rigorously with the proof assistant PVS, and experiments are run to compare its performance with MCS and similar locks.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; **Concurrent algorithms;** • **Theory of computation** → **Parallel computing models**;

Additional Key Words and Phrases: Concurrency, critical section, mutual exclusion, atomicity, efficiency

ACM Reference format:
Wim H. Hesselink and Peter A. Buhr. 2023. MCSH, a Lock with the Standard Interface. *ACM Trans. Parallel Comput.* 10, 2, Article 11 (June 2023), 23 pages.
https://doi.org/10.1145/3584696

1 INTRODUCTION

The mutual exclusion problem was introduced by Dijkstra in 1965 [7, 8]. It can be phrased as follows. There are several concurrent processes or threads that communicate by shared variables and from time to time need exclusive access to shared resources. A shared resource and code manipulating it form a pairing called a **critical section (CS)**, which is a many-to-one relationship; e.g., if multiple files are being written to by multiple threads, then only the pairings of simultaneous writes to the same files are CSs. Regions of code where the thread is not interested in the resource are combined into the **non-critical section (NCS)**. Exclusive access to a resource is provided by

Peter Buhr is funded by the Natural Sciences and Engineering Research Council of Canada.
Authors' addresses: W. H. Hesselink, University of Groningen, Groningen, P.O. Box 407 9700 AK, The Netherlands; email: w.h.hesselink@rug.nl; P. A. Buhr, University of Waterloo, 200 University Ave. West, Waterloo, Ontario, Canada; email: pabuhr@uwaterloo.ca.

2329-4949/2023/06-ART11 $15.00
https://doi.org/10.1145/3584696

mutual exclusion (MX). MX is implemented by some form of *lock*, where the CS is bracketed by lock procedures *acquire* and *release*. Every thread thus executes an unbounded loop of the form

```
(0)   loop of thread p:
          NCS;
          acquire(lock); CS; release(lock)   ← MX/standard interface
      end loop.
```

MX guarantees there is never more than one thread in the CS. MX must also guarantee eventual progress: When there are competing threads, eventually some competing thread succeeds, i.e., acquires the CS, releases it and returns to the NCS. A stronger constraint is that every thread that calls *acquire* eventually succeeds.

1.1 Standard Lock API

The standard lock interface is defined in Reference [21, pp. 58–59] (also called a *context-free* interface by Wang et al. [22, Section 1.1]) as a lock type for declaring instances, and procedures *acquire* and *release* for locking and unlocking, which take a reference to the lock as the *only* parameter. The lock reference can be passed implicitly as an object-oriented receiver or explicitly as an argument, i.e., *lock.acquire()* or *acquire(lock)*. The single parameter lock interface defines an **Application Program Interface (API)** [24] that programmers use to access different lock implementations without changing code, possibly by just relinking to a different lock library.

There are a number of techniques for converting a non-standard interface into a standard interface [21, Figures 4.10 and 4.14]. All these techniques indirectly pass additional information to the lock procedures or between them. Some of these techniques use only basic programming-language mechanisms and, hence, do not rely on the program's **Application Binary Interface (ABI)** [23] for correctness. For example, a lock procedure may create local variables in its activation record for computing or storing temporary results. However, the duration of such local variables does not exceed the call.

Lock implementations that rely on an ABI, even with a standard interface, may fail to compile or work incorrectly when linked with different binary program modules. For example, the original CLH lock does not have a standard interface and requires global memory-space per lock [17, p. 168].[1] Scott [21, Figure 14] presents a CLH with a standard interface but requires an array of size T, where T is the maximum number of executing threads in the program, e.g., a program may create at most 1,000 threads, but often C, the number of contending threads, is $\ll T$. Since C varies with workload, programmers often fall back on a large worst-case T (1000 threads), which is the maximum number of threads created by all workloads. Generalizing this storage problem requires dynamic allocation per thread or **thread-local storage (TLS)** [25] to support an arbitrary number of threads.

In many lock proofs, it is assumed an acquiring or releasing thread is not interrupted and diverted to other work. This assumption is violated when the lock is linked with a binary module that raises signals (interrupts). For example, in CLH acquire [17, Figure 2]:

```
lh_acquire( int **L, **I, **P ) {
  **I = 1;
  atomic{ *P = *L; *L = *I; } /* fetch-and-store */
  while ( **P! = 0 ) { }; /* spin */
}
```

[1] Scott's version [21, Figure 12] is a non-standard interface, because there are two parameters: implicit lock (receiver) and node pointer.

an interrupt after the atomic operation causes a failure if the signal handler recursively calls lh_acquire. The failure occurs because the second call relinks the link node onto the front of the queue from its prior position in the queue. Hence its new P link is no longer pointing to its prior queue position and the queue structure is broken. This failure is an instance of the serially reusable problem [13] and occurs because the link node is associated with the thread rather than the lock.[2] In contrast, MCS assumes each link node is associated with the lock, so a recursive call brings a new node to link into the queue. Thus, for allocation approaches using thread specific memory, like the preallocated array, dynamic allocation per thread, or TLS, there is potential to violate memory assumptions for a lock's proof. This problem also occurs in using user-level threading with time-slicing, because user threads move across kernel threads at arbitrary times accessing different TLS. Hence, using a lock with a standard interface might fail, because its implementation can have underlying ABI issues.

In general, ABI conversions for converting a lock to the standard interface fall into the following categories.

ABI Preserving Conversion: A preserving technique is inserting a *message* variable, `msg`, in the lock, which is written in procedure *acquire* and read in procedure *release*.

(1)
```
type Lock =
    record
        ...;
        Node msg;
    end .

Lock lock.

loop of thread p:
    NCS;
    acquire(lock) { Node n_p; ... ; lock → msg = n };
    CS;
    release(lock) { ... = lock → msg; ... }
end loop.
```

Any information may be copied through the lock [5]. This transformation only uses basic programming-language mechanisms, e.g., stack declarations and simple control structures, and does not rely on advanced language features (e.g., closures that change the calling convention) or side-effects in the programming environment.

The correctness of the converted systems, however, is not completely obvious, because its mutual exclusion relies on the assumption that the sender of the message is also the receiver, and the proof of the latter assumption needs mutual exclusion.

ABI Non-Preserving Conversion: A non-preserving technique is using global or thread variables, which are manipulated in procedures *acquire* and *release*.

(2)
```
thread-local Node n_p;
acquire(lock);    ← n_p used implicit
release(lock);    ← n_p used implicit
```

However, this transformation relies on the programming environment to implement the TLS, which may have side effects for lock usage, as previously noted. Other

[2] In Fortran IV, a procedure had a single preallocated frame for use during a call, so recursive calls were disallowed.

non-preserving ABI conversions exist, e.g., dynamic allocation or fixed sized arrays, as discussed above.

Unfortunately, the correctness of such a conversion is seldom verified. Moreover, issues of serial reusability are usually ignored.

1.2 Contributions

This work selects the well-known, highly used, **first-come first-served (FCFS)** MCS-lock [18, Section 2.4], which does not have a standard interface, and transforms it using only basic programming-language mechanisms into a new FCFS variant, MCSH, with a standard interface. The transformation is formalized, and its correctness is proved. Finally, performance experiments are run to compare MCSH with variants of MCS and similar locks.

Section 2 presents MCS, and Section 3 then develops MCSH. Section 4 gives a brief introduction to the formal treatment of concurrent algorithms. Section 5 proves the correctness of MCSH, where some details of this proof are moved to the Appendix. Section 6 compares MCSH with nine mutual-exclusion locks with different locking properties. Section 7 concludes and suggests future work.

2 MCS ALGORITHM

The MCS algorithm of Mellor-Crummey and Scott [18] is called a hardware lock, because it uses atomic hardware instructions **fetch-and-store (FAS)** and **compare-and-swap (CAS)**. For performance, it is among the best mutual exclusion algorithms available, see Buhr et al. [3, Sections 20, 22]. The good performance is due to its local spinning property and decent **remote memory reference (RMR)** complexity [21, Section 4.5.1].

The classical MCS algorithm of Reference [18] is given in Figure 1 (see also Reference [21, Figure 4.8]).[3] It builds a queue of threads waiting for the CS as a linked list with pointers to records of type *qnode*. The null pointer is denoted by $\bot$. If the queue is nonempty, then the shared variable `lock` holds the tail; otherwise, `lock` $= \bot$.

MCS usage has every thread p pass in a pointer to an allocated record n_p, which is used in both procedures *acquire* and *release*.

(3)
```
qnode* lock := ⊥.    ← MCS lock
loop of thread p:
    NCS;
    qnode n_p;              ← allocated record
    acquire(&lock, &n_p); CS; release(&lock, &n_p)
end loop.
```

where n_p must not be used for other locking activities during the *acquire-release* tenure. The lifespan of n_p is the same or exceeds the *acquire-release* interval.

In Figure 1, the line with FAS is equivalent to the atomic command

$$\langle\ prev_p := \texttt{lock}\ \ \texttt{lock} := my_p\ \rangle$$

and the expression with CAS is equivalent to the atomic function

$$\langle\ \textbf{if}\ \texttt{lock} = my_p\ \textbf{then}\ \texttt{lock} := \bot;\ \textbf{return}\ \textit{true}\ \ \textbf{else return}\ \textit{false}\ \textbf{endif}\ \rangle.$$

[3] All example pseudo-code assumes read/write atomicity and sequential consistency.

```
type qnode =
    record
        qnode* next ;
        boolean locked ;
    end .
qnode* lock := ⊥ .
acquire(qnode** lock, qnode* my_p) is
    my_p → next := ⊥ ;
    qnode* prev_p := FAS(lock, my_p) ;        ← activation record variable
    if prev_p ≠ ⊥ then
        my_p → locked := true ;
        prev_p → next := my_p ;
        await (¬ my_p → locked)
    endif ;
end acquire .
release(qnode** lock, qnode* my_p) is
    if my_p → next = ⊥ then
        if CAS(lock, my_p, ⊥) then return endif ;
        await (my_p → next ≠ ⊥) ;
    endif ;
    my_p → next → locked := false
end release .
```

Fig. 1. The classical version of MCS.

The first argument of FAS and CAS is the address of `lock`, because `lock` is modified. In either case, atomicity of the instruction means that there is no interference by other processes between the two references to `lock`.

For a Boolean expression B, the command **await**(B) is equivalent to a busy-waiting loop

while $\neg B$ **do** *skip* **endwhile**.

MCS has the disadvantage that each thread p must have an allocated record n_p and pass it to the two locking procedures. In terms of Section 1.1, the algorithm does not satisfy the standard lock API. Auslander et al. [2] devised an alternative that fits the standard interface as part of the K42 project at IBM Research, see Reference [21, Figure 4.10]. This proposal is no longer FIFO, has the (theoretical) possibility of starvation and worse RMR properties than normal MCS. Similarly, Wang et al. [22] created MCSg with a standard interface but not FIFO and with starvation. K42 and MCSg use special properties of the lock and its fields to remove thread-specific data.

3 MCSH ALGORITHM

We propose a different alternative with a standard interface given in Figure 2, which is obtained from MCS by two ABI preserving conversions. First, the thread holding the lock does not need its queue node. Therefore, the lifetime of the node can be reduced to the waiting duration in acquire allowing the node to be stack allocated. Second, the lost queue data for the thread holding the lock is preserved within the lock. Therefore, the release procedure can find the next node to reset its spinning flag. A consequence of this approach is to simplify the release procedure by moving most of the MCS release code to acquire.

```
type Lock =
    record
        qnode* tail := ⊥ ;
        qnode* msg ;
        boolean flag := true ;
    end .
Lock lock .
acquire(Lock* lock) is
    qnode mm_p := {⊥, false} ;                    ← activation record variable
    qnode* prev_p := FAS(&lock → tail, &mm_p) ;   ← activation record variable
    if prev_p = ⊥ then
        await (lock → flag) ;
    else
        prev_p → next := &mm_p ;
        await (¬ mm_p.locked)
    endif ;
    lock → flag := false ;
    qnode* succ_p := mm_p.next ;
    if succ_p = ⊥ then
        if ¬ CAS(&lock → tail, &mm_p, ⊥) then
            await (mm_p.next ≠ ⊥) ;
            succ_p := mm_p.next
        endif
    endif ;
    lock → msg := succ_p ;
end acquire .
release(Lock* lock) is
    qnode* succ_p := lock → msg ;                 ← activation record variable
    Fence() ;   ← prevent hardware interchange
    lock → flag := true ;
    if succ_p ≠ ⊥ then
        succ_p → locked := false
    endif
end release .
```

Fig. 2. The new MCSH lock.

In detail, the MCSH algorithm works as follows. The global variable n_p becomes the local variable mm_p of procedure *acquire*. It is, therefore, automatically (stack) allocated when *acquire* is called. The address *&amp*; mm_p serves as the pointer my_p. As the local memory of *acquire* is destroyed after the call and is not available to *release*, some of the activity in *release* related to my_p must be transferred to *acquire*. A consequence of this change is that the call of CAS by thread p can set `tail` $:= \bot$, before thread p has entered CS.

Therefore, a shared Boolean variable `flag` is placed in the lock to prevent a new thread q finding $prev_q = \bot$ and entering CS concurrently with p. As well, the variable `msg` is placed in the lock to transfer the value of $succ_p$ from *acquire* to *release*, giving a standard-lock interface. These two changes fall into the category of ABI Preserving Conversions that use only basic programming-language mechanisms and no static or thread-local storage.

Remarks. Procedure *release* has a local variable *succ* with the same role (and value) as *succ* in *acquire*.

The variable `flag` has at most one thread busy-waiting on it at any given time, giving the desirable local-waiting property.

The first two lines of *release* must not be swapped (see hardware Fence), because if `flag` holds before `msg` is read, then a new thread may be able to modify `msg`.

4 FORMALIZATION

Section 4.1 discusses how to formalize concurrent algorithms as threaded machines. Section 4.2 gives some theory of invariants for concurrent algorithms.

4.1 Machines and Threaded Machines

A *machine* or *state machine* is a tuple $K = (X, X_0, N)$, where X is a set, X_0 is a subset of X, and N is a reflexive binary relation on X. The elements of X are called *states*, where X_0 is the *initial condition*, and N is the *next-state relation*.

An *execution* of machine K is a state sequence in X that begins in an initial state and in which every pair of subsequent states satisfies the next-state relation. Formally, it is a function $xs : \mathbb{N} \to X$ such that $xs(0) \in X_0$ and that $(xs(n), xs(n+1)) \in N$ for all $n \in \mathbb{N}$.

A *predicate* is a Boolean function on the state space X. A predicate P can also be regarded as the subset of X where P holds. A predicate is called an *invariant* of machine K iff it contains all states of all executions of K.

A *threaded* machine has a set T of thread identifiers (natural numbers). Its next-state relation is a union $N = \mathbf{1}_X \cup \bigcup_{p \in T} N_p$, where $\mathbf{1}_X$ is the equality relation on X, and N_p is a next-state relation for thread p. The elements of N_p are regarded as steps that thread p can perform. So, apart from allowing a skip statement in $\mathbf{1}_X$, every step is done by some thread. The skip step is needed to make relation N reflexive.

Remarks. Abadi and Lamport [1] defined a *specification* to be a state machine with a *supplementary property*. The executions of the machine that satisfy the property are called *behaviours* of the specification. The property is commonly used for liveness conditions. For the present purposes, the property can be ignored.

4.2 Invariants

This starts with a theory review concerning invariants and a method for obtaining and proving them.

Let a subrelation of the next-state relation N be called a *command*. For a command S and predicates P and Q, the *Hoare triple* $\{P\}S\{Q\}$ is the proposition that

$$\forall\, x, y : (x, y) \in S \;\wedge\; x \in P \;\Rightarrow\; y \in Q, \text{ or equivalently}$$
$$[P \;\Rightarrow\; \mathbf{wp}(S, Q)],$$

where **wp** stands for Dijkstra's weakest precondition [9, p. 16].

A predicate P is said to be *preserved* by command S iff $\{P\}S\{P\}$. Predicate P is called *stable* if it is preserved by N. A predicate is called *inductive* iff it is stable and holds initially. Every inductive predicate is an invariant.

A predicate P is said to be *threatened* by a command S iff it is not preserved by S. If predicate P is threatened by command S, then a predicate Q is called a *remedy* for P and S iff $\{P \wedge Q\}S\{P\}$.

Let C be a set of commands such that $N = \mathbf{1}_X \cup \bigcup C$. A family of predicates is called *complete* if any member of the family that is threatened by any command in C has some remedy consisting of members of the family. The conjunction of a complete family is stable. The family is said to

```
initially:
    flag = true  ∧  tail = ⊥  ∧  local = {⊥}
    ∧   low = high = 1
    ∧   ∀ q ∈ thread : pc_q = 11  ∧  slot_q = 0 .
loop of thread p :
11      NCS;
        choose my_p ∉ local ; add my_p to local ;
12      next(my_p) := ⊥ ;
13      locked(my_p) := true ;
14      prev_p := tail ; tail := my_p ;
        slot_p := high ; high++ ;
15      if prev_p = ⊥ then
16          await (flag) ;
        else
17          next(prev_p) := my_p ;
18          await (¬ locked(my_p))
        endif ;
19      flag := false ;
20      nxmy_p := succ_p := next(my_p) ;
21      if succ_p = ⊥ then
22          if tail = my_p then tail := ⊥
            else
23              await (next(my_p) ≠ ⊥) ;
24              nxmy_p := succ_p := next(my_p)
            endif
        endif ;
25      msg := succ_p ;
26      succ_p := ⊥ ; remove my_p from local ;
27      CS;
28      succ_p := msg ;
29      flag := true ;
        if nxmy_p = ⊥ then low++ endif ;
30      if succ_p ≠ ⊥ then locked(succ_p) := false ; low++ endif ;
endloop .
```

Fig. 3. State machine of the MCSH lock, with ghost variables.

be *initialized* if the initial condition implies every member of the family. The conjunction of an initialized complete family is an inductive predicate; every member of it is an invariant, because it is implied by an invariant. The family approach is the method used below to obtain and prove invariants. It is presumably well known, but it was first made explicit in Reference [12].

For this article, the proof assistant PVS [20] is used to determine and verify the threats and the remedies for the invariants. The PVS proof script for MCSH is publicly available [11]. Predicate names have the form *Xqd* to allow simple query-replacement in the PVS proof script.

5 THE CORRECTNESS OF THE LOCK MCSH

In Section 5.1, the algorithm MCSH is modelled by the transition system of Figure 3. In Section 5.2, the method of Section 4.2 is used to generate a family of invariants that proves mutual exclusion.

The family is so unwieldy that it has been transferred to an Appendix. Section 5.3 proves that the algorithm is deadlock free.

5.1 Modelling MCSH for Correctness

To verify MCSH, the procedures *acquire* and *release* are put into the loop (0), and the atomic statements are numbered. This combination gives the transition system of Figure 3.

The local variable mm_p is eliminated in favor of its address $my_p = \&mm_p$. A shared variable `local` is introduced to hold the set of allowed pointers. Initially, it only holds the nil pointer $\perp$. When thread p enters *acquire*, a new pointer $my_p \neq \perp$ is added to `local`. This pointer is removed again when the call of *acquire* terminates. This behaviour gives the proof obligation that all pointers are in `local` when referred to and are different from $\perp$ when their fields are inspected.

The type *qnode** is renamed to *pointer* and used as an index domain for two arrays `next` and `locked` declared by

`next` : **array**(*pointer*) **of** *pointer*;
`locked` : **array**(*pointer*) **of** *Boolean*;

For every allocated pointer u, the array elements $\texttt{next}(u)$ and $\texttt{locked}(u)$ stand for the fields of the record pointed to. We thus identify $\texttt{next}(u) = (u \rightarrow next)$ and $\texttt{go}(u) = (u \rightarrow go)$.

Each line number stands for one atomic command. The implicit private variable pc_q indicates the line number that thread q is to execute next. At every line, it is implicitly incremented with 1, unless a keyword **if** or **loop** indicates otherwise. The line numbers start with 11 to facilitate query-replace in the script for the mechanical theorem prover PVS. Note that line 14 is one atomic command because of the atomicity of the FAS. Similarly, line 22 is one atomic statement because of the atomicity of CAS. In line 26, $succ_p$ is reset to $\perp$ to model that thread p leaves the scope of $succ_p$.

For ease of verification, four ghost variables are introduced. Recall that a ghost variable is an auxiliary variable that does not influence the computation and is only used for the verification; atomic commands of the program can be extended with modifications of ghost variables. Here, every thread p gets a private ghost variable $slot_p$ that indicates its latest position in the waiting queue. It is initially 0 and gets a new value from the shared ghost variable `high` when thread p executes line 14. The ghost variables `low` and `high` indicate the bounds of the slots of the competing threads. A persistent private ghost variable $nxmy_p$ is introduced to express the equality of the local variables $succ_p$ of *acquire* and *release*. This variable can be used in line 29, because `low` is a ghost variable as well.

In this way, loop (1) instantiated with MCSH becomes the transition system of Figure 3. To indicate which threads are where in the execution, we use the state-dependent sets of threads:

$$[k] = \{q \mid pc_q = k\},$$
$$[j,k] = \{q \mid j \leq pc_q \leq k\}.$$

The first aim is to prove mutual exclusion. As CS is at line 27, this is expressed by the predicate

$$MX0\colon\ q \in [27]\ \wedge\ r \in [27]\ \Rightarrow\ q = r.$$

From this point onward, the predicates are given with implicit universal quantification over all free variables (here q and r).

5.2 Mutual Exclusion for MCSH

In this section, the method of Section 4.2 is used to generate enough invariants to prove mutual exclusion. The transition system has only 20 transitions. Yet more than 40 invariants are needed to prove mutual exclusion.

For line number k and thread p, let $N_{p,k}$ be the command that corresponds to execution of line k by thread p. Let C be the set of all these commands. The idea is to construct a family of predicates such that every member of it that is threatened by some command in C has a remedy in the family. In most cases, the command is indicated by the line number, while the acting thread p is kept implicit.

Before proceeding into meaningful invariants, note that by construction it always holds that $my_q \neq \bot$ and that $1 \leq \mathtt{low}$ and $1 \leq \mathtt{high}$. These obvious invariants are used implicitly.

The first claim is that the ghost variables $slot_q$ and $\mathtt{low}$ satisfy the invariants

Iq1: $slot_q = slot_r \neq 0 \;\Rightarrow\; q = r$,
Iq2: $q \in M \;\Rightarrow\; slot_q = \mathtt{low}$,

where $M = M_1 \cup M_2 \cup M_3 \cup M_4$ and

$$\begin{aligned} M_1 &= \{q \mid (q \in [15] \wedge prev_q = \bot) \vee q \in [16]) \;\wedge\; \mathtt{flag}\},\\ M_2 &= \{q \mid q \in [18] \;\wedge\; \neg\, \mathtt{locked}(my_q)\},\\ M_3 &= [19, 29],\\ M_4 &= \{q \mid q \in [30] \;\wedge\; nxmy_q \neq \bot\}. \end{aligned}$$

It is easy to see that the predicates *Iq1* and *Iq2* together imply *MX0*. In fact, they imply the much stronger assertion

MX1: $q \in M \;\wedge\; r \in M \;\Rightarrow\; q = r$.

We now take *Iq1* and *Iq2* as the founding members of an initialized complete family. This family is constructed in the following way. For each new member of the family, a list of line numbers of threatening commands is determined, and for each line number, a remedy that is a conjunction of one or more, possibly new, members. It turns out that 43 members are needed to make the family complete. All members hold initially. The complete list is given in the Appendix, as well as the threatenings and the remedies. This proves that *Iq1*, *Iq2*, *MX1*, and all other members of the family are invariants.

In particular, it shows that the waiting threads form a queue because of the invariant

Jq8: $q \in [15, 26] \;\wedge\; r \in [15, 26] \;\wedge\; \mathtt{next}(my_q) = my_r \;\Rightarrow\; slot_q + 1 = slot_r$.

By *Iq3* and *Kq5*, the slots are bounded:

$$q \in [15, 29] \;\Rightarrow\; \mathtt{low} \leq slot_q < \mathtt{high}.$$

The family also proves that the pointers are used only when they have the meaningful values:

Mq3: $\bot \in \mathtt{local}$,
Mq5: $\mathtt{tail} \in \mathtt{local}$,
Mq1: $q \in [12, 26] \;\Rightarrow\; my_q \in \mathtt{local}$,
Mq4: $q \in [15, 17] \;\Rightarrow\; prev_q \in \mathtt{local}$,
Lq4: $q \in [15, 26] \;\Rightarrow\; \mathtt{next}(my(q)) \in \mathtt{local}$,
Kq3: $q \in [21, 30] \;\Rightarrow\; nxmy_q \in \mathtt{local}$.

The equality of the variables *succ* in *acquire* and *release*, and `msg` in between, is expressed in the invariants

Iq9: $q \in [21, 26] \ \vee \ q \in [29, 30] \ \Rightarrow \ succ_q = nxmy_q$,
Jq9: $q \in [26, 28] \ \Rightarrow \ \texttt{msg} = nxmy_q$.

According to Reference [14], a mutual exclusion algorithm has the FCFS property if the procedure *acquire* is the sequential composition of two fragments: a wait-free fragment called the *Doorway* and a waiting fragment called *Waiting*, such that if thread p is in *Waiting* when thread q enters the *Doorway*, then thread q does not enter CS before p does. In MCSH, the Doorway consists of lines 12, 13, and 14. If thread p is in *Waiting* when thread q enters the *Doorway*, then thread p has obtained a slot and thread q gets $slot_q > slot_p$. These numbers do not change while p and q remain competing. According to *Iq2*, thread p is in CS when $slot_p = \texttt{low}$. As the variable `low` only increases, thread p comes into CS before q. This proves FCFS.

5.3 No Deadlock States

A thread is said to be *competing* if it is not at line 11. A state is called a *deadlock state* if there are competing threads and none of them can do a step, i.e., execute a command. As the algorithm has no internal loops, deadlock-freedom is equivalent to the absence of deadlock states. If a competing thread is not at an **await** statement, then it can do the step of its line number. We therefore concentrate on the **await** statements in the lines 16, 18, and 23.

The proof of deadlock freedom needs three invariants with an existential quantification. Several invariants of the family in the Appendix are used.

A thread remains waiting at line 16 iff `flag` is false, which only occurs with a thread in the critical section. This fact is expressed by the inductive invariant:

Nq1: $\texttt{flag} \ \vee \ \exists \, q \in [20, 29]$.

However, the invariant *Jq7* (see Appendix) implies that if q is at line 16, then there is no thread in Reference [19, 24]. Together this gives

At16: $q \in [16] \ \Rightarrow \ \texttt{flag} \ \vee \ \exists \, r \in [25, 29]$.

The set `local` of the meaningful pointers satisfies the inductive invariant

Nq2: $u \in \texttt{local} \ \Rightarrow \ u = \bot \ \vee \ (\exists \, q : u = my_q \ \wedge \ q \in [12, 26])$.

This invariant is one of the ingredients needed to prove

At23: $q \in [23, 24] \ \Rightarrow \ \exists \, r \in [12, 18]$.

Indeed, if $q \in [23, 24]$, then *Jq6* implies $\texttt{tail} \neq \bot$ and *Mq7* implies $\texttt{tail} \neq my_q$. Therefore, *Mq5* and *Nq2* imply that there is a thread $r \neq q$ with $my_r = \texttt{tail}$ and $r \in [12, 26]$. Finally, *MX1* implies $r \notin [19, 26]$, concluding the proof of *At23*.

The third invariant with an existential quantification is

$$\begin{aligned} \textit{Nq3}: \ & q \in [18] \ \wedge \ \texttt{locked}(my_q) \\ & \Rightarrow \ \ \exists \, r : \ r \in [15, 24] \ \wedge \ \texttt{next}(my_r) = my_q \\ & \qquad \vee \ r \in [25, 30] \ \wedge \ nxmy_r = my_q . \end{aligned}$$

This predicate is threatened by the commands 12, 13, 17, 21, and 22. At commands 12 and 13, it has the remedy *Lq2*.

The proof at command 17 is complicated. First assume that $p = q$ executes command 17 and goes to 18. Using *Nq2* and the new auxiliary invariant *Nq4* (see below) to find a thread $r \in [12, 26]$ with $my_r = prev_q$. The invariants *Lq7* and *Mq6* imply $r \in [15, 23]$. The command establishes $\texttt{next}(my_r) = my_q$. This proves that *Nq3* is preserved when $p = q$.

Second assume that $p \neq q$ executes command 17 and modifies next. By the prior assumption, *Nq3* holds in the precondition of the command. Nq3 is threatened only if $\texttt{next}(my_r) = my_q$ and $my_r = prev_p$. Then *Jq8* and *Lq1* imply $slot_q = slot_p > 0$. By *Iq1* this gives $q = p$, a contradiction. Predicate *Nq4* is the obvious inductive invariant

Nq4: $q \in [17] \;\Rightarrow\; prev_q \neq \bot$.

At command 21, predicate *Nq3* has the remedy

Nq5: $q \in [21] \;\wedge\; r \in [18] \;\wedge\; \texttt{next}(my_q) = my_r$
$\Rightarrow \quad nxmy_q = \bot \;\vee\; nxmy_q = my_r$.

At command 22, *Nq3* has the remedies *Iq3*, *Jq8*, *Kq2*. This proves that *Nq3* is preserved when $p \neq q$.

Predicate *Nq5* is threatened only by command 17. It has the remedies *Kq4* and *Mq8*. This completes the proof of the invariant *Nq3*, and thus the preparation of the proof of deadlock freedom.

Theorem 5.1. *Assume there are competing threads. Then some competing thread can do a step.*

Proof. Every thread that is not at an **await** statement can do a step. Therefore, assume that every thread is at one of the lines 11, 16, 18, and 23. If there is a thread at line 16, then the predicate *At16* implies that flag holds, because there are no threads in References [26, 29]. Hence, every thread at line 16 can do a step. Now, assume that every thread is at one of the lines 11, 18, and 23. As there are competing threads, there is at least one thread at line 18 or 23. The predicate *At23* now implies that there is a thread at line 18.

Now let p be the thread at line 18 with the lowest value of $slot_p$. If $\neg\, \texttt{locked}(my_p)$ holds, then thread p can do a step. Otherwise, the invariant *Nq3* implies there is a thread r with $r \in [18] \cup [23]$ and $\texttt{next}(my_r) = my_p$. The invariant *Jq8* implies that $slot_r + 1 = slot_p$. By minimality of p, it follows that thread r is not at line 18, and therefore at line 23. As $\texttt{next}(my_r) = my_p \neq \bot$, thread r can do a step. □

6 PERFORMANCE

Lock performance is dominated by *contention* among the threads. The two interesting contention points are minimal and maximal, i.e., when only one thread is using the lock or T threads simultaneously. Often a lock algorithm is designed to optimize only one of these scenarios, often with a special *fast-path* [15, Figure 2]. As contention diminishes, a lock's performance quickly approaches its minimal (uncontended) performance, because threads arrive more and more at an uncontended lock. Testing the two extremes gives a strong indicator of how a lock algorithm performs.

The overall performance experiment compares MCSH and related algorithms to demonstrate in general how MCSH compares. The goal is to show a range of performance among the algorithms. This can be used by application developers, in conjunction with other factors, as a guide for algorithm selection. No attempt is made to rank the tested algorithms because of their differences. For example, some algorithm do not have a standard interface, some are not FCFS, and some use complex ABI, like TLS. How does MCSH compare among these similar algorithms with different locking properties?

Our performance experiment attempts to eliminate confounding factors such as complex architecture designs and operating system effects, so algorithmic differences stand out. Therefore, only 1–32 threads are tested to reduce or eliminate factors related to different cache structures and NUMA effects. As well, the number of threads is one-to-one with the cores, and all threads are pinned on cores to prevent operating-system scheduling effects during an experiment.

MCSH and the following algorithms are tested for their overall performance.

MCS (see Figure 1) has each acquiring thread provide a node containing a queue link and flag. A waiting thread atomically chains its node to the end of the queue, where the lock points at the tail node. After chaining, a waiting thread spins locally on the flag in its node. The thread releasing the lock atomically checks if it is the last node (empty queue) and resets the lock pointer to null; otherwise, it spins until its link field is set by the chaining (following) thread and then uses this link to reset the flag of the chained thread.

MCSFAS [18, Figure 7] is a non-FCFS MCS lock with a modified release procedure solely to replace the CAS in the release procedure with two FAS instructions. The change requires additional logic to update the lock pointer if the queue is empty, and the spin still exists to wait for the next thread to update the releasing thread's link field so it can reset the next thread's flag. While not FCFS, the fairness results for the maximal contention experiment (see Figure 7), show MCSFAS behaves like an FCFS algorithm (other workloads can result in measurable non-FCFS behaviour).

MCSK42 [21, Figure 4.10] is a non-FCFS MCS lock with a modified acquire procedure to obtain a standard interface. Like MCSH, this algorithm uses an extra field in the lock to copy the next link from the node for the thread holding the lock, releasing this node in acquire rather than release. Scott suggests a way to make it FCFS with more complexity and "significantly poorer" performance [21, p. 59]; no actual algorithm is presented or analysed.

QSpinLock [16] is a user-space variant of the version developed for the Linux kernel. It is a FCFS lock *containing* an MCS lock and flag. Like MCSH, the MCS node is stack allocated for the duration of the acquire call. An acquiring thread first acquires the MCS lock and then spins on the lock flag. Hence, the thread spinning on the flag is the head of the FCFS queue of contending threads. In all cases, the spinning is local either on the flag in an MCS node or the lock. The releasing thread resets the lock flag; the acquiring thread stops spinning and performs an MCS release, unblocking the next contending thread to spin on the lock flag. Hence, lock release has no atomic instructions or spinning, and the next acquiring thread loads its cache with the lock flag in preparation for the flag reset.

CLH [17, Figure 2] is a FCFS MCS, where each thread spins locally on its *predecessor's* node versus its own. While CLH reduces complexity in both acquire and release (no CAS or spin), the downside is lifetime management of the link nodes (like hazard pointers for lock-free data structures [19]). Each thread must provide a node to acquire, but release returns the predecessor's node in the contended case, because the current node still has the next thread spinning on it. Hence, each thread must dynamically allocate a node for use by itself or other participating threads, and subsequently delete the last node it receives from release. Note that once a lock is acquired by a thread, its node is available for use to acquire another lock; therefore, only one node per thread is needed. Eliminating the dynamic allocation requires a riskier approach, where the node is allocated from TLS, requiring thread lifetimes to match at some level to prevent access of TLS for a terminated thread. As mentioned in Section 1.1, standard interface versions of CLH exist but require more complex storage management.

HemLock [6, Listing 1] is a compact CLH lock, without lifetime management issues. Like CLH, the lock points to the tail of the queue; however, each thread node contains just a spinning flag, and the queue link-fields are stack allocated for the duration of the acquire call. The upside is no lifetime management issues, because threads spin on their own flag; the downside is the acquire procedure spins on an atomic FAS and the release procedure spins on a flag. A standard interface version of HemLock exists using TLS storage.

Table 1. Lock Traits

traits algorithms	contended atomics	FCFS	standard interface	lock data	TLS data	release spin	contended space[a]
MCS	FAS + CAS	yes	no	no	no	yes	$O(C)$
MCSFAS	3 FAS	no	no	no	no	yes	$O(C)$
MCSK42	3 CAS	no	yes	no	no	yes	$O(C)$
MCSH	FAS + CAS	yes	yes	yes	no	no	$O(C)$
QSpinLock	FAS + CAS	yes	yes	no	no	no	$O(C)$
CLH	FAS	yes	no or TLS	yes	maybe	no	$O(T)$
HemLock	N FAS + 1 CAS	yes	no or TLS	no	maybe	yes	$O(T)$
SpinLock	∞ TAS	no	yes	no	no	no	$O(1)$
PthreadLock	N FAS + FUTEX	settable	yes	no	no	no	$O(C)$

[a] $C \Rightarrow$ contending threads, $T \Rightarrow$ total threads, $C \leq T$.

SpinLock [18, Figure 1] is a simple lock, commonly implemented as a test-test-and-set with exponential backoff. The test-test-and-set works like double-check locking: If the lock is open, then atomically attempt to set the lock closed.

```
if (*lock == OPEN && TAS(lock) == OPEN) break; // acquired lock ?
```

Hence, if the lock is closed, then the atomic TAS instruction is not executed. To reduce spinning on the atomic TAS, threads spins on a stack allocated variable in the acquire call for a short duration before the next attempt, called backoff. The backoff duration increases exponentially up to a maximum, when it is reset and the exponential climb begins again. The backoff means contending threads randomly interleave attempts to acquire the lock, while mostly spinning locally. However, our results show SpinLock can be extremely unfair, as the underlying hardware can prefer to optimize NUMA location over waiting time.

PthreadLock [10] is a blocking lock with a single fast-path FAS to acquire the lock, like SpinLock, and if that fails, then control drops into the kernel futex-path. Hence, PthreadLock eliminates application spinning during lock acquire but relies on lock release to unblock a kernel thread. PthreadLock is appropriate for guarding a long critical section, as it releases resources for the OS to use with other applications.

The code for all tested algorithms, except PthreadLock, is publicly available for inspection or experiments [4].

Table 1 shows the fundamental traits of the selected locks. The contended atomics estimates the number of atomic operations to acquire/release a lock, as these instructions can have a significant affect on the cache and pipeline. (Note that RMR cost on a store miss can be just as bad as a coherence miss on a CAS/FAS/FAA but is difficult to quantify.) The value N for HemLock and PthreadLock implies spinning on an atomic instruction, but the spinning is bounded, because the locks are FCFS. The value ∞ for SpinLock is unbounded, because a thread can experience starvation while spinning. The value FUTEX for PthreadLock is the blocking/unblocking cost for a kernel thread, which involves crossing the application/kernel boundary and using another set of locks to enqueue and dequeue threads on an OS wait channel. Lock data implies extra data stored in a lock, and TLS data implies extra data stored with a thread. Contended space is the space used while threads are attempting to acquire a lock.

6.1 Experimental Setup

Figure 4 shows the outline of the test harness for the experimental setup, which creates T pthread worker-threads, with T in the range 1–32; $T = 1$ is the minimal and $T > 1$ is the maximal

```
static void * Worker( void * arg ) {
    for ( int r = 0; r < RUNS; r += 1 )
        for ( entry = 0; stop == 0; entry += 1 )
            NCS();  acquire( &lock );  CS();  release( &lock );
}
pthread_t workers[Threads];
ctor();                                          // global algorithm constructor
for ( size_t tid = 0; tid < Threads; tid += 1 ) {   // start workers
    int rc = pthread_create( &workers[tid], NULL, Worker, set[tid] );
    affinity( workers[tid], tid );
} // threads start first experiment immediately
while ( Run < RUNS ) {                           // global variable
    sleep( Time );                               // delay experiment duration
    stop = 1;                                    // stop threads
    while ( Arrived != Threads ) Pause();        // all threads stopped ?
    Run += 1;
    stop = 0;                                    // start threads
    while ( Arrived != 0 ) Pause();              // all threads started ?
}
for ( size_t tid = 0; tid < Threads; tid += 1 ) {   // terminate workers
    int rc = pthread_join( workers[tid], NULL );
}
dtor();                                          // global algorithm destructor
```

Fig. 4. Outline of the test harness for the experimental setup.

contention experiments. After thread creation, the harness blocks for a fixed period and then sets a global stop flag to indicate the experiment is over. The T worker-threads repeatedly attempt entry into a self-checking CS until the stop flag is set. The CS contains a loop with a short delay, 20 iterations, and each iteration performs quick tests for mutual exclusion violation. As well, the NCS has a short delay of 20 iterations for $T > 1$ to disrupt threads convoying through the CS. These dynamic tests buttress the formal proof of an algorithm but, more importantly, verify the algorithm implementation on computers with different memory models, e.g., **total order store (TSO)** or **weak order (WO)**. The verification tests were invaluable during algorithm construction and testing.

Each experiment is run for 60 seconds, during which 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 60 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 for each T. The median value of the five results is plotted.

The performance experiments were run on three different multi-core hardware systems to determine differences across platforms as follows:

(1) Supermicro AS−1123US−TR4 AMD EPYC 7662 64−core socket, hyper-threading × 2 sockets (256 processing units) 2.0 GHz, TSO memory model, running Linux v5.8.0−55−generic, gcc−10 compiler
(2) Huawei ARM TaiShan 2280 V2 Kunpeng 920 48−core socket × 2 sockets 2.6 GHz, WO memory model, running Linux v5.4.0−109−generic, gcc−10 compiler

CLH	2041410577	MCSFAS	1070070839	SpinLock	2770954445
SpinLock	1956219230	MCS	1054716399	CLH	2754392619
MCSFAS	1915379239	QSpinLock	1024050798	QSpinLock	2604973550
HemLock	1903562544	SpinLock	1018898523	MCSFAS	2557004687
MCS	1890273211	CLH	1003620482	MCSK42	2541969467
MCSK42	1835395343	HemLock	958525415	HemLock	2510449023
QSpinLock	1829528261	MCSK42	934787734	MCS	2489269251
MCSH	1822318701	MCSH	930003248	MCSH	2454660856
PthreadLock	1738109666	PthreadLock	894072206	PthreadLock	2284870188
(a) AMD		(b) ARM		(c) Intel	

Fig. 5. Minimal Contention Throughput, $T = 1$, higher value is better.

(3) Supermicro SYS−6029U−TR4 Intel Xeon Gold 5220R 24−core socket, hyper-threading × 2 sockets (48 processing units) 2.2 GHz, TSO memory model, running Linux v5.8.0−59−generic, gcc−10 compiler

All three hardware architectures are different in threading (multithreading vs. hyper), cache structure (MESI/MESIF vs. MOESI), NUMA layout (QPI vs. HyperTransport), memory model (TSO vs. WO), and energy/thermal mechanisms (turbo-boost). Software that runs well on one architecture may run poorly or not at all on another.

All 32-threads are run on a single socket to prevent large NUMA effects. The threads on the ARM are placed on consecutive cores with an L3 cache step at 24 cores. The threads on the Intel and AMD are placed on hyper-threads per core and then consecutively on cores to minimize L3 steps (4 core step on AMD and 24 core step on Intel). Small NUMA effects occur between the L3 caches. No hyper-threading or NUMA effects were observed in the performance results.

Finally, compilation used optimization level -O3, most function calls are inlined (which may not be possible for pre-compiled implementations), and fencing is performed by inspection to match the architecture memory-model. The experiments were run with compiler fencing using _Atomic declarations (not shown) but ran equal to or slower than hand-generated fencing, because the compiler uses the stronger sequential-consistency model rather than the weaker TSO or WO models, respectively. Interestingly, the curves of the algorithms also changed positions using _Atomic across the different architectures; hence, drawing performance conclusions depends on the memory model used.

6.2 Experimental Results

Figure 5 shows the entry counts (throughput) to the critical section for the minimal contention experiment, i.e., an access with no contention ($T = 1$), for each locking algorithm run on the AMD, ARM, and Intel, respectively. The values are sorted to simplify the comparison. There is no NCS delay for this experiment as it would only lengthen the experiment without providing any addition information. Therefore, the performance values for $T = 1$ are not comparable to $T > 1$, which have an NCS delay.

The results for the minimal contention experiment show the algorithms are close in performance, ±15%−20%. In general, MCSH performed below average across the three different architectures for $T = 1$. Note that algorithm performance moves around significantly on the different architectures, making it impossible to select a single algorithm as the best.

Figure 6 shows the entry counts (throughput) to the critical section for the maximal contention experiment, $T = N$, for each locking algorithm run on the AMD, ARM, and Intel, respectively. The graphs start at $T = 2$, because the results for $T = 1$ are up to an order of magnitude greater than $T = 2$ and a broken graph is used, because the results for SpinLock remain high as contention increases. Note that the Y axis scale is different between the two parts of the graph. Without these graphing techniques, the results for $T = 2..32$ are compressed on the Y axis, making it difficult to see differences among them.

Except for SpinLock, the lock algorithms varied by a factor of 1.5 to 2 times across all architectures. (SpinLock performance is discussed next.) Of the locks with a standard interface, MCSK42, QSpinLock, HemLock, SpinLock, and PthreadLock, MCSH ranks about the same as MCSK42. Furthermore, MCSH performed in the middle group of locks across the three different architectures. Locks CLH and QSpinLock did well across the three different architectures, but CLH does not have a standard interface, and several locks (including MCSH) are equal to or better than QSpinLock on the Intel. Again, algorithm performance moves around on the different architectures, making it impossible to select a single algorithm as the best.

Figure 7 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. This relative standard deviation is a measure of long-term fairness across the experiment versus intermediate intervals of short term unfairness. 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. MCSFAS is not FCFS, but at maximal contention, there is rarely a successor in the lock release, so it behaves like FCFS MCS. SpinLock and PthreadLock are not FCFS. SpinLock gains its unfairness from the atomic test-and-set instruction, which is particularly unfair across all tested architectures. Basically, the hardware favours threads for long periods of time, so during a timed experiment, some threads receive diminishing execution time. PthreadLock does not gain as much from its unfairness, because it occasionally blocks the kernel thread rather than spinning, which has a substantial cost.

During development of the performance experiments, we found that the results are dependent on the application, workload, and architecture. For the application, the size of the NCS and CS has a significant effect on lock performance. For the workload, the amount of contention on the lock is the largest factor. For the architecture, the placement of threads on processor units, using hyperthreading, and the cache structure are important for good performance. Without explicit thread placement, the Linux operating system places threads far apart, because it assumes independent sequential programs. Also a best-effort attempt is made by the Linux scheduler to restart blocked kernel threads on or near the last CPU that it ran to preserve cache locality. These different thread placement result in different performance results across general programs. Hence, other criteria may direct lock selection, such as the simplicity of the interface or the requirements on the ABI.

7 CONCLUSIONS

The MCSH algorithm is a variation of the popular FCFS MCS lock with a standard interface requiring only basic programming-language mechanisms and stack allocation; hence, it is a substitutable lock for programs using the standard-lock interface without ABI concerns. The MCSH lock has been proven correct using both the proof assistant PVS and through extensive experimental testing. The experimental results show MCSH is performance equivalent to other MCS-style locks. Therefore, application performance is unlikely to change significantly if MCSH is substituted. This

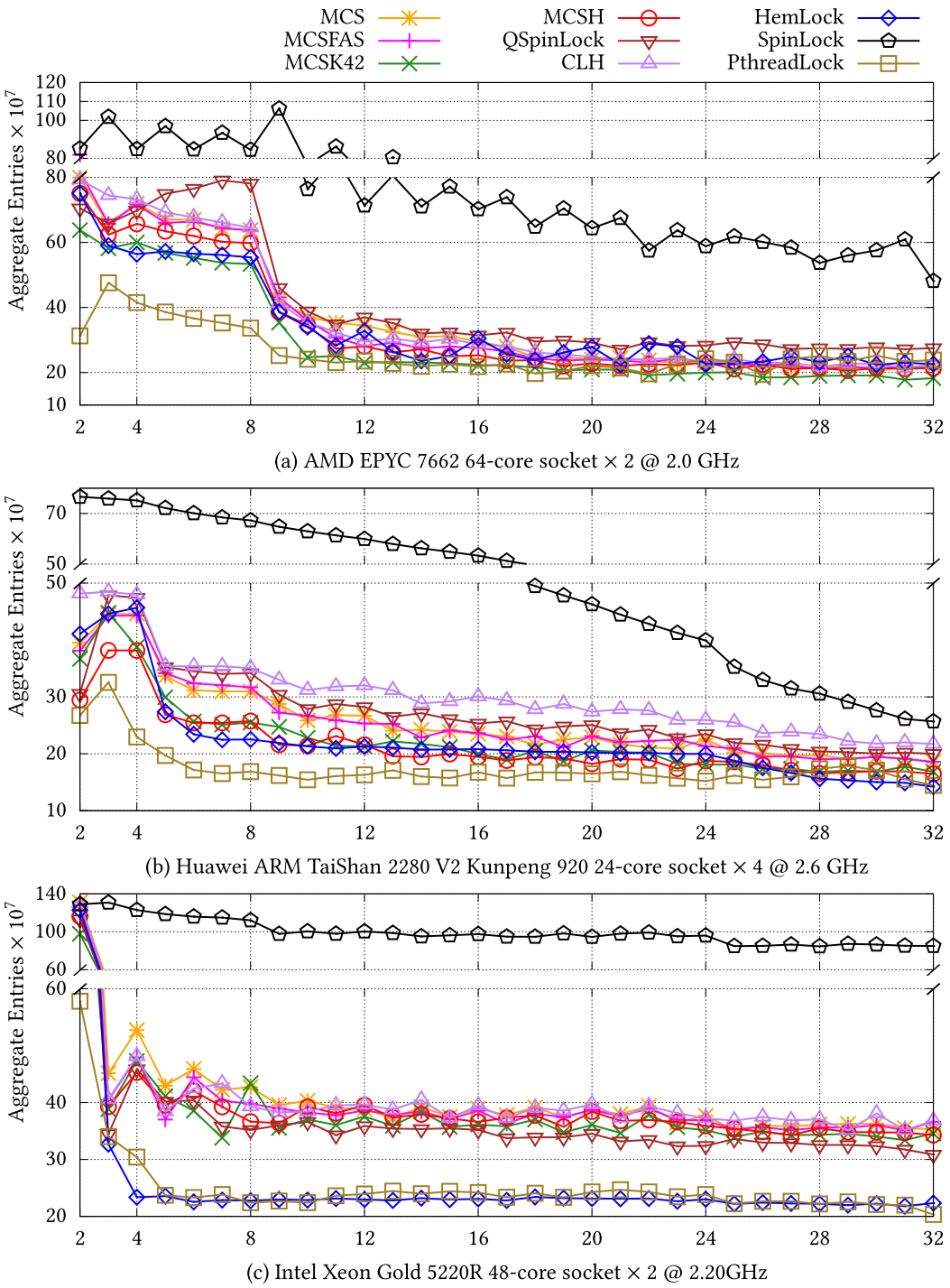
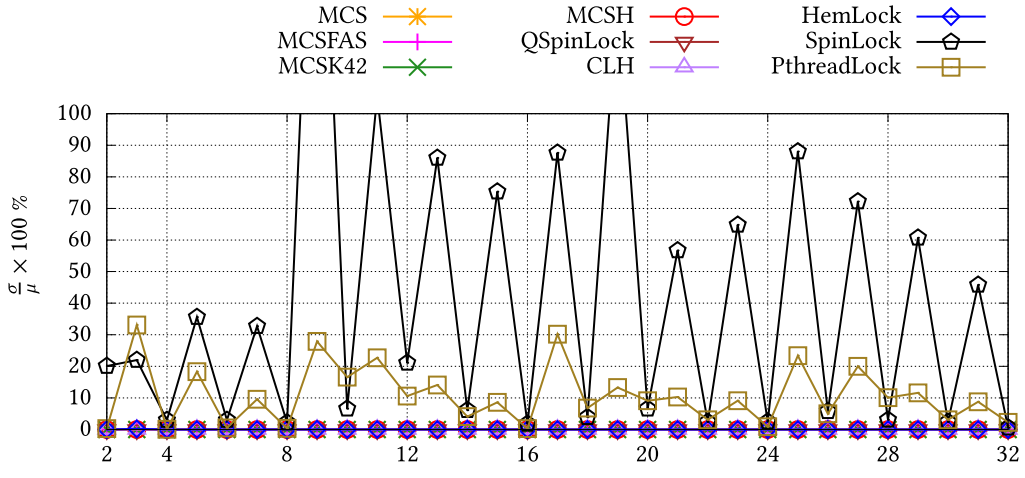
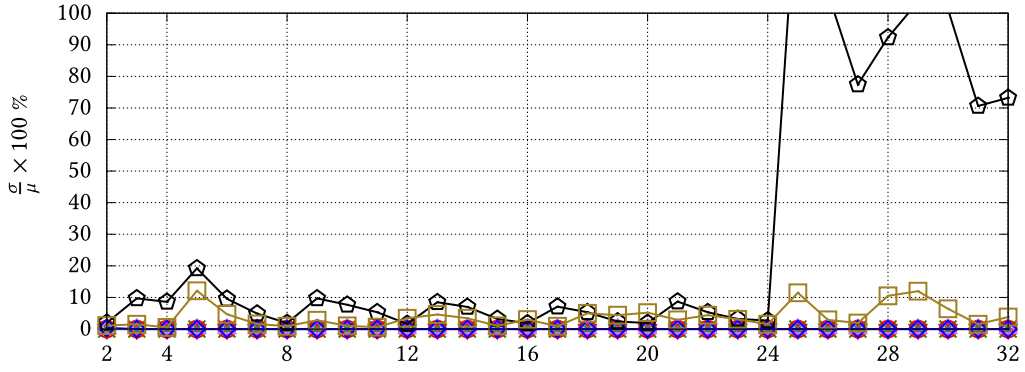


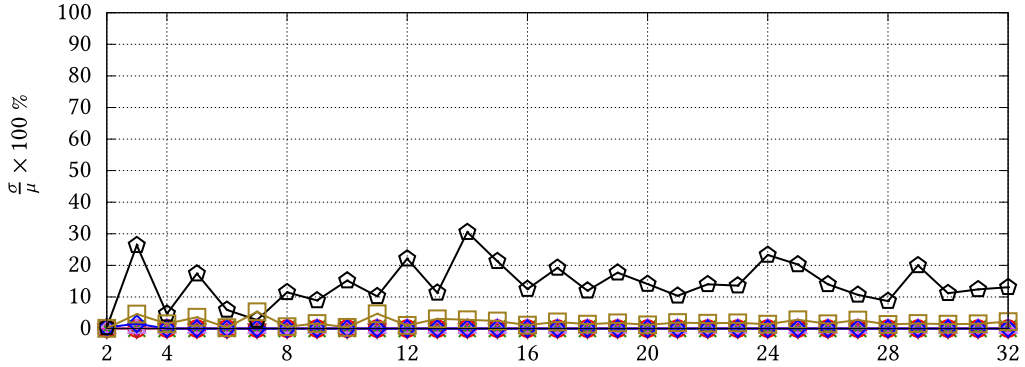
Fig. 6. Critical-section entry-counts, maximal contention: $T = 2..32$, 60 seconds, algorithm performance, higher value is better.



(a) AMD EPYC 7662 64-core socket × 2 @ 2.0 GHz



(b) Huawei ARM TaiShan 2280 V2 Kunpeng 920 24-core socket × 4 @ 2.6 GHz



(c) Intel Xeon Gold 5220R 48-core socket × 2 @ 2.20GHz

Fig. 7. Relative standard deviation, maximal contention: $T = 1..32$, 60 seconds, fairness among threads, where 0% is prefect fairness.

performance equivalence is obtained even though MCH, MCSFAS, CLH, and HemLock do not have a standard interface, and MCSFAS, MCSK42, SpinLock, PthreadLock are not FCFS. Hence, MCSH provides application programmers with a new, competitive, alternative lock with strong interface and behaviour properties.

A APPENDIX

The invariants of Figure 3 needed for the proof of mutual exclusion. The founding members *Iq1* and *Iq2* generate the following complete family. It uses the abbreviations M defined above and

$$F \;=\; \{q \mid (q \in [15] \wedge prev_q = \bot) \;\vee\; q \in [16]\}.$$

Iq1: $slot_q = slot_r \neq 0 \;\Rightarrow\; q = r$.
Iq2: $q \in M \;\Rightarrow\; slot_q = \mathtt{low}$.
Iq3: $slot_q < \mathtt{high}$.
Iq4: $\mathtt{tail} = \bot \;\wedge\; \mathtt{flag} \;\Rightarrow\; \mathtt{low} = \mathtt{high}$.
Iq5: $q \in [14, 17] \;\Rightarrow\; \mathtt{locked}(my_q)$.
Iq6: $q \in F \;\wedge\; r \in [20, 29] \;\Rightarrow\; \mathtt{low} + 1 = slot_q$.
Iq7: $q \in F \;\wedge\; r \in [21, 30] \;\Rightarrow\; nxmy_r = \bot$.
Iq8: $q \in [21, 30] \;\Rightarrow\; nxmy_q = \bot \;\vee\; \mathtt{low} = slot_q$.
Iq9: $q \in [21, 26] \;\vee\; q \in [29, 30] \;\Rightarrow\; succ_q = nxmy_q$.
Jq1: $q \in [21, 30] \;\wedge\; nxmy_q = my_r \;\wedge\; r \in [15, 26] \;\Rightarrow\; \mathtt{low} + 1 = slot_r$.
Jq2: $q \in [20, 29] \;\Rightarrow\; \neg\, \mathtt{flag}$.
Jq3: $q \in [25, 30] \;\wedge\; \mathtt{tail} = \bot \;\Rightarrow\; nxmy_q = \bot$.
Jq4: $q \in [25, 29] \;\wedge\; \mathtt{tail} = \bot \;\Rightarrow\; \mathtt{low} + 1 = \mathtt{high}$.
Jq5: $q \in [21, 30] \;\wedge\; r \in [12, 17] \;\Rightarrow\; nxmy_q \neq my_r$.
Jq6: $q \in [15, 24] \;\Rightarrow\; \mathtt{tail} \neq \bot$.
Jq7: $q \in F \;\Rightarrow\; r \notin [19, 24]$.
Jq8: $q \in [15, 26] \;\wedge\; r \in [15, 26] \;\wedge\; \mathtt{next}(my_q) = my_r \;\Rightarrow\; slot_q + 1 = slot_r$.
Jq9: $q \in [26, 28] \;\Rightarrow\; \mathtt{msg} = nxmy_q$.
Kq1: $q \in [21, 26] \;\wedge\; my_q = \mathtt{tail} \;\Rightarrow\; nxmy_q = \bot$.
Kq2: $q \in [15, 26] \;\wedge\; my_q = \mathtt{tail} \;\Rightarrow\; slot_q + 1 = \mathtt{high}$.
Kq3: $q \in [21, 30] \;\Rightarrow\; nxmy_q \in \mathtt{local}$.
Kq4: $q \in [15, 26] \;\wedge\; r \in [12, 26] \;\wedge\; \mathtt{next}(my_q) = my_r \;\Rightarrow\; r \in [18]$.
Kq5: $q \in [15, 29] \;\Rightarrow\; \mathtt{low} \leq slot_q$.
Kq6: $q \in F \;\wedge\; r \in F \;\Rightarrow\; q = r$.
Kq7: $q \in F \;\wedge\; r \in [18] \;\Rightarrow\; \mathtt{locked}(my_r)$.
Kq8: $q \in [13, 14] \;\Rightarrow\; \mathtt{next}(my_q) = \bot$.
Lq1: $q \in [15, 26] \;\wedge\; r \in [15, 17] \;\wedge\; my_q = prev_r \;\Rightarrow\; slot_q + 1 = slot_r$.
Lq2: $q \in [12, 26] \;\wedge\; r \in [12, 26] \;\wedge\; my_q = my_r \;\Rightarrow\; q = r$.
Lq3: $\mathtt{tail} = \bot \;\vee\; \mathtt{next}(\mathtt{tail}) = \bot$.
Lq4: $q \in [15, 26] \;\Rightarrow\; \mathtt{next}(my(q)) \in \mathtt{local}$.
Lq5: $\mathtt{low} \leq \mathtt{high}$.
Lq6: $q \in [30] \;\Rightarrow\; \mathtt{flag} \;\vee\; nxmy_q = \bot$.
Lq7: $q \in [12, 14] \;\wedge\; r \in [15, 17] \;\Rightarrow\; my_q \neq prev_r$.
Lq8: $q \in [12, 14] \;\Rightarrow\; my_q \neq \mathtt{tail}$.
Mq1: $q \in [12, 26] \;\Rightarrow\; my_q \in \mathtt{local}$.

Mq2: $q \in [15, 17] \Rightarrow prev_q \neq \texttt{tail}$.
Mq3: $\perp \in \texttt{local}$.
Mq4: $q \in [15, 17] \Rightarrow prev_q \in \texttt{local}$.
Mq5: $\texttt{tail} \in \texttt{local}$.
Mq6: $q \in [24, 26] \wedge r \in [15, 17] \Rightarrow my_q \neq prev_r$.
Mq7: $q \in [23, 26] \Rightarrow my_q \neq \texttt{tail}$.
Mq8: $q \in [21, 26] \wedge r \in [15, 17] \wedge my_q = prev_r \Rightarrow nxmy_q = \perp$.
Mq9: $q \in [15, 26] \wedge r \in [15, 17] \wedge my_q = prev_r \Rightarrow \texttt{next}(my_q) = \perp$.

These predicates form a directed graph, when an edge is drawn from P to Q if P is threatened by some command (line number), and Q belongs to a conjunction that serves as a remedy. In the list below, *MX1* can be replaced by *Iq1 & Iq2*. The graph is described as follows.

Iq1 is threatened by line 14 with remedy *Iq3*.
Iq2 threatened by 14: *Iq4*; 17: *Iq5*; 29: *Iq1 & Iq6 & Iq7*; 30: *Iq1 & Iq8 & Iq9 & Jq1* .
Iq3 is inductive.
Iq4 threatened by 22: *Jq2*; 29: *Jq3 & Jq4*, 30: *Iq9 & Jq3*.
Iq5 threatened by 30: *Iq9 & Jq5*.
Iq6 threatened by 14: *Jq4 & Jq6*; 19: *Jq7*; 29: *MX1*; 30: *Iq1 & Iq2 & Iq8 & Iq9*.
Iq7 threatened by 14: *Jq3 & Jq6*; 20, 24: *Jq7*.
Iq8 threatened by 20, 24: *Iq2*; 29: *Iq1 & Iq2*; 30: *Iq1 & Iq9*.
Iq9 threatened by 28: *MX1 & Jq9*.
Jq1 threatened by 14: *Jq5*; 20, 24: *Iq2 & Jq8*; 29, 30: *MX1 & Iq9*.
Jq2 threatened by 29: *MX1*.
Jq3 threatened by 21, 24: *Jq6*; 22: *Iq1 & Iq2 & Iq8 & Kq1*.
Jq4 threatened by 21, 24: *Jq6*; 22: *Iq2 & Kq2*; 29: *MX1*; 30: *MX1 & Iq9*.
Jq5 threatened by 11: *Kq3*; 20, 24: *Kq4*.
Jq6 threatened by 22: *Iq1 & Iq2 & Iq3 & Kq2 & Kq5*.
Jq7 threatened by 14: *Jq6*; 16: *Kq6*; 18: *Kq7*.
Jq8 threatened by 14: *Kq4 & Kq8*; 17: *Jq9* threatened by 25: *MX1 & Iq9*.
Kq1 threatened by 14: *Lq2*; 20, 24: *Lq3*.
Kq2 threatened by 14: *Lq2*.
Kq3 threatened by 20, 24: *Lq4*; 26: *Iq2 & Jq1*.
Kq4 threatened by 11: *Lq4*; 14; *Kq8*; 17: *Lq2*; 18: *Iq1 & Iq2 & Jq8 & Kq5*.
Kq5 threatened by 14: *Lq5*; 29: *Iq1 & Iq2 & Lq5*; 30: *Iq1 & Iq8 & Iq9*.
Kq6 threatened by 14: *Jq6*.
Kq7 threatened by 14: *Jq6*, 17: *Iq5*; 30: *MX1 & Iq9 & Lq6*.
Kq8 threatened by 17: *Lq7*.
Lq1 threatened by 14: *Kq2 & Lq7 & Lq8*.
Lq2 threatened by 11: *Mq1*.
Lq3 threatened by 14: *Kq8*; 17: *Mq2*.
Lq4 threatened by 12: *Mq3*; 14: *Kq8 & Mq3*; 17: *Mq1*; 26: *Iq2 & Jq8 & Kq5*.
Lq5 threatened by 29: *Iq2 & Iq3*; 30: *Iq3 & Iq8 & Iq9*.
Lq6 threatened by 19: *MX1*.
Lq7 threatened by 11: *Mq4*; 14: *Lq8*.
Lq8 threatened by 11: *Mq5*; 14: *Lq2*.
Mq1 threatened by 26: *Lq2*.

Mq2 threatened by 14: *Lq7* & *Lq8*; 22: *Iq1* & *Iq2* & *Iq3* & *Kq2* & *Kq5*.
Mq3 is inductive.
Mq4 threatened by 14: *Mq5*; 26: *Mq6*.
Mq5 threatened by 14: *Mq1*; 22: *Mq3*; 26: *Mq7*.
Mq6 threatened by 14: *Mq7*; 21: *Iq9* & *Mq8*; 22: *Mq2*; 23: *Mq9*.
Mq7 threatened by 14: *Lq2*; 21: *Iq9* & *Kq1*.
Mq8 threatened by 14: *Kq1*; 20, 24: *Mq9*.
Mq9 threatened by 14: *Kq8* & *Lq3*; 17: *Iq1* & *Lq1*.

ACKNOWLEDGMENTS

We thank Dave Dice for inspiring the MCSH lock and his cooperation, suggestions, and criticisms while writing the article. Discussions with Thierry Delisle and Colby Parsons also helped us to understand the lock behaviours.

REFERENCES

[1] M. Abadi and L. Lamport. 1991. The existence of refinement mappings. *Theor. Comput. Sci.* 82 (1991), 253–284.
[2] M. A. Auslander, D. J. Edelsohn, O. Y. Krieger, B. S. Rosenburg, and R. W. Wisniewski. 2003. Enhancement to the MCS lock for increased functionality and improved programmability. U.S. patent application number 20030200457 (abondoned).
[3] P. A. Buhr, D. Dice, and W. H. Hesselink. 2015. High-performance *N*-thread software solutions for mutual exclusion. *Concurr. Computat.: Pract. Exp.* 27 3 (March 2015), 651–701. http://dx.doi.org/10.1002/cpe.3263.
[4] Peter A. Buhr, David Dice, and Wim H. Hesselink. 2022. Locking Micro-Benchmarks. Retrieved from https://github.com/pabuhr/concurrent-locking.
[5] Dave Dice and Alex Kogan. 2021. Fissile locks. In *Networked Systems*, Chryssis Georgiou and Rupak Majumdar (Eds.). Vol. LNCS 12129. Springer International Publishing, Cham, Switzerland, 192–208.
[6] Dave Dice and Alex Kogan. 2021. Hemlock: Compact and scalable mutual exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, New York, NY, 173–183.
[7] Edsger W. Dijkstra. 1965. *Cooperating Sequential Processes*. Technical Report. Technological University, Eindhoven, 87 pages. https://pure.tue.nl/ws/files/4279816/344354178746665.pdf.
[8] Edsger W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (September 1965), 569.
[9] E. W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.
[10] Ulrich Drepper. 2022. Futexes Are Tricky, Red Hat Inc., 12 pages. Retrieved from https://cis.temple.edu/~giorgio/cis307/readings/futex.pdf.
[11] Wim H. Hesselink. 2022. PVS Proof Scripts for Verification of Hardware Locks. Retrieved from http://wimhesselink.nl/mechver/HardwareLocks.
[12] Wim H. Hesselink. 2022. Trylock, a case for temporal logic and eternity variables. *Sci. Comput. Program.* 216, C (April 2022), 12.
[13] IBM. 2021. Serially Reusable Programs. Retrieved from https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=structures-serially-reusable-programs.
[14] L. Lamport. 1974. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* 17 (1974), 453–455.
[15] L. Lamport. 1987. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.* 5 (1987), 1–11.
[16] Waiman Long. 2013. qspinlock: Introducing a 4-byte Queue Spinlock Implementation. Retrieved from https://lwn.net/Articles/561775.
[17] Peter S. Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*. IEEE Computer Society, Los Alamitos, CA, 165–171.
[18] J. M. Mellor-Crummey and M. L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9 (1991), 21–65.
[19] Maged M. Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504.
[20] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. 2020. *PVS Language Reference Version 7.1*. SRI International, Computer Science Laboratory, Menlo Park CA, 113 pages. Retrieved February 7, 2023 from https://pvs.csl.sri.com/doc/pvs-language-reference.pdf.

[21] M. L. Scott. 2013. *Shared-memory Synchronization*. Morgan & Claypool.
[22] Tianzheng Wang, Milind Chabbi, and Hideaki Kimura. 2016. Be my guest: MCS lock now welcomes guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. ACM, New York, NY, Article 21, 12 pages.
[23] WikipediA. 2022. Application Binary Interface.Retrieved February 7, 2023 from https://en.wikipedia.org/wiki/Application_binary_interface.
[24] WikipediA. 2022. Application Program Interface.Retrieved February 7, 2023 from https://en.wikipedia.org/wiki/API.
[25] WikipediA. 2022. Thread-local Storage. Retrieved February 7, 2023 from https://en.wikipedia.org/wiki/Thread-local_storage.

Received 29 June 2022; accepted 14 February 2023