Safety and Liveness of MCS Lock—Layer by Layer

Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao

Yale University

Abstract. The MCS Lock, a small but complex piece of low-level software, is a standard algorithm for providing inter-CPU locks with FIFO ordering guarantee and scalability. It is an interesting target for verification—short and subtle, involving both liveness and safety properties. We implemented and verified the MCS Lock algorithm as part of the CertiKOS kernel [8], showing that the C/assembly implementation *contextually refines* atomic specifications of the acquire and release lock methods. Our development follows the methodology of *certified concurrent abstraction layers* [7, 9]. By splitting the proof into layers, we can modularize it into separate parts for the low-level machine model, data abstraction, and reasoning about concurrent interleavings. This separation of concerns makes the layered methodology suitable for verified programming in the large, and our MCS Lock can be composed with other shared objects in CertiKOS kernel.

1 Introduction

The MCS algorithm for scalable fair inter-CPU mutex locks makes for an interesting case study in program verification. Although the program is short, the proof is challenging. First, the implementation of a lock algorithm can not itself use locks, so it has to rely solely on atomic memory instructions and be robust against any possible interleavings between CPUs. This is the most challenging type of concurrency, so-called lock-free programming. Second, unlike algorithms which only promise mutual exclusion, the MCS algorithm also aims for fairness among CPUs. To check that it got it right, our correctness theorem needs to guarantee not only mutual exclusion (a safety property) but also bounded waiting time (a liveness property).

Previous work [18, 21] has studied the correctness of the algorithm itself, but those verification efforts did not produce executable code, and did not explore how to integrate the proof of the algorithm into a larger system. We have created a fully verified implementation and addeded it as part of the CertiKOS kernel [8], which consists of 6500 lines of C and assembly implementation and 135K lines of Coq proofs.

In order to manage such a large verification effort, the CertiKOS team developed a methodology known as *certified (concurrent) abstraction layers*, as well as a set of libraries and theorems to support it. Previous papers [7, 9] described this framework, but many readers found them dense and hard to follow because they immediately present the formalism at its most abstract and general. This paper aims to be a complement: by zooming in on the implementation of one small part of the kernel (the MCS Lock module), we illustrate what it is like to *use* the framework, how to write specifications in the “layers” style, and what the corresponding proof obligations are. We hope this paper will be an easier entry point for understanding our verification framework.

As we will see, CertiKOS-style verification has several distinctive features which stem from the requirements of a large kernel. First, it is suitable for **dealing with low-level code**. To make the proofs tractable we mainly work at the C level (relying on the

CompCert verified compiler [16]), but sometimes we need to go lower. For example the MCS algorithm needs to use atomic CPU instructions (*fetch-and-store* and *compare-and-swap*), so we need a way to mix C and assembly code, while stating precisely what semantics we assume that the assembly code has. At the same time, C itself is too low-level to conveniently reason about, so we need **data abstraction** to hide the details about representation in memory.

Second, in order to handle large developments we need **separation of responsibilities**. In a small proof of an algorithm in isolation, you can state the specification as a single pre- and post-condition which specifies the shape and ownership of the data structure, the invariants (e.g. mutual exclusion), the liveness conditions, and even the behavior of the lock's client code (the critical section code). But such a proof is not modular and not re-usable. In our development, these are done as separate refinement steps, in separate modules with explicit interfaces, and can even be the responsibility of different software developers.

Finally, the layers approach is **general purpose**, in the sense that the same semantic framework can be used for proving all kinds of properties. The model of program execution exposed to the programmer is simple, mostly the same as for sequential code and with a notion of logs of events to model concurrency. Unlike working in a special-purpose program logic, we did not have to add any features to show a liveness property, because we can directly reason in Coq about **how long** an execution will take.

In the remaining parts of the paper, we first explain the C code that we will be verifying (Sec. 2). Then in the bulk of the paper, we explain our proof strategy by going through each abstraction layer in turn, concluding with the safety and starvation freedom properties (Sec. 4). Finally we explain how our proofs fit as a part of the larger CertiKOS development (Sec. 5) and discuss related and future work (Sec. 6). We revisit several points that have been mentioned in previous publications:

– We show how to customize the machine model by adding a trusted specification of particular instructions that we need. (Sec. 4.1.)
– We locally verify the execution of a single CPU, and treat the rest of the system as an opaque *concurrent context*. (Sec. 4.2.)
– We illustrate how to abstract away from a C implementation, by refining it into a functional specification which can be conveniently reasoned about. (Sec. 4.3.)
– Similarly, we show how the same type of refinement can be used to gradually add ghost state to a specification while hiding un-needed details. (Sec. 4.4.)

We also make novel contributions:

– It provides a concrete example of CertiKOS-style verification; in particular we can see how to customize the machine model (Sec. 4.1) and how to split the verification effort into CPU-local reasoning (Sec. 3.1 and 4.2).
– We show a way to prove that an atomic specification refines a concurrent implementation, while still using downward rather than upward simulations. The trick is to provide a *function* from low-level to high-level logs of events. (Sec. 4.5–4.6.)
– We propose a new way to specify the desired—atomic—behavior of the lock/unlock methods. To ensure liveness, the specification of the lock method itself includes a promise to later call unlock; we do this using a bounding counter. (Sec. 4.5.)

– And of course, we provide the first implementation of the MCS algorithm that has been both rigorously verified (with a mechanized proof) and at the same time realized (as part of a running kernel).

2 The MCS algorithm

The MCS algorithm [20] is a list-based queuing lock, which provides a *fair* and *scalable* mutex mechanism for multi-CPU computers. Fairness means that CPUs that compete for the lock are guaranteed receive it in the same order as they asked for it (FIFO order). With an unfair lock, CPUs that try to take the lock can get nondeterministically passed over (even a million times in a row [3]) creating unpredictable latency.

Fairness is also important to verification, because without it there is the possibility that one particular CPU is continuously passed over so it loops forever—this is infinitely improbable, but not impossible. So unless the lock guarantees fairness, there is no way to prove a termination-sensitive refinement between the implementation and a simple (terminating) specification. With a non-fair lock, we would have to settle for either an ugly specification that allowed non-termination, or for a weaker notion of correctness such as termination-insensitive refinement.

```c
typedef struct _mcs_node{
  uint next;
  uint busy;
  uint _node_padding[14];
}mcs_node;
```

```c
typedef struct _mcs_lock{
  uint last;
  uint _lock_padding[15];
  mcs_node ndpool[TOTAL_CPU];
}mcs_lock;
```

```c
void mcs_acquire(uint lk_id){
  uint cpuid, prev;
  cpuid = get_CPU_ID();
  LK[lk_id].ndpool[cpuid].busy = BUSY;
  LK[lk_id].ndpool[cpuid].next =
     TOTAL_CPU;
  prev = FAS(&(LK[lk_id].last),cpuid);
  if(prev == TOTAL_CPU) return;
  LK[lk_id].ndpool[prev].next = cpuid;
  while(LK[lk_id].ndpool[cpuid].busy==
     BUSY);
  return;
```

```c
}
void mcs_release(uint lk_id){
  uint cpuid, nid;
  cpuid = get_CPU_ID();
  if(CAS(&(LK[lk_id].last),cpuid,
     TOTAL_CPU) return;
  while (LK[lk_id].ndpool[cpuid].next==
     TOTAL_CPU);
  nid = LK[lk_id].ndpool[cpuid].next;
  LK[lk_id].ndpool[nid].busy = FREE;
  return;
}
```

Fig. 1: Data Structure and the implementation of MCS Lock (in C).

The data structure of an MCS Lock (Fig. 1) has one global field pointing to the last node of the queue structure, and per-CPU nodes forming each node in the queue. This is similar to an ordinary queue data structure(, but note that it only has a pointer to the tail, not the head of the queue). If the queue is empty, we set `last` to the value `TOTAL_CPU`, which acts as a null value (we could also have used e.g. -1). The queue is used to order the waiting CPUs, in order to ensure that lock acquisition is FIFO. The structs also include padding to take up a full cache lines and avoid false sharing. Each node is owned by one particular CPU (the array is indexed by CPU id). This is what makes the lock scalable: a CPU looping waiting for the lock will only read its own busy flag, so there is no cache-line bouncing. Simpler lock algorithms make all the CPUs read the same memory location, which does not scale past 10-40 CPUs [1].

Fig. 1 shows the code for the acquire lock and release lock operations. The acquire lock function uses an atomic *fetch-and-store* expression to *fetch* the current `last` value and *store* its CPU-id as the `last` value of the lock in a single action (line 7). Then, if the previous `last` value was `TOTAL_CPU`, the CPU can directly acquire the lock and enter the critical section (line 9). If the previous `last` value was not `TOTAL_CPU`, it means that some other CPUs are in the critical section or in the queue waiting to enter it (line 10 to line 13). In this case, the current CPU will wait until the previous node in the queue sets the current CPU's busy flag as `FREE` during the lock release.

Release lock also has two execution paths, based on the result of an atomic operation, *compare-and-swap* (line 21). The `CAS` operation succeeds, immediately releasing the lock, if the current CPU is the only one in the queue. If the `CAS` fails, this implies that some other CPU has already performed the *fetch-and-store* operation (line 7). Thus, the current CPU busy waits until that other CPU sets the `next` field (line 10), and then passes the lock to the head of the waiting queue by assigning `busy`.

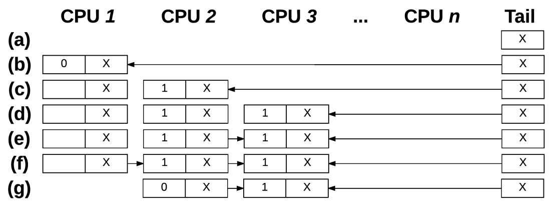

Fig. 2: A possible execution sequence for an MCS Lock.

Fig. 2 illustrates a possible sequence of states taken by the algorithm. At the beginning (a), the lock is free, and CPU 1 can take it in a single atomic `FAS` operation (b). Since CPU 1 did not have to wait for the lock, it does not need to update its *next*-pointer. After that, CPUs 2 and 3 each try to take the lock ((c) and (d)). The last value will be updated correctly thanks to the property of the atomic expression. However, there can be some delay in-between a CPU updating the *tail* pointer, and it adjusting the *next*-pointer of the previous node in the queue; as this example illustrates, that means although there are three nodes which logically makes up the queue of waiting CPUs, any subset of the next-pointers may be unset. At (e), although CPU 1 wants to release the lock, the `CAS` call will return false (because `tail` is 3, not 1). In this case, CPU 1 has to wait in a busy-loop until CPU 2 has set its *next*-pointer (f). After that, the CPU 1 can set the busy flag to `FREE` for the next node in the queue, CPU 2's node, which releases the lock (g).

Because the algorithm is fair, it satisfies a *liveness* property:

> "Suppose all clients of the lock are well-behaved, i.e. whenever they acquire a lock they release it again after some finite time, and suppose the scheduling of operations from different CPUs is fair. Then whenever `mcs_acquire` or `mcs_release` are called they will succeed within some finite time."

A big part of our formal development is devoted to stating and proving this. Let us give a an informal proof sketch here. Consider a CPU which starts executing `mcs_acquire`. At this time, the queue contains a finite number of CPUs already waiting for the lock. By fairness of scheduling, the CPU at the head of the queue will get scheduled periodically, say every F steps. Each time it is scheduled, it will go through three phases. First, it will execute the code in `mcs_acquire`; because it is at the head of the queue the loop will terminate immediately. Then it executes the code in the critical section; by assumption this is over after some finite number k of operations. Finally it executes `mcs_release`; this either finishes immediately, or enters the waiting loop, in which case it will complete as soon as the next CPU in the queue gets scheduled.

3 Abstraction Layers

The most distinctive thing about CertiKOS-style verification is the notion of *abstraction layers* [7]. Of course, any large-scale programming or verification project uses layers of abstraction, but typically these are merely an informal organization that the programmer has in mind when writing the program. In CertiKOS, we formalize layers as objects defined in Coq, these layers are treated as first-class objects, and we use the framework to vertically compose them. We split the MCS Lock verification into five layers, each building on the interface exposed by the layer below.

Our notion of a "layer interface" is a particular style of state machine where the transitions correspond to function calls, while a "layer" in our development is a proof of refinement between interfaces. More formally, an *abstraction layer* is a tuple (L_1, M, L_2), together with a refinement proof showing that the code M, when run on top of a system specified by the interface L_1, faithfully implements the interface L_2. Then another layer (L_2, M', L_3) can run on top of the first one. Functions in M' can call functions in M, but we only need to look at the specification L_2 to prove them correct.

The code M is a set of functions written in C or assembly, and the entire stack of layers can be compiled to executable code using a modified version of CompCert called CompCertX [7]. It is also possible to have a layer with no code at all. Such a "pure refinement" layer represents a proof that the interfaces L_1 and L_2 are equivalent. The last two layers in our development are pure refinements.

Each layer interface L is a pair $L = (A, P)$, where A is Coq data type (usually a record type) which we call the *abstract state type*, and P is a set of named *primitive specifications* which describe the behavior of C/assembly functions. Each primitive specification $\sigma \in P$ is written as a Coq function of type $\sigma : (val^* \times mem \times A) \rightarrow$ `option` $(val \times mem \times A)$. The types val and mem are borrowed from CompCert's operational semantics for C; val and val^* are the type of C values and lists of values (for the function return value and arguments), and mem is the type of C memory states.

The idea is that a pair $(m, d) : mem \times A$ represents the state of the computer. A typical refinement proof for a layer $((A_1, P_1), M, (A_2, P_2))$ will give a relation R saying that the fields in A_2 represents certain objects stored in memory. Then the high-level specifications in P_2 can refer to the abstract value d when specifications in P_1 had to talk about the memory state m. In particular, in Sec. 4.1 we will define a layer which proves a relation between the array `LK` (see Fig. 1) and an abstract state. The specifications in all layers about it never need to mention memory again, so they avoid all the side conditions to do with C memory accesses.

3.1 Events, logs, and concurrent contexts

In order to handle concurrent programs, the verification framework imposes some structure on the specifications [9]. Each record type A must include at least a *log of events* (written l) and a *concurrent context* (written ε, further explained in Sec. 4.2). For almost all of the MCS Lock development, these are the only two fields that matter. Instead of representing the state of shared memory by an arbitrary type A, it will be represented using the log.

```
Inductive TicketOracleEvent :=
  // Events for MCS-lock primitives
  | SWAP_TAIL (bound: nat) (IS_CPU_NUM: bool) | CAS_TAIL (success : bool)
  | GET_NEXT | SET_NEXT (old_tail: Z) | GET_BUSY (busy : bool) | SET_BUSY
  // Events for the high-level queue-lock
  | WAIT_LOCK (n: nat) | REL_LOCK.

Inductive SharedMemEvent := ...
```

Fig. 3: Event set for MCS Lock

An *event* is any action which has observable consequences for other CPUs. Each specification must define events for all the points in the program where it reads or writes to shared memory (but not for accesses to thread-local memory) The *log* is a list of events, representing all actions that have happened in the computer since it began running. Actions from different CPUs are interleaved in the list. When we write a specification we can chose the set of events, as long as it is fine-grained enough to capture all scheduling interleavings that may happen. Fig. 3 shows the event definition used to model lock acquire and release. They correspond to the part of the MCS lock source code in Fig. 1 and releasing the lock after we show starvation freedom.

Because all CPUs see a single linear log, this model assumes that the machine is sequentially consistent. Even with this assumption, verifying the MCS algorithm is not easy (the other proofs we are aware of assume sequential consistency too), so we leave weak memory models to future work.

`SWAP_TAIL bound success` event is for the operations from line 5 to 7 in Fig. 1 and takes two arguments. The second argument is a boolean flag indicating whether the previous "last" value of MCS lock was `MCS_TOTAL_CPU`, which means it records whether the if-statement at line 9 took the fast path or not. The first argument is the *bound number*, which is a key idea in our development. Every client that invokes `mcs_acquire` has to promise a bound for the critical section. This number does not influence the compiled code in any way, but the *specification* says that it is invalid to hold the critical section for longer than that (c.f. the counter c_1 in Sec. 4.4). This bound number enables *local* reasoning about liveness. For the thread waiting for acquiring or releasing a lock, its wait time can be estimated based on other threads' bound number. For the lock holder, it has to guarantee to exit the critical section within its own bound. Thus, by showing that each thread follows this protocol, we can derive the liveness property for the whole system. (To be precise, the bound number is a limit on the number of events that can get appended to the log, see the counter `c1` in Sec. 4.4. Every CPU adds at least one event every time it "does something", e.g. each loop iteration in `mcs_release` appends a GET_NEXT event, so as we will see in Sec. 4.5 this sufficies to give a bound of the number of loop iterations in the lock acquire function. In the following we often speak of "number of operations", which does not mean single CPU instructions, but instead whatever operation is represented by particular events.)

`SET_NEXT prev_last` event corresponds to the code at line 10. the `prev_last` represents the `prev_id` in the code.

`GET_BUSY busy` event shows the busy waiting in the acquire lock function. The first argument will be true when the last value is same with the current CPU-id that calls the primitive which generates this event. It will be false when the last value is not same with the current CPU-id that calls the primitive.

`CAS_TAIL busy` represents the atomic expression at line 21 in Fig. 1. In addition, the "busy" corresponds to the result of the `CAS` operation in Fig. 1.

`GET_NEXT` corresponds to the primitive that try to get the next value of the current CPU's node, and abstracts busy waiting in release lock function.

`SET_BUSY` represents the last three lines in `mcs_release`.

Those six events are used to show the functional correctness of an MCS Lock. However, for clients that use the MCS Lock to build shared objects they expose too much implementation details. In Sec. 4.5 we will prove linearizability and starvation freedom, to replace them with just two events

`WAIT_LOCK bound` corresponds to lock acquire. The "bound" number in here is exactly same with the "bound" number in `SWAP_TAIL bound success` event.

`REL_LOCK` corresponds to the lock release.

In addition to the above eight events, which are generated by the lock acquire and release functions, the clients of the lock will also generate events while they are in the critical section. Mutex locks in CertiKOS are used to protect blocks of shared memory, so we call the events generated by the client code **shared memory events**. The final specification we prove will entail that a shared memory event from CPU i can only happen in the interval between an lock acquire event for i and a lock release event for i, which is how we express the mutual exclusion property.

4 Verification—Layer by layer

We build five layers, starting from a base layer which represents the machine model that our compiled code will run on.Fig. 4 shows the overall structure of our development. For simplicity the figure only includes lock primitives, and not primitives passed through from below. The arrows show dependencies between adjacent layers, for example the definition of `wait_lock` in `MMCSLockOp` uses three primitives (`mcs_swap_tail`, `mcs_set_next`, and `mcs_get_busy`) from the `MMCSLockAbsIntro` layer.

The layers `MCSMCurID` through `MMCSLockAbsIntro` introduces getter and setter functions for accessing memory (Sec. 4.1 and 4.2). These layers also contain logical primitives which record events to the log; we are in effect manually implementing a model of concurrent execution by extending a sequential operational semantics for C.

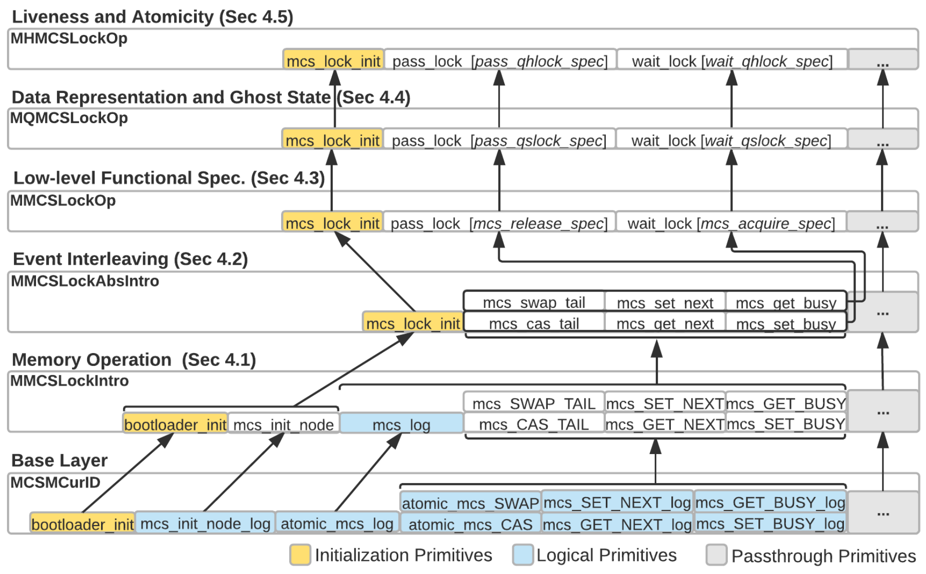

Fig. 4: MCS Lock Layers

The layer `MMCSLockOp` contains the C code from Fig. 1. This layer proves low-level functional correctness, i.e. it reasons about the C code and abstracts away details about memory accesses, integer overflows, etc, to expose an equivalent specification written as a Coq function (Sec. 4.3).

The two top layers, `MQCSLockOp` and `MHMCSLockOp`, do not introduce any new primitives. They simplify the specifications of the release- and acquire lock functions (`pass_lock` and `wait_lock`), i.e. each layer ascribes a different specification (with a different log replay function and set of events) to the same C function. Those specification names are notated inside the square bracket in Fig. 4.

The layer `MQMCSLockOp` adds ghost state, keeping track of a queue of waiting CPUs. (Sec 4.4). This queue is key to the liveness proof but is not explicitly represented in the C implementation. The top layer `MHMCSLockOp` proves starvation freedom and liveness (Sec 4.5). This lets us ascribe atomic specifications where taking or releasing a lock generates just a single event to the log.

4.1 Memory operations layers

Although we glossed over this in Fig. 1, our actual C implementations of `msc_acquire` and `msc_release` do not access memory directly. Instead, they call a collection of helper functions with names like `mcs_set_next`. The lowest two layers in our proofs are devoted to implementing these helper functions.

We first describe the first and the lowest tuple in our proofs, (L_0, M, L_1). The interface (L_0) represents the machine model that our compiled code will run on. All primitives defined in L_0 are part of the trusted computing base, and correspond to empty functions in our compiled code.

Eight of the primitives in L_0 are closely related to the MCS Lock verification:

$$\{\text{atomic\_mcs\_log},\ \text{atomic\_mcs\_SWAP},\ \text{atomic\_mcs\_CAS},\ \text{mcs\_init\_node\_log},\\ \text{mcs\_GET\_NEXT\_log},\ \text{mcs\_SET\_NEXT\_log},\ \text{mcs\_GET\_BUSY\_log},\ \text{mcs\_SET\_BUSY\_log}\}$$

Two primitives, `atomic_mcs_SWAP` and `atomic_mcs_CAS` are for the two atomic instructions *fetch-and-store* and *compare-and-swap*, and will be further discussed below.

The other six are used to update the log. As we noted in Sec. 3.1, the log is part of the abstract state. Ordinary assembly instructions only modify physical memory, not abstract state, so in order for programs to be able to append events to the log we include these six primitives in L_0. For example, the specification of `mcs_SET_NEXT_log` updates the log by adding one (`SET_NEXT prev_id`) event as follows:

```
Function mcs_SET_NEXT_log_spec(abid cpuid prev_id: Z)(adt:RData): option RData:=
    ...
    match ZMap.get abid (multi_log adt) with
    | MultiDef l =>
      let l' := (... (SET_NEXT prev_id))) :: l in
      Some adt {multi_log: ZMap.set abid (MultiDef l') (multi_log adt)}
    ...
  end.
```

In the compiled code, these primitives appear as empty functions that do nothing, they are only used to modify the logical state.

The code M in the layer contains the functions which actually modifies the memory in the way the event announces. Each function in M calls the corresponding primitive from `MCSMCurID` inside the function to add the event to the log. For example, `mcs_SET_NEXT`, one function in M, writes to `next` and also calls the empty function `mcs_SET_NEXT_log`:

```
void mcs_SET_NEXT(uint lk_id, uint cpuid, uint pv_id) {
    mcs_SET_NEXT_log(lk_id, cpuid, pv_id);
    (LK[lk_id].ndpool[pv_id]).next = cpuid; }
```

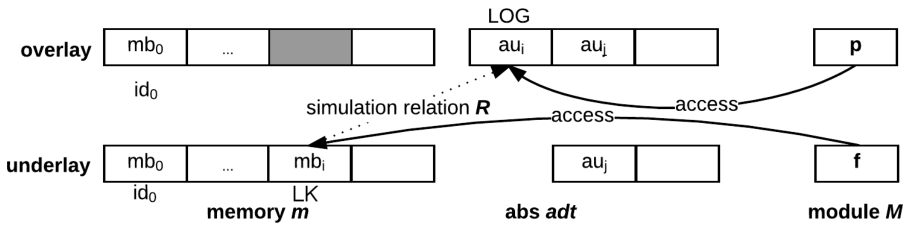

Fig. 5: The structure of the memory operations layer

The interface `MMCSLockIntro` contains the high level specification for each function defined in M. The high level specifications works on the log instead of the exact memory slot `LK`. Therefore, after proving the *refinement* between the memory (`LK` in Fig. 5) and the abstract state (*log* in Fig. 5), we only need to care about the abstract state.

For the refinement proof, we need two more ingredients. The first one is a *log replay function*. A log is merely a list of events, but what specifications need to know is what the state of the system will look like after those events have executed, and a replay function calculates that. Different layers may define different replay functions in order to interpret the same log in a way that suits their proofs. In L_1, we define `CalMCSLock`, which has the following type:

```
CalMCSLock : MultiLog -> option MCSLock
```

where

```
Inductive  MCSLock :=
| MCSLOCK (tail: Z) (lock_array: ZMap.t (bool * Z)) ...
```

The return type of this log replay function closely corresponds to C data structures, which makes it easy to prove the refinement. (`ZMap` is a finite map from `Z` to `bool*Z`.) The second ingredient is a relation R which shows the relationship between the concrete memory in underlay L_0 and the abstract state in overlay L_1. As a part of R, we define `match_MCSLOCK` as follows:

Definition 1 (`match_MCSLOCK`). *Suppose that 'loc' is among the proper field accessors for the MCS Lock (i.e. '`last`', '`ndpool[i].next`', or '`ndpool[i].busy`' when '$0 \leqslant i <$ `TOTAL_CPU`'). And, assuming that `l` is a shared log. Then define*

$$\texttt{match\_MCSLOCK (l: Log) (b: block) loc}$$
$$\textit{iff}\ (\exists\ \texttt{val, Mem.load Mint32 m b loc = Some(val)} \wedge \texttt{Mem.valid\_access m b loc}$$
$$\wedge\ \texttt{(CalMCSLock(l) = Some(mcsval) -> loc}_a\texttt{@mcsval = val))}$$

when '$\texttt{loc}_a\texttt{@mcsval}$' represents the corresponding value to the '$\texttt{loc}_a$' in the '`mcsval`' and '$\texttt{loc}_a$' corresponds to the value of 'loc'.

Intuitively, the definition says that the value that `CalMCSLock` calculates from the log always corresponds to the value in the memory with the same identifiers. The memory access functions `Mem.load` and `Mem.valid_access` are from CompCert's operational semantics for C. Using the definition, we prove one theorem for each primitive, which shows that the memory refines the shared log. E.g., for `mcs_SET_NEXT` we prove:

Theorem 2 (Simulation for `mcs_SET_NEXT`). *Let R be the relation defined as `match_MCSLOCK` over `LK@`mem and `LOG@`A_1, identity relation for other parts of mem, A_0 and A_1. Then*

$$\begin{array}{l}
\forall (\mathtt{m}_1\ \mathtt{m}_1'\ \mathtt{m}_0 : mem)\ (\mathtt{d}_0\ : A_0)\ (\mathtt{d}_1\ \mathtt{d}_1' : A_1),\\
\textit{if}\ \texttt{mcs\_SET\_NEXT}_{L_1}(v, \mathtt{m}_1, \mathtt{d}_1) = \texttt{Some}(\mathtt{m}_1', \mathtt{d}_1')\ \textit{and}\ R\ (\mathtt{m}_1, \mathtt{d}_1)\ (\mathtt{m}_0, \mathtt{d}_0),\\
\textit{then there exists}\ (\mathtt{m}_0' : mem)\ (\mathtt{d}_0' : A_0),\ \textit{such that}\\
\texttt{mcs\_SET\_NEXT}_{L_0}(v, \mathtt{m}_0, \mathtt{d}_0) = \texttt{Some}(\mathtt{m}_0', \mathtt{d}_0')\ \textit{and}\ R\ (\mathtt{m}_1', \mathtt{d}_1')\ (\mathtt{m}_0', \mathtt{d}_0').
\end{array}$$

One interesting variation is the semantics for fetch-and-store and compare-and-swap. These instructions are not formalized in the x86 assembly semantics we use, so we cannot prove that replay function is correctly defined. Instead we modify the last ("pretty-printing") phase of the compiler so that these primitive calls map to assembly instructions, and one has to trust that they match the specification.

4.2 Event interleaving layer

After abstracting memory accesses into the operation on the log, we then need to model possible interleaving among multiple CPUs. In our approach, this is done through a new layer which adds *context queries*.

The concurrent context ε (sometimes called the "oracle") is a function of the CPU-id and the log which has the type ε : `Z -> list event -> list event`. It is one component of the abstract state, and it represents *all the other CPUs*, from the perspective of code running on the current CPU. Any time a program does an operation which reads or writes shared memory, it should first query ε by giving it the current log. The oracle will reply with a list of events that other CPUs have generated since then, and we update the log by appending those new events to it.

Primitive specifications are provided read-only access to a context ε by the verification framework, and the framework also guarantees that two properties are true of ε: 1) the returned partial log from the oracle query does not contain any events generated by the given CPU-id; and 2) if we query the oracle with the well-formed shared log, the updated log after the oracle query will be well-formed.

Similar to Sec. 4.1, we provide primitives in L_0 which query ε and extend the log. Then in this second layer, we can model abstract operations with interleaving. For example, `mcs_SET_NEXT` can be re-written as

```
void mcs_set_next(uint lk_id, uint cpuid, uint pv_id){
    mcs_log(lk_id, cpuid);
    mcs_SET_NEXT(lk_id, cpuid, pv_id); }
```

by using the logical primitive which corresponds to the oracle query (The function `mcs_log` refines the semantics of `atomic_mcs_log` in the lowest layer by the `match_MCSLOCK` relation). To model the interleaving, all the setter and getter functions defined in Sec. 4.1 should be combined with the oracle query.

Trust in the machine model Some of the design decisions in the memory access layers have to be trusted, so the division between machine model and implementation is unfortunately slightly blurred. Ideally, we would have a generic machine model as proposed by Gu et al [8], where memory is partitioned into thread-local memory (no events), lock-protected memory (accesses generate PUSH/PULL events), and atomic memory (each access generates one READ/WRITE/SWAP/etc event). However, our starting point is the CompCert x86 semantics, which was designed for single-threaded programs, and does not come with a log, so we add a log and memory access primitives ourselves. But because the spinlock module is only code in the OS that uses atomic memory, we do not add a generic operation called read_word etc. Instead we take a shortcut and specify the particular 6 memory accesses that the lock code uses: `mcs_get_next` etc. For these procedures to correctly express the intended semantics, there are two trusted parts we must take care to get right. First, each access to non-thread-local memory must generate an event, so we must not forget the call to `mcs_SET_NEXT_log`. Second, to account for interleavings between CPUs (and not accidentally assume that consecutive operations execute atomically) we must not forget the call to `mcs_log` after each access.

4.3 Low-level functional specification

Using the primitives that we have defined in lower layers, we prove the correctness of lock acquire, `mcs_acquire`, and release, `mcs_release`. The target code in this layer is identical to the code in Fig. 1 except two aspects. First, we replaced all operations on memory with the getters and setters described in Sec. 4.2. Second, `mcs_acquire` has one more argument, which is the bound number for the client code of the lock.

Since the functions defined in Sec. 4.2 already abstract interleaving of multiple CPUs, the proofs in this layers work just like sequential code verification. We find out the machine state after the function call by applying the C operational semantics to our function implementation, and check that it is equal to the desired state defined in our specification.

However, writing the specifications for these functions is slightly subtle, because they contain while-loops without any obviously decreasing numbers. Since our specifications are Coq functions we need to model this by structural recursion, in some way that later will let us show the loop is terminating. So to define the semantics of `mcs_wait_lock`, we define an auxiliary function `CalMCS_AcqWait` which describes the behavior of the first n iterations of the loop: each iteration queries the the environment context ε, replays the log to see if if `busy` is now `false`, and appends a `GET_BUSY` event. If we do not succeed within n iterations the function is undefined (Coq `None`). Then, in the part of the specification for the acquire lock function (`CalMCS_AcqWait`) where we need to talk about the while loop, we say that it loops for some "sufficiently large" number of iterations `CalWaitLockTime tq`.

```
Fixpoint CalMCS_AcqWait (n: nat) (i : Z) l o : option MultiLog :=
 match n with
   | O => None
   | S n' =>
     let l' :=  (to i l) ++ l in
     match CalMCSLock l' with
       | Some (MCSLOCK tail la _) =>
         match ZMap.get i la with
           | (false, _) => Some ((TEVENT i (TTICKET (GET_BUSY false))) :: l')
           | _ =>
             CalMCS_AcqWait n' i ((TEVENT i (TTICKET (GET_BUSY true))) ::l') to
         end
       | _ => None
     end
   end.
```

The function `CalWaitLockTime` computes a suitable number of loop iterations based on `tq`, the time-bounds which each of the queuing CPUs promised to respect. We will show how it is defined in Sec. 4.5. However, in *this part* of the proof, the definition doesn't matter. Computations where n reaches 0 are considered crashing, and our ultimate theorem is about safe programs, so when proving that the C code matches the specification we only need to consider cases when `CalMCS_AcqWait` returned `(Some l)`. It is easy to show in a downward simulation that the C loop can match any such finite run, since the C loop can run any number of times.

4.4 Data representation and ghost state

From here on, we never have to think about C programs again. All the subsequent reasoning is done on Coq functions manipulating ordinary Coq data types, such as lists, finite maps, and unbounded integers. Verifying functional programs written in Coq's Gallina is exactly the situation Coq was designed to deal with. However, the data computed by the replay function in in the previous layer still corresponds exactly to the array-of-structs that represents the state of the lock in memory. In particular, the intuitive reason that the algorithm is fair is that each CPU has to wait in a queue, but this conceptual queue is not identical with the linked-list in memory, because the next-pointers may not be set.

In order to keep the data-representation and liveness concerns separate, we introduce an intermediate layer, which keeps the same sequence of operations and same log of events, but manipulates an *abstracted data representation*. We provide a different replay function with the type `QS_CalLock: Multi_Log -> option (nat * nat * head_status * list Z * ZSet.t * list nat)`. The tuple returned by this replay function provides the information we need to prove liveness, similar to the concepts used in the informal proof in Sec. 2. The meaning of a tuple `(c1, c2, b, q, slow, t)` is as follows: `c1` and `c2` are upper bounds on how many more operations the CPU which currently holds the lock will generate as part of the critical section and of releasing the lock, respectively. They are purely logical ghost state but can be deduced from the complete history of events in the system. `b` is either `LHOLD` or `LFREE`, the lock status of the head of the queue. `q` is the list of the CPUs currently waiting for the lock, and `t` is the list of bound numbers that corresponds to each element in `q`. `slow` is a finite set which represents the subset of CPUs in `q` that have not yet executed their *set next* operation. Our liveness proof is based on the fact that each CPU only needs to wait for CPUs that are ahead of it in `q`.

The tuple returned by this replay function provides the information we need to prove liveness, similar to the concepts used in the informal proof in Sec. 2. The meaning of a tuple `(c1, c2, b, q, slow, t)` is as follows:

c1 An upper bound on how many more operations the CPU which currently holds the lock will do as part of the critical section. It is as known as a bound number.

c2 An upper bound on how many more operations the CPU which currently holds the lock will do as part of releasing the lock. We need this number because release lock also contains a waiting loop.

b Either `LHOLD`, meaning that a CPU currently holds the lock, or `LFREE` meaning that it has been released and the next CPU in the queue has not yet taken it.

q The list of the CPUs currently waiting for the lock (CPUs are identified by a numerical id). It is a waiting queue and the starvation freedom proof is mainly related to the property of the decrease of this queue. jieung

slow A finite set (using the finite set type `MSet` from the Coq standard library) which represents the subset of CPUs in `q` which have not yet executed their *set next* operation.

t This list specifies, for each CPU in `q`, a maximum number of operations which that CPU promises it will not exceed once it holds the lock.

Some of this information is implicit in the state of the memory, while some of it (for example `c1` and `c2`) is purely ghost state. But in any case, it can be deduced from the complete history of events in the system, which is what the replay function `QS_calLock` does. A few representative cases of the function are shown in Fig. 6. For example, the event `SET_BUSY` indicates that a thread releases the lock. If the CPU i is already the front of the queue q, it currently holds the lock (`LHOLD`), and the bound `c2` has not yet reached zero, and i is not slow, then generating this event will reset the lock status to `LEMPTY` and remove the head element (i) from `q` and `t`. In any of those side conditions are not satisfied, on the other hand, the replay function is undefined (`None`). Similar considerations hold executing memory operations (you must be in the critical section, and it decrements `c1`) and querying the busy flag (you must have executed `SET_NEXT` first).

```
Fixpoint QS_CalLock (l: MultiLog)  :=
  match l with | (TEVENT i e) :: l' =>
    match QS_CalLock l' with
    | Some (c1, c2, b, q, slow, t) =>
      | i0 :: q', p0 :: tq' =>
        if zeq i i0
        then match b, e with
             | LHOLD, TTICKET SET_BUSY =>
               match c2 with
               | O => None
               | S c2' => if (negb (ZSet.mem i slow))
                          then Some (0, 0, LEMPTY, q', slow, t') else None
               end
             | LHOLD, TSHARED _ =>
               match c1 with
               | O => None
               | S c1' => Some (c1', c2,LHOLD,q,slow,t)
               end
               ...
        else match b, e with
             | _, TTICKET (GET_BUSY true) =>
                  if ZSet.mem i slow
                  then None else Some (self_c, rel_c, b, q, slow, t)
      ...
```

Fig. 6: The replay function `QS_CalLock`

Invariant The replay function plays two different roles. When it returns `Some v`, for some tuple `v`, it describes what the current state of the system is, which lets us write the specifications for the primitives. At the same time, the cases where the function is defined to return `None` are also important, because this can be read as a description of events that are *not* possible. For example, from inspecting the program, we know that each CPU will create exactly one `SET_NEXT` event before it starts generating `GET_BUSY` events, and this fact will be needed when doing the proofs in the later layers (Sec. 4.5). By taking advantage of the side conditions in the replay function, we can express all the invariants about the log in a single statement, “the replay function is defined”:

∃ `c1 c2 b q s t. QS_CalLock(l) = Some(c1, c2, b, q, s, t)`

This type for the replay function is optimized to only expose exactly the information needed by the subsequent liveness proof. We need to expose the queue and the set of slow CPUs in order to define the termination measure M (Sec 4.5). On the other hand, this is not enough information to bridge the gap from the low-level functional specification.

In order to show that the memory cells for a valid linked-list and therefore respects the queue ordering, we need to track exactly what the valid state transitions are. So inside the ghost state layer, we also introduce a different relation `Q_CalMCSLock` which is mostly the same as `QS_CalLock` but written as an (functional) inductive relation in Coq instead of a recursive function, and which has even more preconditions for when it is defined. We then add one more condition in the layer invariant saying that `Q_CalMCSLock` and `QS_CalLock` output the same result. Most of the proofs inside the ghost layer are done using the relation instead of the function. For simplicity, we will ignore the distinction in the rest of the paper, and write the lemma statements about `QS_CalLock` even if they used `Q_CalMCSLock` in the actual Coq code.

To show that the ghost layer refines the previous layer, we show a one-step forward-downward refinement: if the method from the higher layer returns, then method in the lower layer returns a related value. For this particular layer the log doesn't change, so the relation in the refinement is just equality, and the programmer just have to show that the lower-level methods are at least as defined and that they return equal results for equal arguments.

As we prove this, we need lemmas to show that we can satisfy the preconditions for the operations in the lower layer, by relating the data in `la` to the abstract queue. For example, when trying to take the lock, the high level specification checks if the current CPU is at the head of `q`, which the low specification tests if the `busy` field is true, so we need Lemma 5 to show that they will follow the same path of code.

Lemma 3 (tail soundness). *If* `CalMCSLock l = Some (tl, la, tq)` *and* $QS\_CalLock = Some(c1, c2, q, s, t)$*, then* `tl` *is* `NULL` *if* $q = \texttt{nil}$*, and* `tl` *the last element of* `q` *if* $q \neq \texttt{nil}$.

Lemma 4 (next-correctness). *Let's assume that* `CalMCSLock l = Some (tl, la, tq)` *and* `QS_CalLock = Some (c1,c2,q_1++`i`::`j`::q_2,s,t)`*, then* `lock_array[`i`] = (_, TOTAL_CPU)` *if* $j \in$ `s`*, and* `lock_array[`i`] = (_, `j`)` *if* $j \notin$ `s`.

Lemma 5 (tail is busy). *If* `CalMCSLock l = Some (tl, la, tq)` *and* `QS_CalLock = Some ( c1,c2,i::` q, s, t`)` *and* $j \in$ `q`*, then* `lock_array[`j`] = (true, _)`.

Theorem 6 (simulation for the ghost layer). *Suppose* `d` *satisfies the invariant and* `wait_qslock_spec(d)= Some(d')`*. Then* `mcs_acquire_spec(d)= Some(d')`.

4.5 Liveness and atomicity

The specification in the previous section is still too low-level and complex to be usable by client code in the rest of the system. First, the specification of the `mcs_acquire` and `mcs_release` primitives contain loops, with complicated bounds on the number of iterations, which clients certainly will not want to reason directly about. More importantly, since the specifications generate multiple events, clients would have to show all interleavings generate equivalent results.

To solve this we propose a basic design pattern: build a new layer with *atomic specifications*, i.e. each primitive is specified to generate a single event. For an atomic layer there is a therefore a one-to-one mapping between events and primitives, and the global log can be seen as a record of which primitives were invoked in which order. Thus, the refinement proof which ascribes an atomic specification proves once and for all that overlapping and interleaved primitive invocations give correct results. In this layer, the specifications only use three kinds of events: taking the lock (`WAIT_LOCK n`), releasing it (`PASS_LOCK`), and modifications of the shared memory that the lock protects (`TSHARED _`).

Fig. 7 shows the final specification for the wait primitive. We show this one in full detail, with no elisions, because this is the interface that clients use. First, the specification for the lock aquire function itself (`mcs_wai_hlock_spcec`) takes the function arguments `bound`, `index`, `ofs`, and maps an abstract state (`RData`) to another. When writing this specification we chose to use two components in the abstract state, the log (`multi_log`) and also a field (`lock`) which records for each numbered lock if it is free (`LockFalse`) or owned by a CPU (`LockOwn`). The `lock` field is not very important, because the same information can also be computed from the log, but exposing it directly to clients is sometimes more convenient.

The specification returns `None` in some cases, and it is the responsibility of the client to ensure that does not happen. So clients must ensure that: the CPU is in kernel/host mode (for the memory accesses to work); the index/offset (used to compute the lock id) are in range; the CPU did not already hold the lock (`LockFalse`); and the log is well-formed (`H_CalLock l'` is defined, which will always be the case if `H_CalLock l` is defined). When all these preconditions are satisfied, the specification queries the context once, and appends a single new `WAIT_LOCK` event to the log. Fig. 7 also shows the replay function `H_CalLock`. It has a much simpler type than `QS_CalLock` in the previous layer, because we have abstracted the internal state of the lock to just whether it is free (`LEMPTY`), held (`LHOLD`), and if taken, the CPU id (`Some i`) of the holder of the lock. Unlike the three bound numbers in the previous layer, here we omit the numbers for the internal lock operations and only keep the bound `self_c` for the number of events generated during the critical section. Again, it's the client's responsibility to avoid the cases when `H_CalLock` returns `None`. In particular, it is only allowed to release the lock or to generate memory events if it already holds the lock (`zeq i i0`), and each memory event decrements the counter, which must not reach zero. The client calling `wait_lock` specifies the initial value n of the counter, promising to take at most n actions within the critical section.

In the rest of the section, we show how to prove that the function does in fact satisfy this high-level atomic specification. Unlike the previous layers we considered, in this case the log in the upper layer differs from the one in the lower layer. For example, when a CPU takes the lock, the log in the upper layer just has the one atomic event (`WAIT_LOCK n`), while the log in the underlay has a flurry of activity (swap the tail pointer, set the next-pointer, repeatedly query the busy-flag). Because the log represents shared data, our framework requires a slightly strengthened refinement theorem for the log-component of the state. Usually a refinement simulation works by specifying some relation R between machine state and abstract state, and then proving that the state transitions preserve the relation. Indeed, for thread-local data this is exactly what CertiKOS does also.

However, an arbitrary relation R is not enough for local reasoning about concurrent programs. For example, suppose one particular execution of the system generates the log $\mathtt{l}_L$. A normal simulation theorem for CPU 1 would tell us that there *exists* a log $\mathtt{l}_H$ that meets CPU 1's local specification and satisfies the relation ($R\ \mathtt{l}_H$,$\mathtt{l}_L$). Similarly, the local proof for CPU 2 would say there exists some log $\mathtt{l'}_H$. But in order to derive a simulation for the entire system, we need the constraint that that $\mathtt{l}_H$ is equal to $\mathtt{l'}_H$. Our

```
Fixpoint H_CalLock (l: MultiLog) : option (nat * head_status * option Z) :=
  match l with
    | nil => Some (O, LEMPTY, None)
    | (TEVENT i e) :: l' =>
      match H_CalLock l' with
        | Some (S self_c', LHOLD, Some i0) =>
          match zeq i i0, e with
            | left _, TTICKET REL_LOCK => Some (O, LEMPTY, None)
            | left _, TSHARED _ => Some (self_c', LHOLD, Some i0)
            | _, _ => None end
        | Some (_, LEMPTY, None) =>
          match e with
            | TTICKET (WAIT_LOCK n) => Some (n, LHOLD, Some i)
            | _ => None end
        | _ => None end end.
```

```
Definition mcs_wait_hlock_spec (bound index ofs :Z) (adt: RData) : option RData :=
  let cpu := CPU_ID adt in
  match (ikern adt, ihost adt, index2Z index ofs) with
  | (true,  true, Some abid) =>
   match ZMap.get abid (multi_log adt), ZMap.get abid (lock adt) with
   | MultiDef l, LockFalse =>
     let to := ZMap.get abid (multi_oracle adt) in
     let l1 := (to cpu l) ++ l in
     let l' := (TEVENT cpu (TTICKET (WAIT_LOCK (Z.to_nat bound)))) :: l1 in
     match H_CalLock l' with
     | Some _ =>
       Some adt {multi_log: ZMap.set abid (MultiDef l') (multi_log adt)}
                {lock: ZMap.set abid LockOwn (lock adt)}
     | _ => None end
   | _, _ => None end
  | _ => None end.
```

Fig. 7: The final, atomic, specification of the aquire lock function.

solution is to require the relation R to be a function f. In other words, when proving the simulation, we find a function f for the logs, such that $f(\mathtt{l}_L) = \mathtt{l}_H$.

As for the MCS Lock, we define a function `relate_mcs_log` from the implementation log to the atomic log. Fig. 8 shows by example what it does. It keeps the shared memory events as they are, discards the events that are generated while a CPU wait for the lock, and maps just the event that finally takes or releases the lock into `WAIT_LOCK` and `REL_LCOK`.

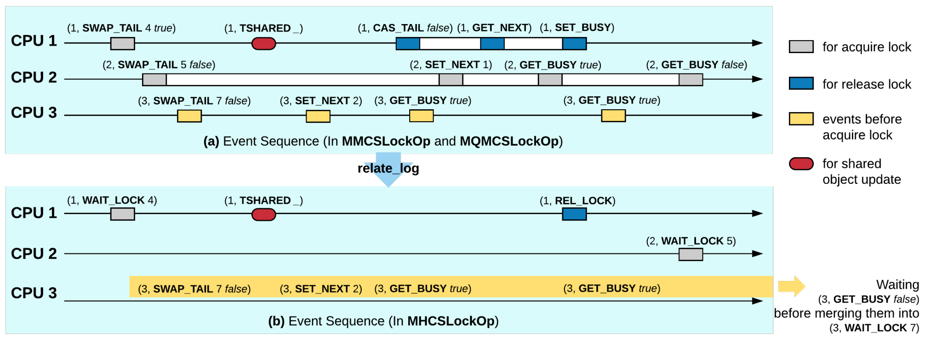

Fig. 8: Log Sequence and Log Refinement Example

We then prove a one-step refinement theorem from the atomic specification to the implementation, in other words, that if a call to the atomic primitive returns a value, then a call to its implementation also returns with a related log:

Theorem 7 (MCS Wait Lock Exist). *Suppose* $\mathtt{d}_{\mathit{MHMCSLockOp}}$ *and* $\mathtt{d}_{\mathit{MQMCSLockOp}}$ *satisfy the layer invariants and are related by* $\mathtt{relate\_mcs\_log}(\mathtt{d}_{\mathit{MQMCSLockOp}}) = \mathtt{d}_{\mathit{MHMCSLockOp}}$. *If* $\mathtt{wait\_hlock\_spec}(\mathtt{d}_{\mathit{MHMCSLockOp}}) = \mathtt{Some}(\mathtt{d}'_{\mathit{MHMCSLockOp}})$, *then there exists some* $\mathtt{d}'_{\mathit{MQMCSLockOp}}$ *which is* $\mathtt{wait\_qslock\_spec}(\mathtt{d}_{\mathit{MQMCSLockOp}}) = \mathtt{d}'_{\mathit{MQMCSLockOp}}$ *and is related with* $\mathtt{d}'_{\mathit{MHMCSLockOp}}$ *by* $\mathtt{relate\_mcs\_log}(\mathtt{d}'_{\mathit{MQMCSLockOp}}) = \mathtt{d}'_{\mathit{MHMCSLockOp}}$.

The proof requires a *fairness assumption*. A CPU cannot take the lock until the previous CPU releases it, and the previous CPU cannot release it if it never gets to run. At its most fundamental, the CertiKOS machine model is a nondeterministic transition system (which is subsequently viewed as a log of events), and there is nothing in the basic model that ensures fairness, so we have to add an extra assumption somewhere. In principle it would be possible to modify the machine model itself, and then pass the fairness assumptions along in the specification of each layer until we reach the layers related to mutex locks, but in our development we choose a more expedient solution, and express the fairness assumption as an extra axiom talking about the logs in the data representation layer (Sec. 4.4). By doing that, our framework can use the previous machine model as it is, and can reuse most previous proofs.

Specifically, we assume that there exists some constant F (for "fairness") such that no CPU that enters the queue has to wait for more than F events until it runs again. In Coq we provide a function `CalBound` which "counts down" until CPU i gets a chance to execute (`CalBound : Z -> MultiLog -> nat`).

The fairness assumption, then is that for all logs `l`, when the low level log replay function returns a value (`QS_CalLock(l) = Some(c1,c2,h,q,s,t)`) and j is in the waiting queue ($j \in$ `q`), then `CalBound` j `l > 0`.

We then define a natural-number valued termination measure M_i`(c1,c2,h,q,s,l)`. This is a bound on how many events the CPU i will have to wait for in a state represented by the log `l`, and where `QS_CalLock(l) = Some(c1,c2,h,q++`i`::q`$_0$`,s,t++n::t`$_0$`)`. Note that we partition the waiting queue into two parts `q` and i`::q`$_0$, where `q` represents the waiting CPUs that were ahead of i in the queue. The function M has two cases that depend on the head status.

$$M_i(\texttt{c1,c2,LEMPTY,q,s,l}) = \texttt{CalBound}_{\mathsf{hd}(q)}(\texttt{l}) + (K_1(\Sigma\texttt{t}) + |\texttt{q}\cup\texttt{s}|) \times K_2$$
$$M_i(\texttt{c1,c2,LHOLD,q,s,l}) = \texttt{CalBound}_{\mathsf{hd}(q)}(\texttt{l}) + \texttt{BoundValAux} \times K_2$$
$$\text{where } \texttt{BoundValAux} = (\texttt{c1}+\texttt{c2}+(\Sigma(\mathsf{tl}(\texttt{t})) \times K_1 + |\mathsf{tl}(\texttt{q})\cup\texttt{s}|)$$

In short, if the lock is not taken, the bound M is the sum of the maximum time until the first thread in the queue gets scheduled again ($\texttt{CalBound}_{\mathsf{hd}(q)}(\texttt{l})$), plus a constant times the sum of the number of operations to be done by the CPUs ahead of i in the queue (Σ`t`) and the number of CPUs ahead of i which has yet to execute `SET_NEXT` operation ($|$`q`$\cup$`s`$|$). If the lock is currently held, then `c1 + c2` is a bound of the number of operations it will do(and we can ignore the first element of `q` and `t`, since they are accounted for). The constants and fairness assumption is general enough to handle the cases which takes a slightly longer execution than it is expected to. The constants ($K_1 = F + 5$ and $K_2 = F + 4$) are chosen somewhat arbitrary, and certainly M is not the tightest possible bound. It doesn't need to be, since it does not occur in our final theorem statement.

. The definition of M is justified by the following two lemmas. First, we prove that M decreases if CPU i is waiting and some other CPU j executes an event `e`$_j$.

Lemma 8 (Decreasing measure for other CPUs). *Assuming that `QS_CalLock(l) = Some(c1,c2,h,q`$_1$`++`i`::q`$_2$`,s,t`$_1$`++c::t`$_2$`)`, where $|q_1| = |t_1|$ as well as `QS_CalLock(e`$_j$`::l) = Some(c1',c2',h',q',s',t')` for some $j \neq i$ and `CalBound(e`$_j$`::l) > 0` . Then we can split `q' = q`$_1$`'++`i`::q`$_2$`'`, and M_i`(c1',c2',h',q`$_1$`',s',t`$_1$`',e`$_j$`::l)` $<$ M_i`(c1,c2,h,q`$_1$`,s,t`$_1$`,l)`.*

Proof. The proof follows the informal outline in Sec. 2. We consider all possible events `e`$_j$ which could make `QS_CalLock` return `Some`. If j is not the CPU at the head of the queue gets scheduled, it will not be able to make any progress, so the abstract state of the queue remains the same, but the counter `CalBound` decreases. Otherwise, the counter `CalBound` will reset to the upper bound we assumed on fairness, F. However, in this case the algorithm will make some progress that changes `c1`, `c2`, `q`, or `s`. For example, CPU j may execute a `SET_NEXT` (which decreases the size of `s`), it may enter the critical section (which moves some measure from the head of `q` to the counters `c1+c2`) or it may exit the section (and that event will decrement `c2`).

The second lemma ensures that the waiting loop will eventually terminate. (The preconditions that i is somewhere in the waiting queue, and that it has already left the set `s`, correspond the the set-up which `wait_lock` does before it starts looping).

Lemma 9 (Loop termination). *Let's assume that `QS_CalLock(l) = Some(c1,c2,h,q`$_1$`++` i`::q`$_2$`,s,t`$_1$`++c::t`$_2$`)`, where $|q_1| = |t_1|$, with $i \notin q_1$ and $i \notin$ `s`. If $k > M_i$`(c1,c2,h,q`$_1$`,s,t`$_1$`)`, then there exists `l'` such that `CalWaitGet(`k`,`i`,l) = Some(l')`.*

Proof. The proof is by induction on k, the number of loop iterations. The most interesting part of the proof is to show that each event generated by the function will decrease the measure. As it pulls more event to the log form the context, we appeal to

Lemma 8, which says that the metric decreases. Then, there are two cases in the proof depending on whether i has arrived at the head of the queue (so `q = nil`) or not. If it has, `wait_qslock_spec` will generate a `GET_BUSY false` even and return, so we are good. Otherwise, it will generate a `GET_BUSY true` event, and start another loop iteration. That event does not change the state of the lock, but it does decrement the `CalBound` on when the head CPU will get scheduled next, so the measure decreases as required.

To prove the termination of the loop in `wait_qslock_spec`, we also need to show that the busy-loop in `pass_qslock_spec` terminates, but that proof is easier. A CPU holding the lock will set the next pointer before it does anything else, so we are only waiting for the CPU at the head of the queue to get scheduled at all. Now, to prove that the loop in `mcs_acquire` specification is defined, we just have to pick the function `CalWaitLockTime` so that `CalWaitLockTime(t)` is greater than M at that point. The rest of the simulation proof for Theorem 7 is straightforward. Except the waiting loop, other operations in the wait lock function are deterministic and finite.

4.6 From downwards- to upwards-simulation

When moving from sequential to concurrent programs we must re-visit some fundamental facts about refinement proofs. Ultimately, the correctness theorem we want to prove is "all behaviors of the machine satisfy the specification". If we model the machine and the specification as two transition systems M and S, then this corresponds to *upwards simulation*: if $S \sim M$ and $M \Longrightarrow^* M'$, then $\exists S'.S' \sim M'$ and $S \Longrightarrow^* S'$, and if M is stuck then S is stuck also. But directly proving an upwards simulation is difficult. You are given a long sequence of low-level steps, and have to somehow reconstruct the high-level steps and high-level ghost state corresponding to it. One of the insights that made the CompCert project possible [17] is that as long as M is deterministic and S is not stuck, it suffices to prove a *downward simulation*: if $S \sim M$ and $S \Longrightarrow S'$, then $\exists M'.S' \sim M'$ and $M \Longrightarrow^* M'$. (The assumption that S is not stuck is standard, it corresponds to only proving refinement for "safe" clients regarding to the specifications.)

Unfortunately, concurrent programs are *not* deterministic: we want to prove that every interleaving of operations from different CPUs in the low-level machine results in correct behavior. So if we had directly modeled the implementation as a nondeterministic transition system, then we would have to work directly with upwards simulations, which would be intractable when reasoning about the low-level details of C programs.

In our approach, all the nondeterminism is isolated to the concurrent context ε. Any possible interleaving of the threads can be modelled by initializing the abstract state with a particular ε_L, and the execution proceeds deterministically from there. Therefore we can still use the Compcert/CertiKOS method of first proving a downward simulation and then concluding the existence of a upward simulation as a corollary.

The context-formalism is also helpful because ε_L contains the entire execution of the other threads, both past and future, so we have enough information to directly prove a *forward* simulation. Otherwise it may not be clear if a given low-level operation can really "commit" (and generate a high-level event) until we see what the other cores do, so proofs about fine-grained concurrency can require a difficult backwards-simulation from the end-state of the program. [6]

There is still an obligation to show that for every ε_L, there in fact exists an ε_H with the right properties. (Specifically, it should the always output logs which respect the program invariants, i.e. the replay function is defined, and also it should respect the refinement relation f.) But this can be managed by the framework in a generic way [9]. When verifying a particular layer, the programmer only needs to define f.

5 Evaluation

```
uint palloc (uint cid){
  ...
  acquire_lock_AT();
  ...
  release_lock_AT();
  return palloc_free_index; }
```

```
Inductive SharedMemEvent :=
| OMEME (l: list Integers.Byte.int)
| OATE (a: ATable)
| OPALLOCE (b: Z)
...
```

Fig. 9: `palloc` Example

Clients The verified MCS lock code is used by multiple clients in the CertiKOS system. To be practical the design should require as little extra work as possible compared to verifying non-concurrent programs, both to better match the programmer's mental model, and to allow code-resuse from the earlier, single-processor version of CertiKOS.

For this reason, we don't want our machine model to generate an event for every single memory access to shared memory. Instead we use what we call a *push/pull memory model* [8,9]. A CPU that wants to access shared memory first generates a "pull" event, which declares that that CPU now owns a particular block of memory. After it is done it generates a "push" event, which publishes the CPU's local view of memory to the rest of the system. In this way, individual memory reads and writes are treated by the same standard operational semantics as in sequential programs, but the state of the shared memory can still be replayed from the log. The push/pull operations are logical (generate no machine code) but because the replay function is undefined if two different CPUs try to pull at the same time, they force the programmer to prove that programs are well-synchronized and race-free. Like we did for atomic memory operations, we extend the machine model at the lowest layer by adding logical primitives, e.g. `release_shared` which takes a memory block identifier as argument and adds a `OMEME (l:list Integers .Byte.int)` event to the log, where the byte list is a copy of the contents of the shared memory block when the primitive was called.

When we use `acquire/release_shared` we need a lock to make sure that only one CPU pulls, so we begin by defining combined functions `acquire_lock` which takes the lock (with a bound of 10) and then pulls, and `release_lock` which pushes and then releases the lock. The specification is similar to `pass_hlock_spec`, except it appends *two* events.

Similar to Sec. 4.5, logs for different layers can use different types of pull/push events. Fig. 9 (right) shows the events for the `palloc` function (which uses a lock to protect the page allocation table). The lowest layer in the palloc-verification adds `OMEME` events, while higher layers instead add (`OATE (a: ATable)`) events, where the relation between logs uses the same relation as between raw memory and abstract `ATable` data. Therefore, we write wrapper functions `acquire/release_lock_AT_spec`, where the implementation just calls `acquire/release_lock` with the particular memory block that contains the allocation table, but the specification adds an `OATE` event.

We can then ascribe a low-level functional specification `palloc'_spec` to the `palloc` function. As shown in Fig 10, this is decomposed into three parts, the acquire/release

```
Definition release_lock_AT_spec adt := ...
  let l' := TEVENT cpu (TTICKET REL_LOCK)::TEVENT cpu (TSHARED(OATE(AT adt)))::l
  in match H_CalLock l' with Some _ => Some (adt { ... l' ...}) | None => None ...
```

```
Function palloc'_spec (n: Z) (adt: RData): option (RData * Z) :=
  match acquire_lock_AT_spec adt with
  | Some adt1 => match palloc_aux_spec n adt1 with
                 | Some (adt2, i) =>
                   match release_lock_AT_spec adt2 with
                   | Some adt3 => Some (adt3, i)
                   | _ => None end
                 | _ => None end
  | _ => None end.
```

Fig. 10: Specification for `palloc`

lock, and the specification for the critical section. The critical section spec is exactly the same in a sequential program: it does not modify the log, but instead only affects the `AT` field in the abstract data.

Then in a final, pure refinement step, we ascribe a high-level atomic specification `lpalloc_spec` to the `palloc` function. In this layer we no longer have any lock-related events at all, a call to `palloc` appends a single `OPALLOCE` event to the log. This is when we see the proof obligations related to liveness of the locks. Specifically, in order to prove the downwards refinement, we need to show that the call to `palloc'_spec` doesn't return `None`, so we need to show that `H_CalLock l'` is defined, so in particular the bound counter must not hit zero. By expanding out the definitions, we see that `palloc'_spec` takes a log `l` to `REL_LOCK :: (OATE (AT adt)) :: (TSHARED OPULL) :: (WAIT_LOCK 10) :: l`. The initial bound is 10, and there are two shared memory events, so the count never goes lower than 8. If a function modified more than one memory block there would be additional push- and pull-events, which could be handled by a larger initial bound.

Like all kernel-mode primitives in CertiKOS, the `palloc` function is total: if its preconditions are satisfied it always returns. So when verifying it, we show that all loops inside the critical section terminate. Through the machinery of bound numbers, this guarantee is propagated to the the while-loops inside the lock implementation: because all functions terminate, they can know that other CPUs will make progress and add more events to the log, and because of the bound number, they cannot add push/pull events forever. On the other hand, the framework completely abstract away how long time (in microseconds) elapses between any two event in the log.

Code reuse The same `acquire`/`release_lock` specifications can be used for all clients of the lock. The only proofs that need to be done for a given client is the refinement into abstracted primitives like `release_lock_AT_spec` (easy if we already have a sequential proof for the critical section), and the refinement proof for the atomic primitive like `lpalloc_spec` (which is very short). We never need to duplicate the thousands of lines of proof related to the lock algorithm itself.

Using more than one lock The layers approach is particularly nice when verifying code that uses more than one lock. To avoid deadlock, all functions must acquire the locks in the same order, and to prove the correctness the ordering must be included in the program invariant. We *could* do such a verification in a single layer, by having a single log with different events for the two locks, with the replay function being undefined if the events are out of order. But the layers approach provides a better way. Once we have ascribed an atomic specification to `palloc`, as above, all higher layers can use it freely without even knowing that the `palloc` implementation involves a lock. For example, some function in a higher layer could acquire a lock, allocate a page, and release the lock; in such an example the the order of the layers provides an order on the locks implicitly.

Proof Effort Among the whole proofs, the most challenging parts are the proofs for starvation freedom theorems like Thm. 7, and the functional correctness proofs for `mcs_acquire` and `mcs_release` functions in Sec. 4.3. The total lines of codes for starvation freedom is 2.5K lines, 0.6K lines for specifications, and 1.9k lines for proofs. This is because of the subtlety of those proofs. To prove the starvation freedom theorems and show the evidence of loop termination, lots of lemmas are required to express state changes by replaying the log. For instance, when `QS_CalLock(l) = Some(c1, c2, b, q, s, t)` and `q = nil`, `s` $= \varnothing$ and `t = nil`. It looks trivial in the hand-written proofs, but requires multiple lines of codes in the mechanized proof.

The total lines of codes for the low-level functional correctness of `mcs_acquire` and `mcs_release` are 3.2K lines, 0.7K lines for specifications, and 2.5K lines for proofs. It is much bigger than other code correctness proofs for while-loops in CertiKOS, because these loops do not have any explicit decreasing value. One another big part in our MCS Lock proofs is the proofs for Thm. 2 and the lines of code for this part is approximately 5K lines. The log replay function (`CalMCSLock`) always return the whole MCS Lock values (`MCSLock`) related to the `mcs_lock` structure defined in Fig. 1. In this sense, we always have to give the exact values for all memory chunks and prove the correspondence between the memory and the abstract data even the event associated with reading values (e.g. `GET_NEXT`). Hence, those proofs contain a lot of duplicate proofs for the memory access. However, they are quite straightforward and easy to produce. On top of that, we strongly believe that they can be easily reduced by introducing mechanized user-defined tactics later.

As an evaluation, we do not count the total lines of code in Coq for our entire MCS Lock module due to the two following reasons. First, our MCS Lock implementation is a part of CertiKOS. Therefore, our MCS Lock module also contains several definitions and proofs that are totally irrelevant to MCS Lock verification. This implies that counting the total lines of code for MCS Lock module has a high possibility of misinterpretation due to the lines of code for those definitions and proofs. Second, we intensively use contextual refinement approach to build the whole system rather than focusing on verifying the correctness and liveness of MCS Lock. Therefore, our proof efforts are mainly focus on proving MCS Lock that is able to be easily combined with multiple client codes rather than the efficient lock verification itself.

As can be seen from these line counts, proofs about concurrent programs have a huge ratio of lines of proof to lines of C code. If we tried to directly verify shared objects that use locks to perform more complex operations, like thread scheduling and inter-process communication, a monolithic proof would become much bigger than the current one, and would be quite unmanageable. The modular lock specification is essential here.

6 Related work and conclusions

Verified system software CertiKOS is an end-to-end verified concurrent system showing that its assembly code indeed "implements" (contextually simulates) the high-level specification. Other verified systems [10, 15, 26], are single-threaded, or use a per-core big kernel lock. The Verisoft team used VCC [2] to verify spinlocks in a hypervisor by directly postulating a Hoare logic rather than building on top of an operational semantics for C, and only proved properties about the low-level primitives rather than the full functionality of the hypervisor. By contrast, CertiKOS deals with the problem of formulating a specification in a way that can be used as one layer inside a large stack of proofs. As for CertiKOS itself, while we discussed the "local" verification of a single module, other papers explain how to relate the log and context to a more realistic nondeterministic machine model [8], how to "concurrently link" the per-CPU proofs into a proof about the full system [9], and how this extends to multiple threads per CPU [9].

Fine-grained concurrency The MCS algorithm uses low-level operations like CAS instead of locks. There is much research about how to reason about such programs, more than we have space to discuss here. One key choice is how much to prove. At least all operations should be linearizable [13] (a safety property). Some authors have considered mechanized verification of linearizability (e.g. [4, 6]), but on abstract transition system models, not directly on executable code. The original definition of linearizability instrumented programs to record a global history of method-invocation and method-return events. However, that's not a convenient theorem statement when verifying client code. Our formulation is closer to Derrick et al [4], who prove a simulation to a history of single atomic actions modifying abstract state. Going beyond safety, one also wants to prove a progress property such as wait-freedom [11] or (in our case) starvation-freedom [12].

Liang *et al* [19] showed that the linearizability and progress properties [12] for concurrent objects is exactly equivalent to various termination-sensitive versions of the contextual simulation property. Most modern separation-style concurrent logics [5,14,22–25] do not prove the same strong termination-sensitive contextual simulation properties as our work does, so it is unclear how they can be used to prove both the linearizability and starvation-freedom properties of our MCS Lock module. Total-TaDA [23] can be used to prove the total correctness of concurrent programs but it has not been mechanized in any proof assistant and there is no formal proof that its notion of liveness is precisely equivalent to Helihy's notion of linearizability and progress properties for concurrent objects [12]. FCSL [24] attempts to build proofs of concurrent programs in a "layered" way, but it does not address the liveness properties. Many of these program logics [14,25], however, support higher-order functions which our work does not address.

Other work on the MCS algorithm We are aware of two other efforts to apply formal verification methods to the MCS algorithm. Ogata and Futatsugi developed a mechanized proof using the UNITY program logic. [21] They work with an abstract transition system, not executable code. Like us, their correctness proof works by refinement (between a fine-grained and a more atomic spec) but they directly prove backward simulation.

One difference is that Ogata and Futatsugi's proof is done using a weaker fairness assumption. They assume "every CPU gets scheduled infinitely often", while we require a maximum scheduling period (F in Section 4.5). This is because we write our specification of `wait_lock` as a Coq function defined by recursion on a natural number, and all Coq functions must be total. So although our ultimate theorem only states that the method terminates "eventually", as an intermediate lemma we need to prove an explicit natural number bound on when a given call to `wait_lock` will finish. We could avoid this by e.g. using Coq's facilities to define functions by well-founded recursion, and making the termination measure M_i take ordinal instead of number values, but in practice assuming a fixed F seems like a reasonable model of multi-core concurrency.

The other MCS Lock verification we know of is by Liang and Feng [18], who define a program logic LiLi to prove liveness and linearizability properties and verify the MCS algorithm as one of their examples. The LiLi proofs are done on paper, so they can omit many "obvious" steps, and they work with a simple while-loop language instead of C. Many of the concepts in our proof are also recognizable in theirs. In their invariant and precondition they use specificational variables ta and tb (like `la` in Sec. 4.3), tl and S (like q and s in Sec. 4.4), and their termination measure $f(\mathfrak{G})$ includes the length of tl and the size of S (like M in Sec. 4.5). On the other hand, the fairness constant makes no appearance in $f(\mathfrak{G})$, because fairness assumptions are implicit in their inference rules.

A big difference between our work and LiLi is our emphasis on modularity. Between every two lines of code of a program in LiLi, you need to prove all the different invariants, down to low-level data representation in memory. In our development, these concerns are in different modules which can be completed by different programmers. Similarly, we aim to produce a stand-alone specification of the lock operations. In the LiLi example, the program being verified is an entire "increment" operation, which takes a lock, increments a variable and releases the lock. The pre/post-conditions of the code in the critical section includes the low-level implementation invariants of the lock, and the fact the lock will eventually be released is proved for the "increment" operation as a whole. Our locks are specified using *bound* numbers, so they can be used by many different methods.

Apart from modularity, one can see a more philosophical difference between the CertiKOS approach and program logics such as LiLi. Liang and Feng are constructing a program logic which is tailor-made precisely to reason about liveness properties under fair scheduling. To get a complete mechanized proof for a program in that setting would require mechanizing not only the proof of the program itself, but also the soundness proof for the logic, which is a big undertaking. Other parts of the program will favor other kinds of reasoning, for example many researchers have studied program logics with inference rules for reasoning about code *using* locks. One of the achievements of the CertiKOS style of specification is its flexibility, because the same model—a transition system with data abstraction and a log of events—works throughout the OS kernel. When we encountered a feature that required thinking about liveness and fairness, we were able to do that reasoning without changing the underlying logical framework.

Conclusion and Future Work Using the "layers" framework by Gu et al. [7] made our MCS lock proofs modular and reusable. It also lets us verify the code from end to end and extract certified executable code. Those proofs are also combined with client code using MCS Locks, which shows they can be used in a large scale system verification without increasing the complexity dramatically. In the future, we are planning to devise generic methods for building oracles, log replay functions, liveness proofs, and so on. We intend to generalize the machine model to handle weak memory models instead of assuming sequential consistency. And we also plan to apply this approach to other concurrent algorithms.

Acknowledgments

We would like to thank our anonymous referees for helpful feedbacks that improved this paper significantly. This research is based on work supported in part by NSF grants 1521523 and 1319671 and DARPA grants FA8750-12-2-0293, FA8750-16-2-0274, and FA8750-15-C-0082. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. S. Boyd-wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Ottawa Linux Symposium (OLS 2012)*, 2012.
2. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 23–42, 2009.
3. J. Corbet. Ticket spinlocks. https://lwn.net/Articles/267968/, February 2008.
4. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4:1–4:43, Jan. 2011.
5. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP'10*, pages 504–528, 2010.
6. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *Formal Techniques for Networked and Distributed Systems – FORTE 2004*, pages 97–114. Springer, 2004.
7. R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, pages 595–608, 2015.
8. R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjoberg, , and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.
9. R. Gu, Z. Shao, X. Wu, J. Kim, J. Koenig, T. Ramananandro, V. Sjoberg, H. Chen, and D. Costanzo. Language and compiler support for building certified concurrent abstraction layers. Technical Report YALEU/DCS/TR-1530, Dept. of Computer Science, Yale University, New Haven, CT, October 2016.
10. C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.
11. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.
12. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
13. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

14. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, pages 637–650, 2015.
15. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *SOSP'09: the 22nd ACM SIGOPS Symposium on Operating systems principles*, pages 207–220, 2009.
16. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
17. X. Leroy. A formally verified compiler back-end. *J. of Automated Reasoning*, 43(4):363–446, 2009.
18. H. Liang and X. Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 385–399, New York, NY, USA, 2016. ACM.
19. H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR*, pages 227–241, 2013.
20. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1), Feb. 1991.
21. K. Ogata and K. Futatsugi. Formal verification of the MCS list-based queuing lock. In *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, ASIAN '99, pages 281–293, London, UK, UK, 1999. Springer-Verlag.
22. P. D. R. Pinto, T. Dinsdale-Young, and P. Gardner. Tada: A logic for time and data abstraction. In *ECOOP'14*, pages 207–231, 2014.
23. P. D. R. Pinto, T. Dinsdale-Young, P. Gardner, and J. Sutherland. Modular termination verification for non-blocking concurrency. In *ESOP'16*, pages 176–201, 2016.
24. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI'15*, pages 77–87, 2015.
25. A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356, 2013.
26. J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. 2010 ACM Conference on Programming Language Design and Implementation*, pages 99–110, 2010.