DOI:10.1145/3356903

Building Certified Concurrent OS Kernels

By Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo

Abstract

Operating system (OS) kernels form the backbone of system software. They can have a significant impact on the resilience and security of today's computers. Recent efforts have demonstrated the feasibility of formally verifying simple general-purpose kernels, but they have ignored the important issues of concurrency, which include not just user and I/O concurrency on a single core, but also multicore parallelism with fine-grained locking. In this work, we present CertiKOS, a novel compositional framework for building verified concurrent OS kernels. Concurrency allows interleaved execution of programs belonging to different abstraction layers and running on different CPUs/threads. Each such layer can have a different set of observable events. In CertiKOS, these layers and their observable events can be formally specified, and each module can then be verified at the abstraction level it belongs to. To link all the verified pieces together, CertiKOS enforces a so-called contextual refinement property for every such piece, which states that the implementation will behave like its specification under any concurrent context with any valid interleaving. Using CertiKOS, we have successfully developed a practical concurrent OS kernel, called mC2, and built the formal proofs of its correctness in Coq. The mC2 kernel is written in 6500 lines of C and x86 assembly and runs on stock x86 multicore machines. To our knowledge, this is the first correctness proof of a general-purpose concurrent OS kernel with fine-grained locking.

1. INTRODUCTION

Operating system (OS) kernels and hypervisors form the backbone of safety-critical software systems. Hence, it is highly desirable to verify the correctness of these programs formally. Recent efforts[5, 6, 10, 13, 17] have shown that it is feasible to formally prove the functional correctness of simple general-purpose kernels, file systems, and device drivers. However, none of these systems have addressed the important issues of concurrency,[2] such as not only user and I/O concurrency on a single CPU but also multicore parallelism with fine-grained locking. This severely limits the applicability of today's formally verified system software.

What makes the verification of concurrent OS kernels so challenging? First, concurrent kernels allow interleaved execution of kernel/user modules belonging to different abstraction layers; they contain many interdependent components that are difficult to untangle. Several researchers[22, 23] believe that the combination of fine-grained concurrency and the kernels' functional complexity makes formal verification intractable, and even if it is possible, its cost would far exceed that of verifying a sequential kernel.

Second, concurrent kernels need to make all three types of concurrency (i.e., user, I/O, and multicore) coherently work together. User and I/O concurrency are difficult to reason about because they rely on thread yield/sleep/wakeup primitives or interrupts to switch control and support synchronization but still provide the illusion that each user process is executed uninterruptedly and sequentially. Multicore concurrency with fine-grained locking may utilize sophisticated spinlock implementations such as MCS locks[21] that are also hard to verify.

Third, concurrent kernels may also require that some of their system calls eventually return, but this depends on the progress of the concurrent primitives used in the kernels. Formally proving starvation-freedom[15] for concurrent objects only became possible recently.[20] Standard Mesa-style condition variables (CV)[18] do not enforce starvation-freedom; this can be fixed by storing CVs in a FIFO queue. But the solution is not trivial, and even the popular, most up-to-date OS textbook,[2] has gotten it wrong.

Fourth, given the high cost of building certified c oncurrent kernels, it is important that these kernels can be quickly adapted to support new hardware platforms and applications.[3] However, if we are unable to model the interference among different components in an extensible way, even a small change to the kernel could incur a huge reverification overhead.

In this paper, we present CertiKOS, a compositional framework that tackles all these challenges. We believe that, to control the complexity of concurrent kernels and to prove a strong support of extensibility, we must first have a *compositional* specification that can untangle *all* the kernel interdependencies and encapsulate interference among different kernel objects. Because the very purpose of an OS kernel is to build layers of abstraction over bare machines, we insist on uncovering and specifying these layers, and then verifying each kernel module at the abstraction level it belongs to.

The functional correctness of an OS kernel is often stated as a *refinement*—that is, the behavior of the C/assembly implementation of a kernel K is fully captured by its abstract functional specification S. Of course, the ultimate goal of having a certified kernel is to reason about programs

The original version of this paper is entitled "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels" and was published in the *Proceedings of 12th USENIX Symposium on Operating System Design and Implementation*, 2016, 653–669

running on top of (or along with) the kernel. It is thus important to ensure that given any kernel extension or user program P, the combined code $K \oplus P$ also refines $S \oplus P$. If this fails to hold, the kernel is functionally incorrect as P can observe some behavior of K that does not satisfy S.

In the concurrent setting, such a *contextual refinement* property must hold not only for any context program P but also for any *environment* context ε. When focusing on some thread set, each ε defines a specific instance on how other threads/CPUs respond to this thread set. With shared-memory concurrency, interference between ε and the focused thread set is both necessary and common.

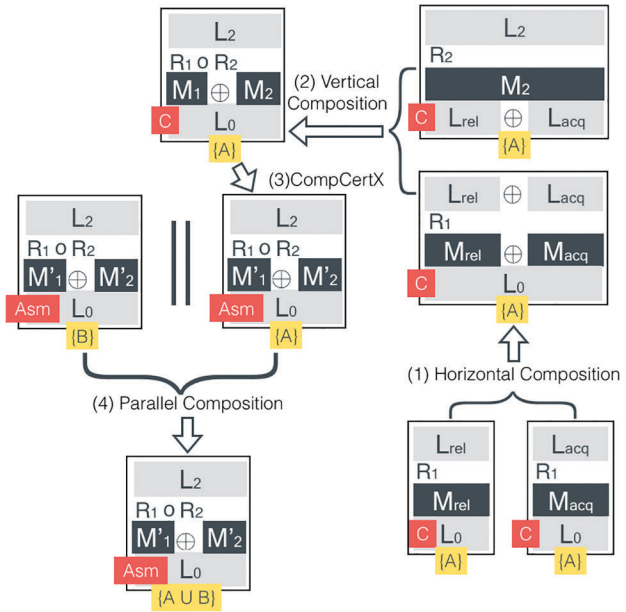
In CertiKOS, we introduce *certified concurrent abstraction layers* to state such contextual refinement properties (see Figure 1). Each abstraction layer, parameterized over some specific ε, is an *assembly-level machine* extended with a particular set of abstract objects, that is, abstract states plus atomic primitives. These layers enable modular verification and can be composed in several manners. Later in Section 3, we show how the use of ε at each layer allows us to verify concurrent programs using standard techniques for verifying sequential programs. Indeed, most of our kernel components are written in a variant of C (called ClightX[10]) and verified at the C level. These certified C layers can be compiled and linked together into certified assembly layers using CompCertX[10, 12]—a *thread-safe* version of the CompCert compiler.[19] Thus, under CertiKOS, an otherwise prohibitive verification task can be decomposed into many simple and easily automatable ones, and proven global properties can be propagated down to the assembly level.

Using CertiKOS, we have successfully developed a fully certified concurrent OS kernel mC2 in the Coq proof assistant. The mC2 kernel consists of 6500 lines of C and x86 assembly, supports both fine-grained locking and thread yield/sleep/wakeup primitives, and can run on stock x86 multicore machines. mC2 can also double as a hypervisor and boot multiple instances of Linux in guest virtual machines (VM) running on different CPUs. It guarantees not only functional correctness, that is, the mC2 kernel implementation satisfies its system-call specification, but also liveness property, that is, all system calls will eventually return. The entire proof effort for supporting concurrency took less than two person-years. To the best of our knowledge, mC2 is the first fully verified general-purpose concurrent OS kernel with fine-grained locking.

2. OVERVIEW OF OUR APPROACH

In this section, to illustrate our layered techniques, we will walk through a small example (see Figure 1) that uses a lock to protect a critical section. In this example, client program P has two threads running on two different CPUs; each thread makes one call to primitive foo provided by concurrent layer interface L_2. Interface L_2 is implemented by concurrent module M_2, which in turn is built on top of interface L_1. Method foo calls two primitives f and g in a critical section protected by a lock. The lock is implemented over interface L_0 using the ticket lock algorithm[21] in module M_1. The lock maintains two integer variables n (the "now serving" ticket number) and t (the "next" ticket number). Lock-acquire method acq fetches and increments the next ticket number (by FAI_t) and spins until the fetched number is served. Lock-release

Figure 1. The certified (concurrent) abstraction layer, $L_0 \vdash_{R_1} M_{acq} : L_{acq}$, is a predicate plus its mechanized proof object showing that the implementation of the ticket lock acquire M_{acq} running on the underlay interface L_0 indeed faithfully implements the desirable overlay interface L_{acq}. The implementation M_{acq} is written in C, whereas the interfaces L_0 and L_{acq} are written in Coq. The implementation relation is denoted as R_1. This layer can be (1) horizontally composed with another layer (e.g., the lock release operation) if they have identical state views (i.e., with the same R_1) and are based on the same underlay interface L_0. The composed layer can also be (2) vertically composed with another layer that relies on its overlay interface. Certified C layers can be compiled into certified assembly layers using our (3) CompCertX compiler. In the concurrent setting, these layers can also be (4) composed in parallel.




```
1  struct ticket_lock {
2    volatile uint n, t;
3  };
4  //Methods provided by L0
5  extern uint get_n();
6  extern void inc_n();
7  extern uint FAI_t();
8  extern void f();
9  extern void g();
10 extern void pull();
11 extern void push();
12 //M1 module
13 void acq () {
14   uint my_t = FAI_t();
15   while(get_n()!=my_t){};
16   pull();
17 }
18 void rel () {
19   push();
20   inc_n();
21 }
22 //Methods provided by L1
23 extern void acq();
24 extern void rel();
25 extern void f();
26 extern void g();
27 //M2 module
28 void foo () {
29   acq();
30   f(); g();
31   rel();
32 }
33
34 //Methods provided by L2
35 extern void foo();
36
37 //Client program P
38 //Thread running on CPU 1
39 void T1 () { foo(); }
40 //Thread running on CPU 2
41 void T2 () { foo(); }
```

method rel simply increments the "now serving" ticket number by inc_n. These primitives are provided by L_0 and implemented using x86 atomic instructions. Interface L_0 also provides primitives f and g that are later passed on to L_1, as well as *ghost* primitives pull and push that logically mark the acquisition and release of locks. Such ghost primitives only help the verification process and are not needed for the program to execute.

Here, the concurrent layer interface (e.g., L_0) provides a set of primitives that can be invoked at this level and uses *events* to capture primitives' effects that are *visible* to other CPUs/threads. For example, event 1.FAI_t represents the invocation of FAI_t by CPU 1. In this way, one execution of a concurrent program running on a layer machine can be *specified* by a sequence of events, which we call a *logical log*. For example, if two CPUs are executed in the order 1–2–2–1–1–2–1–2–1–1–1–2–2, running program P (see Figure 1) over the layer machine of L_0 generates the log:

$$\begin{array}{l}\boxed{1.\text{FAI\_t}} \bullet \boxed{2.\text{FAI\_t}} \bullet \boxed{2.\text{get\_n}} \bullet \boxed{1.\text{get\_n}} \bullet \boxed{1.\text{pull}} \bullet \boxed{2.\text{get\_n}} \\ \bullet \boxed{1.\text{f}} \bullet \boxed{2.\text{get\_n}} \bullet \boxed{1.\text{g}} \bullet \boxed{1.\text{push}} \bullet \boxed{1.\text{inc\_n}} \bullet \boxed{2.\text{get\_n}} \bullet \boxed{2.\text{pull}}\end{array} \quad (2.1)$$

Thus, a concurrent module M over L can be specified by how M produces events (provided by L). M can then be verified by building a *certified abstraction layer*, $L \vdash_R M : L'$, stating that the events generated by M over L are fully captured by the desirable interface L'. Note that the events provided by L and L' might not be exactly the same, and the relation between events at different layers is denoted as R.

Take the lock-acquire implementation M_{acq} in Figure 1 as an example. The goal is to prove that $L_0 \vdash_{\mathbf{id}} M_{acq} : L_{acq}$ holds with an identical relation **id** (between events at two layers), where the events generated by L_{acq} (on behalf of thread t) satisfy the pattern:

$$\ldots \bullet \boxed{\text{t.FAI\_t}} \bullet \ldots \bullet \boxed{\text{t.get\_n}} \bullet \ldots \bullet \boxed{\text{t.get\_n}} \bullet \ldots \bullet \boxed{\text{t.get\_n}} \bullet \ldots \bullet \boxed{\text{t.pull}} \bullet \ldots$$

Events generated by other threads (or CPUs) are omitted here.

To achieve modular verification, we parameterize each layer interface L with an *active* thread set A, and then carefully define its set of valid *environment contexts*, denoted as EC(L, A). Each environment context ε captures a specific instance—from a particular run—of the list of events that other threads or CPUs (not in A) return when responding to the events generated by threads in A. We can then define a new *thread-modular* machine $\Pi_{L(A)}(P, \varepsilon)$ that will operate like the usual assembly machine when P switches control to threads in A, but will only obtain the list of events from the environment context ε when P switches control to threads outside A. Here, we use $L(A)$ to denote the layer interface with an active thread set A that consists the same set of abstract objects with L.

Note that if A is a singleton, for each ε, $\Pi_{L(A)}$ behaves like a sequential machine: it first queries ε for the events generated by other threads, and then executes the next instruction of the active thread. We use ▶ to denote a query to ε. The lock-acquire function, on behalf of thread t, can be specified in $L_{acq}(\{t\})$ as:

$$\blacktriangleright\boxed{\text{t.FAI\_t}} \blacktriangleright \boxed{\text{t.get\_n}} \blacktriangleright \boxed{\text{t.get\_n}} \blacktriangleright \boxed{\text{t.get\_n}} \ldots \blacktriangleright \boxed{\text{t.pull}} \quad (2.2)$$

In this model, other threads' behaviors and the potential interleaving are encapsulated into those queries to ε. We can then verify module M_{acq} as it were sequential:

$$L_0(\{t\}) \vdash_{\mathbf{id}} M_{acq} : L_{acq}(\{t\})$$

By verifying that the lock-release function M_{rel} also meets its specification L_{rel}, we can apply the *horizontal composition* rule to obtain the composed layer (where we use L_1' to denote $L_{acq} \oplus L_{rel}$):

$$L_0(\{t\}) \vdash_{\mathbf{id}} M_{acq} \oplus M_{rel} : L_1'(\{t\}) \quad (2.3)$$

If every valid environment context $\varepsilon \in \mathrm{EC}(L_1', \{t\})$ guarantees that the loop of get_n in thread t terminates, we can lift $L_1'(\{t\})$ to a higher level layer interface $L_1(\{t\})$, which specifies the lock-acquire as ▶t.acq. We use R_1 to denote the relation between the events of $L_1'(\{t\})$ and $L_1(\{t\})$, for example, t.acq is mapped to the event sequence in (2.2). We can prove the following certified layer:

$$L_1'(\{t\}) \vdash_{R_1} \varnothing : L_1(\{t\}) \quad (2.4)$$

where Ø states that no code is involved at this step. By applying the *vertical composition* rule to (2.3) and (2.4), we have that:

$$L_0(\{t\}) \vdash_{\mathbf{id} \circ R_1} M_{acq} \oplus M_{rel} : L_1(\{t\})$$

With our new compositional layer semantics, these "per-thread" certified layers can be soundly composed in parallel when their *rely conditions* (i.e., the constraints to environmental interference) are compatible with each other. For example, we can also derive the certified layer for the ticket lock on behalf of some thread $t'(\neq t)$. By showing that the events generated by t' belong to EC(L_1, $\{t\}$) and vice versa, we can apply the *parallel composition* rule to derive:

$$L_0(\{t, t'\}) \vdash_{\mathbf{id} \circ R_1} M_{acq} \oplus M_{rel} : L_1(\{t, t'\})$$

Any observable behavior of running P with $M_{acq} \oplus M_{rel}$ (denoted as M_1 in Figure 1) over $L_0(\{1, 2\})$ can be captured by running P directly on top of $L_1(\{1, 2\})$. For example, the behavior in (2.1) can be captured by the following log over $L_1(\{1, 2\})$:

$$\boxed{1.\text{acq}} \bullet \boxed{1.\text{f}} \bullet \boxed{1.\text{g}} \bullet \boxed{1.\text{rel}} \bullet \boxed{2.\text{acq}}$$

Based on the layer interface $L_1(\{t\})$, we can continue verifying that the module M_2 satisfies a higher level interface $L_2(\{t\})$, where foo is specified as ▶t.foo. The relation between these two layer interfaces maps the event t.foo of $L_2(\{t\})$ into the event sequence t.acq • t.f • t.g • t.rel of $L_1(\{t\})$. As the primitive foo is specified by a single event, we call it an *atomic object*. The observable behaviors of running P over the layer machine $L_2(\{1, 2\})$ consist of only two logs: 1.foo • 2.foo and 2.foo • 1.foo.

In this way, we can decompose our mC2 kernel K into many modules and verify them at the layer interfaces they belong to, as if there were only a single active, sequential thread. These per-thread layers (whose topmost layer

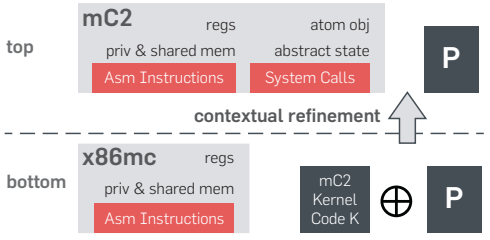
interface is L_{mC2}) can be composed into per-CPU layers and then further combined into a single multicore machine (see Section 3 and Figure 5). We use x86mc to denote this assembly-level multicore machine, $[\![\cdot]\!]_{x86mc}$ to denote the whole-machine semantics for x86mc, and $[\![\cdot]\!]_{mC2}$ to denote the machine semantics equipped with the topmost layer interface. The composed certified layers imply the *contextual refinement* property:

$$\forall P,\ [\![K \oplus P]\!]_{x86mc} \sqsubseteq [\![P]\!]_{mC2}$$

which says that, for any context user program P, the observable behaviors of running P together with K over the multicore machine x86mc are fully captured by running P directly over L_{mC2} (see Figure 2). We call L_{mC2} a *deep specification*[10] of K over x86mc, because there is no need to ever look at K again; any property about K over x86mc can be proved using L_{mC2} alone.

Overview of the mC2 kernel. Figure 3 shows the system architecture of mC2. The mC2 system was initially developed in the context of a large DARPA-funded research project. It is a concurrent OS kernel that can also double as a hypervisor. It runs on an unmanned ground vehicle (UGV) with a multicore Intel Core i7 machine. On top of mC2, we run three Ubuntu Linux systems as guests (one each on the first three cores). Each virtual machine runs several robot architecture definition language (RADL) nodes that have fixed hardware capabilities such as access to GPS, radar, etc. The kernel also contains a few simple device drivers (e.g., interrupt controllers, serial and keyboard devices). More complex devices are either supported at the user level, or passed through (via IOMMU) to various guest Linux VMs. By running different RADL nodes in different VMs, mC2 provides strong isolation so that even if attackers take control of one VM, they still cannot break into other VMs and compromise the overall mission of the UGV.

Figure 2. The contextual refinement property that has been proved for mC2.

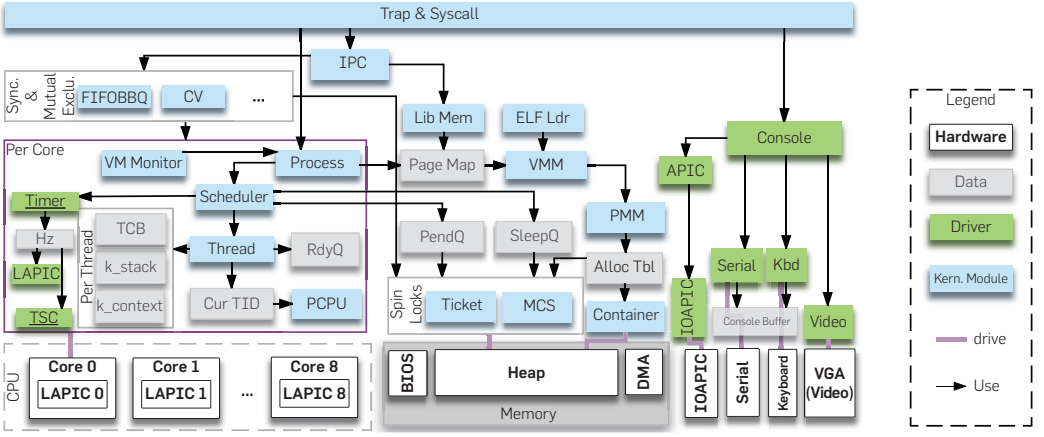


What have we proved? Using CertiKOS, we have successfully built a fully certified version of the mC2 kernel and proved its contextual refinement property with respect to a high-level deep specification for mC2. This functional correctness property implies that all system calls and traps will always strictly follow high-level specifications, run *safely*, and eventually *terminate*; there will be no data race, no code injection attacks, no buffer overflows, no null pointer access, no integer overflow, etc.

More importantly, because for any program P, we have $[\![K \oplus P]\!]_{x86mc}$ refines $[\![P]\!]_{mC2}$, we can also derive the *behavior equivalence* property for P, that is, whatever behavior a user can deduce about P based on the high-level specification for the mC2 kernel K, the actual linked system $K \oplus P$ running on the concrete x86mc machine would indeed behave exactly as expected. All global properties proven at the system-call specification level can be propagated down to the lowest assembly machine.

Assumptions and limitations. The mC2 kernel is not as comprehensive as real-world kernels such as Linux. For example, mC2 currently lacks a certified storage system. The main goal of this work is to show that it is feasible to build certified concurrent kernels with fine-grained locking. We

Figure 3. The mC2 hypervisor kernel contains various shared objects such as spinlock modules (Ticket, MCS), sleep queues (SleepQ, for implementing queuing locks and condition variables), pending queues (PendQ, for waking up a thread on another CPU), container-based physical and virtual memory management modules (Container, PMM, VMM), a Lib Mem module (for implementing shared-memory IPC), synchronization modules (FIFOBBQ, CV), and an IPC module. Within each core (the purple box), we have the per-CPU scheduler, the kernel-thread management module, the process management module, and the virtualization module (VM Monitor). Each kernel thread has its own thread-control block (TCB), context, and stack.



did not try to incorporate all the latest advances for multicore kernels into mC2.

Regarding specification, there are 450 lines of Coq code (LOC) to specify the system calls (the topmost layer interface; see Table 1) and 943 LOC to specify the x86 hardware machine model (the bottommost layer interface). These are in our trusted computing base. We keep them small to limit the room for errors and ease the review process.

Our assembly machine assumes strong sequential consistency for all atomic instructions. We believe our proof should remain valid for the x86 TSO model because (1) all our concurrent layers guarantee that nonatomic memory accesses are properly synchronized; and (2) the TSO order guarantees that all atomic synchronization operations are properly ordered. Nevertheless, more formalization work is needed to turn our proofs over sequential-consistent machines into those over the TSO machines.[23]

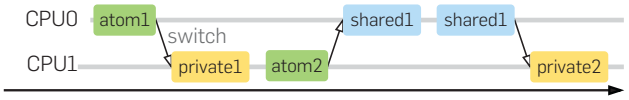
Also, our machine model only covers a small portion of the x86 hardware features and cannot be used to verify some kernel components, such as a bootloader, a PreInit module (which initializes the CPUs and the devices), an ELF loader, and some device drivers (e.g., disk driver). Their verification is left for future work.

We also trust the Coq proof checker and the CompCertX assembler for converting assembly into machine code.

3. CONCURRENT LAYER MACHINES

In this section, we explain the concurrent layer design principles, and show how to introduce per-CPU layer interfaces, based on a multicore hardware machine model.

Π_{x86mc} **multicore hardware model** allows arbitrary interleavings at the level of *assembly instructions*. At each step, the hardware *nondeterministically* picks one CPU and executes the next assembly instruction on that CPU. Each assembly instruction is classified as *atomic*, *shared*, or *private*, depending on the memory it accesses. One interleaving of an example program running on two CPUs is:



The memory locations are *logically* categorized into two kinds: the ones *private* to a single CPU/thread and the ones *shared* by multiple CPUs/threads. Private memory accesses do not need to be synchronized, whereas nonatomic shared memory accesses need to be protected by some synchronization mechanisms (e.g., locks), which are normally implemented using atomic instructions (e.g., fetch-and-add). With proper protection, each shared memory operation can be viewed as if it were atomic.

The *atomic object* is an abstraction of some segment of well-synchronized shared memory, combined with operations that can be performed over that segment. It consists of a set of primitives, an initial state, and a *logical log* containing the entire history of the operations that were performed on the object during an execution schedule. Each primitive invocation records a *single* corresponding event in the log. For example, the above interleaving produces the logical log $\boxed{0.\text{atom1}} \bullet \boxed{1.\text{atom2}}$. We require that these events contain enough information so we can derive the current state of each atomic object by *replaying* the entire log over the object's initial state.

As shown in Figure 4, a *concurrent layer interface* contains both *private objects* (e.g., O_i) and *atomic objects* (e.g., O_j), along with some invariants imposed on them. These objects are verified by building certified concurrent layers via forward simulations, which imply strong *contextual refinement* relations:

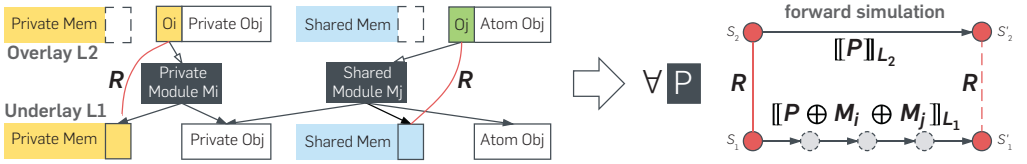
DEFINITION 1 (CONTEXTUAL REFINEMENT). *We say that machine* Π_{L_1} contextually refines *machine* Π_{L_2} *(written as* $\forall P, [\![P]\!]_{L_1} \sqsubseteq [\![P]\!]_{L_2}$*), if, and only if, for any P that does not get stuck on* Π_{L_2}*, we also have that (1) P does not get stuck on* Π_{L_1}*; and (2) any observable behavior of P on* Π_{L_1} *is also observed on* Π_{L_2}.

However, proving such contextual refinements directly on a multicore, nondeterministic hardware model is difficult

Table 1. Verified system calls of the mC2 hypervisor kernel.

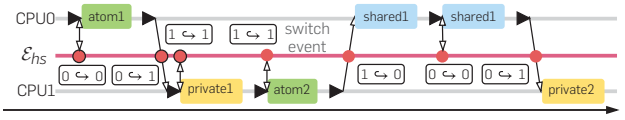
kernel_init, get_quota, send, recv, rz_spawn, spawn, sleep, yield, wakeup, kill, getc, putc, get_tsc_per_ms, get_curid, vm_exit_info, vm_mmap, vm_set_seg, vm_get_reg, vm_set_reg, vm_get_next_eip, vm_inject_event, vm_check_int_shadow, vm_run, vm_check_pending_event, vm_intercept_int_window, vm_get_tsc_offset, vm_set_tsc_offset, vm_rdmsr, vm_wrmsr

Figure 4. The overlay interface L_2 is a more abstract interface, built on top of the underlay interface L_1, and implemented by private module M_i and shared module M_j. Private objects in L_2 only access the private memory of L_1. Atomic objects are implemented by shared modules (e.g., M_{acq} in Figure 1) that may access lower-level atomic objects (e.g., FAI_t), private objects, and shared memory. Memory regions of L_1 accessed by the layer implementation are hidden and replaced by newly introduced objects of L_2. The simulation relation R is defined between these memory regions and objects, for example, R_1 in Section 2. Then, the certified concurrent layer $L_1 \vdash_R M_i \oplus M_j : L_2$ can be built by proving the forward simulation: whenever two states s_1, s_2 are related by R, and running any P over the layer machine based on L_2 takes s_2 to s_2' in one step, then there exists s_1' such that running $P \oplus M_i \oplus M_j$ over L_1 takes s_1 to s_1' in multiple steps, and s_1' and s_2' are also related by R.



because we must consider all possible interleavings. In the rest of this section, we show how to gradually refine this hardware model into a more abstract one that is suitable for reasoning about concurrent code in a CPU-local fashion.

Π_{hs}: machine model with hardware scheduler. By parameterized with a *hardware scheduler* ε_{hs} that specifies a particular interleaving for an execution, the machine model Π_{hs} becomes *deterministic*. To take a program from Π_{x86mc} and run it on top of Π_{hs}, we insert a *logical switch point*, denoted as ▶, before each assembly instruction. At each switch point, the machine first queries ε_{hs} and gets the CPU ID that will execute next. All the *switch decisions* made by ε_{hs} are stored in the logical log state as switch events, for example, $i \hookrightarrow j$ denotes a switch event from CPU i to j. The previous example on Π_{x86mc} can then be simulated on Π_{hs} by the following ε_{hs}:



The log recorded by this execution is as follows:

$$0 \hookrightarrow 0 \bullet \texttt{0.atom}_1 \bullet 0 \hookrightarrow 1 \bullet 1 \hookrightarrow 1 \bullet 1 \hookrightarrow 1 \bullet \texttt{1.atom}_2 \bullet 1 \hookrightarrow 0 \bullet 0 \hookrightarrow 0 \bullet 0 \hookrightarrow 1$$

The behavior of running a program P over this machine with a particular ε_{hs} is the generated log denoted as $\Pi_{hs}(P, \varepsilon_{hs})$. We write EC_{hs} to represent the set of all possible hardware schedulers. Then, the whole-machine semantics can be defined as a set of logs:

$$[\![P]\!]_{hs} = \{\Pi_{hs}(P, \varepsilon_{hs}) \mid \varepsilon_{hs} \in \mathrm{EC}_{hs}\}$$

To ensure the correctness of this machine model, we prove that it is *contextually refined* by the hardware model Π_{x86mc}:

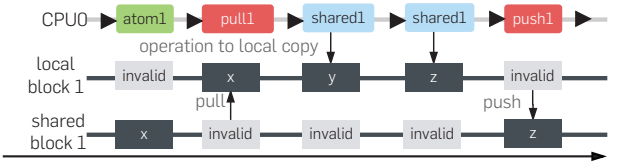
Lemma 1 (Correctness of Π_{hs}). $\forall P, [\![P]\!]_{x86mc} \sqsubseteq [\![P]\!]_{hs}$

Π_{lcm}: machine with local copies of the shared memory. To enforce that shared memory accesses are well synchronized, we introduce a new machine model (Π_{lcm}) that equips each CPU with local copies of shared memory blocks along with *valid bits*. The relation between CPU's local copies and the global shared memory is maintained through two new *ghost* primitives, pull and push.

The pull operation over a particular CompCert-style memory block[19] updates a CPU's local copy of that block to be equal to the one in the shared memory, marking the local block as valid and the shared version as invalid. Conversely, the push operation updates the shared version to be equal to the local block, marking the shared version as valid and the local block as invalid.

If a program tries to pull an invalid shared memory block or push/access an invalid local block, the program gets stuck. We make sure that every shared memory access is always performed on its valid local copy, thus systematically enforcing valid accesses to the shared memory. Note that all of these constructions are completely *logical* and do not introduce any performance overhead.

The shared memory updates of the previous example can be simulated on Π_{lcm} as follows:



Among each shared memory block and all of its local copies, only one can be valid at any moment of the machine execution. Therefore, for any program P with a potential *data race*, there exists a hardware scheduler such that P gets stuck on Π_{lcm}. By showing that a program P is safe (never gets stuck) on Π_{lcm} for *all possible* hardware schedulers, we guarantee that P is data-race free.

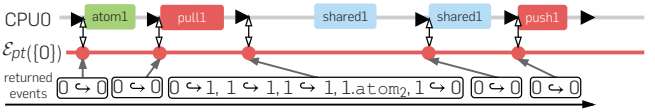
We have shown (in Coq) that Π_{lcm} is correct with respect to the previous machine model Π_{hs} with the EC_{hs}:

Lemma 2 (Correctness of Π_{LCM}). $\forall P, [\![P]\!]_{hs} \sqsubseteq [\![P]\!]_{lcm}$

Π_{pt}: partial machine with environment context. To achieve local reasoning, we introduce a partial machine model Π_{pt} that can be used to reason about the programs running on a subset of CPUs, by parametrizing the model over the behaviors of an *environment context*, that is, the rest of the CPUs.

We call a given local subset of CPUs the *active CPU set* (denoted as A). The partial machine model is configured with an active CPU set and it queries the environment context whenever it reaches a switch point that attempts to switch to a CPU outside the active set.

The set of environment contexts for A in this machine model is denoted as $\mathrm{EC}(pt, A)$. Each environment context $\varepsilon_{pt(A)} \in \mathrm{EC}(pt, A)$ is a *response function*, which takes the current log and returns a list of events from the context programs, that is, those outside of A. The response function simulates the observable behavior of the context CPUs and imposes some invariants over the context. The hardware scheduler is also a part of the environment context. In other words, the events returned by the response function also include switch events. The execution of CPU 0 in the previous example can be simulated with an $\varepsilon_{pt(\{0\})}$ function:



For example, at the third switch point, $\varepsilon_{pt(\{0\})}$ returns the event list $0 \hookrightarrow 1 \bullet 1 \hookrightarrow 1 \bullet 1 \hookrightarrow 1 \bullet \texttt{1.atom}_2 \bullet 1 \hookrightarrow 0$.

Suppose we have verified that two programs, separately running with two *disjoint* active CPU sets A and B, produce event lists satisfying invariants INV_A and INV_B, respectively. If INV_A is consistent with the environment-context invariant of B, and INV_B is consistent with the environment-context invariant of A, then we can compose the two separate programs into a single program with active set $A \cup B$. This combined program is guaranteed to produce event lists satisfying the combined invariant $\mathrm{INV}_A \wedge \mathrm{INV}_B$. Using the machine semantics as a set of produced logs, this composition can then be defined as a contextual refinement:

Lemma 3 (Composition of partial machines).

$$\forall P, [\![P]\!]_{pt(A\cup B)} \sqsubseteq [\![P]\!]_{pt(A)} \cap [\![P]\!]_{pt(B)} \qquad if\ A \cap B = \varnothing$$

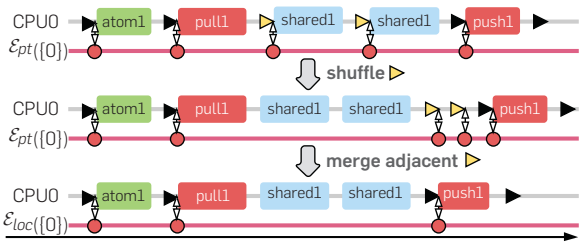
After composing the programs on all CPUs, the context CPU set becomes empty and the composed invariant holds on the whole machine. As there is no context CPU, the environment context is reduced to the hardware scheduler, which only generates the switch events. In other words, letting C be the entire CPU set, we have that $EC(pt, C) = EC_{hs}$. Thus, we can show that this *composed machine* with the entire CPU set C is refined by Π_{lcm}:

Lemma 4 (Correctness of Π_{PT}). $\forall P, [\![P]\!]_{lcm} \sqsubseteq [\![P]\!]_{pt(C)}$

Π_{loc}: CPU-local machine model. If we focus on a single active CPU i, the partial machine model provides a sequential-like interface configured with an environment context representing all other CPUs. However, in this model, there is a switch point before each instruction, so program verification still needs to handle many unnecessary interleavings, for example, those between private operations. Thus, we introduce a CPU-local machine model (denoted as Π_{loc}) for a CPU i, in which switch points only appear before atomic or push/pull operations. The switch points before shared or private operations are removed via two steps: *shuffling* and *merging*.

Every switch point before a shared or private operation can be shuffled to the front of the next atomic operation by introducing a *log cache*. For such switch points, query results from the environment context are stored in the log cache. The cached events are applied to the logical log just before the next atomic or push/pull operations. This is sound because a shared operation can only be performed when the current local copy of shared memory is valid, meaning that no other context program can interfere with the operation.

Once the switch points are shuffled properly, we merge all the adjacent switch points together. When we merge switch points, we also need to merge the switch events generated by the environment context. For example, the change of switch points for the previous example on CPU-local machine is as follows:



Lemma 5 (Correctness of Π_{LOC}).

$$\forall P, [\![P]\!]_{pt(\{i\})} \sqsubseteq [\![P]\!]_{loc(\{i\})}$$

Finally, we obtain the refinement relation from the multicore hardware model to the CPU-local machine model by composing all of the refinement relations together (see Figure 5).

We introduce and verify the mC2 kernel on top of the CPU-local machine model Π_{loc}. The refinement proof guarantees that the proven properties can be propagated down to the multicore hardware model Π_{x86mc}.

All our proofs (such as every step in Figure 5) are implemented, composed, and machine-checked in Coq. Each refinement step is implemented as a CompCert-style upward-forward simulation from one layer machine to another. Each machine contains the usual (CPU-local) abstract state, a logical global log (for shared state), and an environment context. The simulation relation is defined over these two machine states, and matches the informal intuitions given in this and next sections.

4. CERTIFYING THE mC2 KERNEL

Based on the CPU-local layer machine model Π_{loc}, the certified mC2 kernel can be built by introducing a series of logical abstraction layers and decomposing the otherwise complex verification tasks into a large number of small tractable ones.

In the mC2 kernel, the preinitialization module forms the bottom layer machine that connects to Π_{loc}, instantiated with a particular *active CPU c*. The trap handler forms the top layer machine that provides system call interface and serves as a specification to the whole kernel, instantiated with a particular active thread running on that active CPU c. Our main theorem states that any global properties proved at the topmost layer machine can be propagated down to the lowest hardware machine. In this section, we explain selected components in more detail.

The preinitialization layer machine defines some x86 hardware behaviors, such as page walking upon memory load (when paging is turned on), saving and restoring the trap frame in the case of interrupts and exceptions (e.g., page fault), and exchanging data between devices and memory. The hardware memory management unit (MMU) is modeled in a way that mirrors the paging hardware (see Figure 6a). When paging is enabled, memory accesses made by both

Figure 5. Contextual refinement between concurrent layer machines.

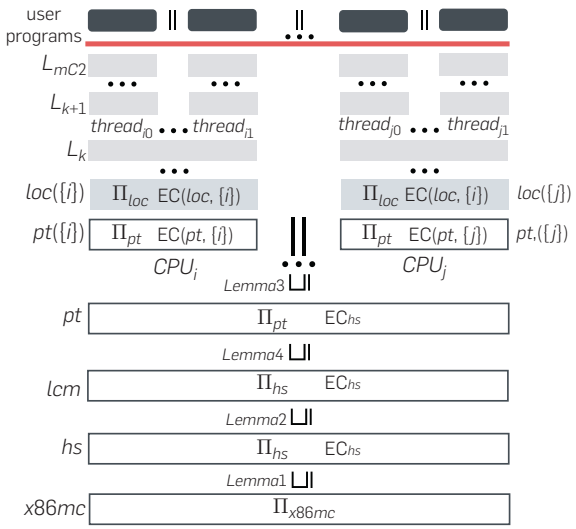
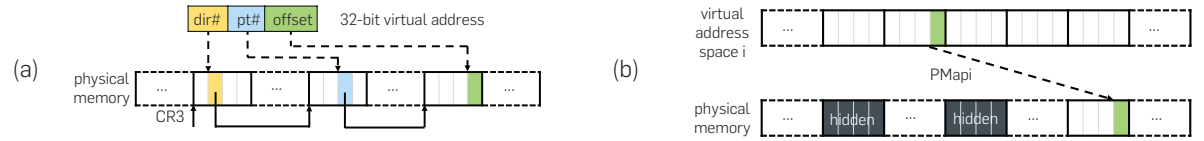


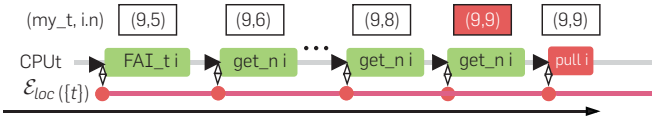
Figure 6. (a) Hardware MMU using two-level page map; (b) virtual address space *i* set up by page map .



the kernel and the user programs are translated using the page map pointed to by the register CR3. When page faults occur, the fault information is stored in CR2 and the page fault handler is triggered.

Spinlock module provides fine-grained lock objects as the base of synchronization mechanisms. Figure 1 shows one spinlock implementation using the ticket lock algorithm. It depends on an *atomic ticket object* consisting of two fields: next ticket number t and now-serving ticket number n. In mC2, we introduce an array of ticket objects; each of them (identified by a specific lock index i) can be used to protect a segment of shared memory. The ticket objects can only be manipulated via atomic primitives that generate events. For example, fetch-and-increment operation (FAI_t) to the i-th t done by CPU *c* generates an event c.FAI_t i. Note that FAI_t is implemented using instruction xaddl with the lock prefix in x86.

The lock implementation generates a list of events; for example, when CPU *c* acquires the lock i, it continuously generates the event c.get_n i (line 15) until the latest n is increased to the ticket value returned by the event c.FAI_t i (line 14), and then followed by the event c.pull i (line 16):



Verifying the linearizability and starvation-freedom of the ticket lock is equivalent to proving that under a *fair* hardware scheduler ε_{hs}, the ticket lock implementation is a *termination-sensitive* contextual refinement of its atomic specification.[20] There are two main proof obligations: (1) the lock guarantees *mutual exclusion*, and (2) the acq operation eventually succeeds.

The *mutual exclusion* property relies on the fact that, at any time, only the thread whose ticket t is equal to the current serving ticket (i.e., n) can hold the lock, and each thread's ticket t is unique. Here, we must also handle potential integer overflows for t and n. As long as the total number of CPUs (i.e., #CPU) in the machine is less than 2^{32} (determined by the uint type), this uniqueness property can be ensured. Then, it is safe to pull the shared memory associated with the lock i to the local copy at line 16. Before releasing the lock, the local copy is pushed back to the shared memory at line 19.

The *starvation-freedom* property relies on the fairness of the scheduler, that is, any CPU can be scheduled within *n* steps for some *n*. We define invariant INV_{lock} over the environment context to say that environmental lock-holders will release the lock within *m* steps. By enforcing INV_{lock}, we can prove that the while-loop in acq (line 15) terminates in $n \times m \times$ #CPU iterations on a CPU-local machine.

After showing the above two properties, we can build a *certified CPU-local layer*, whose overlay interface contains an atomic specification (L_{acq}) that simply generates an event t.acq i. These per-CPU certified layers can be composed in parallel as long as INV_{lock} holds on each CPU's local execution.

This event-based specification for the spinlock is also general enough to capture other implementations such as the *MCS Lock*. In mC2, we have also implemented a version of MCS locks.[16] The starvation-freedom proof is similar to that of the ticket lock. The difference is that the MCS lock-release operation waits in a loop until the next waiting thread (if it exists) has added itself to a linked list, so we need similar proofs for both acquisition and release.

Shared memory management provides a protocol to share physical pages among different user processes. A physical page can be mapped into multiple processes' page maps. For each page, we maintain a *logical owner set*. For example, a user process k_1 can share its private physical page *i* to another process k_2 and the logical owner set of page *i* is changed from $\{k_1\}$ to $\{k_1, k_2\}$. A shared page can only be freed when its owner set is a *singleton*.

Shared queue library abstracts the queues implemented as doubly linked lists into abstract queue states (i.e., Coq lists). Local enqueue and dequeue operations are specified over the abstract lists. Shared queue operations are protected by spinlocks and are specified by queue events t.enQ i e and t.deQ i. These events can be replayed (with the function $\mathbb{R}_{queue}$) to construct the queue state. For example, if the current log of the *i*-th shared queue is [t0.enQ i 2], and the event list returned by ε is [t1.enQ i 3, t2.enQ i 5], then the resulting log of calling deQ is:

$$\texttt{t0.enQ i 2} \bullet \texttt{t1.enQ i 3} \bullet \texttt{t2.enQ i 5} \bullet \texttt{t0.deQ i}$$

By replaying the log, the queue state is [3;5] and deQ returns 2.

Thread management introduces the thread control block and manages the resources of dynamically spawned threads (e.g., via quotas) and their metadata (e.g., children, thread state). For each thread, one page (4KB) is allocated for its *kernel stack*. We use an external tool[4] to show that the stack usage of our compiled kernel is less than 4KB, so stack overflows cannot occur inside the kernel.

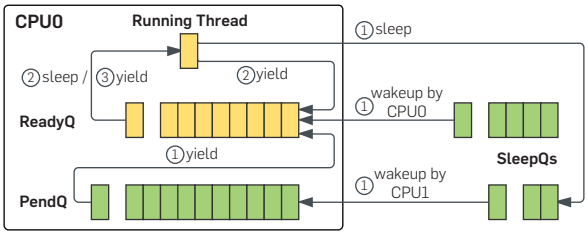
Thread control switches are implemented by the context switch function. This assembly function saves the register set of the current thread and restores the register set of another thread on the same CPU. As the instruction pointer

register (EIP) and stack pointer register (ESP) are saved and restored in this procedure, this kernel context switch function does not satisfy the C calling convention and has to be verified at the assembly level. Based on this context switch function and the shared queue library, we can verify three scheduling primitives: yield, sleep, and wakeup (see Figure 7).

Thread-local machine models can be built based on the thread management layers. The first step is to extend the environment context with a *software scheduler* (i.e., abstracting the concrete scheduling procedure), resulting in a new environment context ε_{ss}. The scheduling primitives generate the `t.yield`, `t.sleep i` and `t.wakeup i` events. ε_{ss} responds with the next thread ID to execute. The second step is to introduce the *active thread set* to represent the active threads on the active CPU, and extend ε_{ss} with the *context threads*, that is, the rest of the threads running on the active CPU. The composition structure is similar to the one of Lemma 3. In this way, higher layers can be built upon a thread-local machine with a single active thread on the active CPU (see Figure 5).

Condition variable (CV) is a synchronization object that enables a thread to wait for a change to be made to a shared state. Standard Mesa-style CVs[18] do not guarantee starvation-freedom: a thread waiting on a CV may not be signaled within a bounded number of execution steps. We have implemented a starvation-free CV using condition queues as shown by the popular, most up-to-date OS textbook.[2] However, we have found a bug in the FIFOBBQ implementation as shown in that textbook. Their system can get stuck in two cases: (1) when the destroyed CV is kept inside the remove queue (rmvQ), which will block the insert call to wake up the proper waiter; (2) when multiple CVs are woken up within a short period and the lock-holding CV thread is not the head of rmvQ, that thread will be removed from rmvQ and return to sleep, but will never be woken up again. We fixed this issue by postponing the removal of the CV thread from rmvQ, until woken thread that is allowed to proceed finishes its work; this thread is now responsible for removing itself from rmvQ, as well as waking up the next thread in rmvQ.

Figure 7. Each CPU has a private ready queue ReadyQ and a shared pending queue PendQ. The environmental CPUs can insert threads to the current CPU's PendQ. The mC2 kernel also provides a set of shared sleeping queues SleepQs. The yield primitive moves a thread from PendQ to ReadyQ and then switches to the next ready thread. The sleep primitive simply adds the running thread to a SleepQ and runs the next ready thread. The wakeup primitive contains two cases. If the thread to be woken up belongs to the current CPU, it will be added to the corresponding ReadyQ. Otherwise, the thread is added to PendQ of the CPU it belongs to.



5. EVALUATION

5.1. Proof effort and cost of change

Overall, our certified mC2 kernel consists of 6500 lines of C and x86 assembly. The concurrency extensions were completed in about two person-years. The new concurrency framework (to specify, build, and link certified concurrent abstraction layers) took about one person-year to develop. We extended the certified sequential mCertiKOS kernel[5, 8, 10] (which took another two person-years to develop in total) with various features, such as dynamic memory management, container support for controlling resource consumption, Intel hardware virtualization support, shared memory IPC, two-copy synchronous IPC, ticket and MCS locks, new schedulers, condition variables, etc. Some of these features were initially added in the sequential setting but later ported to the concurrent setting. The verification of these features was completed around one person-year. During this development process, many of our certified layers underwent many modifications and extensions. The CertiKOS framework allows such incremental development to take place much more smoothly. For example, certified layers in the sequential kernel can be directly ported to the concurrent setting if they only access private state. We have also adapted the work by Chen et al.[5] on interruptible kernels with device drivers to our multicore model.

Regarding the proof effort, there are 5249 lines of additional specifications for the various kernel functions, and about 40K LOC used to define auxiliary definitions, lemmas, theorems, and invariants. Additionally, there are 50K lines of proof scripts for proving the newly added concurrency features.

5.2. Bugs found

Other than the FIFOBBQ bug, we have also found a few other bugs during verification. Our initial ticket-lock implementation contains a particularly subtle bug: the spinning loop body (line 15 in Figure 1) was implemented as while(get_n() < my_t). This passed all our tests, but during the verification, we found that it did not satisfy the atomic specification as the ticket field might overflow. For example, if the next ticket number t is $(2^{32}-1)$, an overflow will occur in acq (line 14 in Figure 1) and the returned ticket my_t will equal to 0. In this case, current-serving number n is not less than my_t and acq gets the lock immediately, violating the mutual exclusion property.

5.3. Performance evaluation

Although performance is not the main emphasis of this work, we have run a number of micro and macro benchmarks to measure the speedup and overhead of mC2, and to compare mC2 with existing systems such as KVM and seL4. All experiments have been performed on a machine with one Intel Xeon CPU with four cores running at 2.8 GHz. As the power control code has not been verified, we disabled the turbo boost and power management features of the hardware during experiments.

5.4. Concurrency overhead

The runtime overhead introduced by concurrency in mC2 mainly comes from *the latency of spinlocks*.

The mC2 kernel provides two kinds of spinlocks: ticket lock and MCS lock. They have the same interface and thus are interchangeable. In order to measure their performance, we put an empty critical section (payload) under the protection of a single lock. The latency is measured by taking a sample of 10000 consecutive lock acquires and releases (transactions) on each round.

Figure 8a shows the results of our latency measurement. In the single-core case, ticket locks impose 34 cycles of overhead, whereas MCS locks impose 74 cycles as shown in the line chart. As the number of cores grows, the latency increases rapidly. As the slowdown should be proportional to the number of cores, to show the actual efficiency of the lock implementations, we normalize the latency against the baseline (single core) multiplied by the number of cores ($n*t_1/t_n$). As can be seen from the bar chart, efficiency remains about the same for MCS locks, but decreases for ticket locks.

Now that we have compared MCS locks with ticket locks, we present the remaining evaluations in this section using only the ticket lock implementation of mC2.

5.5. Hypervisor performance

To evaluate mC2 as a hypervisor, we measured the performance of some macro benchmarks on Ubuntu 12.04.2 LTS running as a guest. We ran the benchmarks on Linux as guest in both KVM and mC2, as well as on the bare metal. The guest Ubuntu is installed on an internal SSD drive. KVM and mC2 are installed on a USB stick. We use the standard 4KB pages in every setting—huge pages are not used.

Figure 8b contains a compilation of standard macro benchmarks: unpacking a Linux 4.0-rc4 kernel archive, compiling the Linux 4.0-rc4 kernel source, running Apache HTTPerf on loopback, and the DaCaPo Benchmark 9.12. We normalize the running times of the benchmarks using the bare metal performance as a baseline (100%). The overhead of mC2 is moderate and comparable to KVM. In some cases, mC2 performs better than KVM; we suspect this is because KVM has a Linux host and thus has a larger cache footprint. For benchmarks with a large number of file operations, such as Uncompress Linux source and Tomcat, mC2 performs worse. This is because mC2 exposes the raw disk interface to the guest via VirtIO (instead of passing it through), and its disk driver does not provide good buffering support.

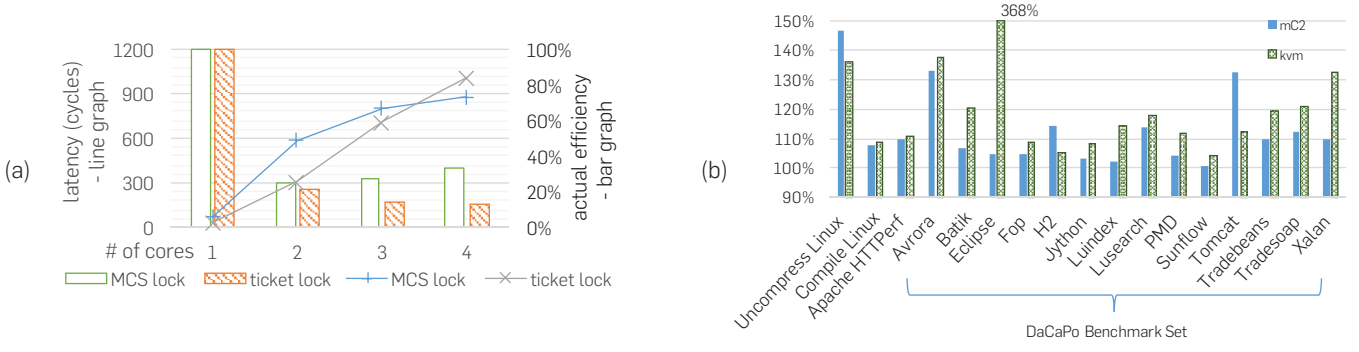
6. RELATED WORK

Dijkstra[9] proposed to "realize" a complex program by decomposing it into a hierarchy of linearly ordered abstract machines. Based on this idea, Gu et al.[10] developed new languages and tools for building certified abstraction layers with *deep* specifications, and showed how to apply the layered methodology to construct certified (sequential) OS kernels in Coq. Costanzo et al.[8] showed how to prove security properties over a deep specification of a certified OS kernel, and then propagate these properties from the specification level to its correct assembly-level implementation. Chen et al.[5] extended the layer methodology to build certified kernels and device drivers running on multiple *logical* CPUs. They treated the driver stack for each device as if it were running on a logical CPU dedicated to that device. Logical CPUs do not share any memory, and are all eventually mapped onto a single physical CPU. None of these systems, however, can support shared-memory concurrency with fine-grained locking.

The seL4 team[17] was the first to verify the functional correctness and security properties of a high-performance L4-family microkernel. The seL4 microkernel, however, does not support multicore concurrency with fine-grained locking. Peters et al.[22] and von Tessin[23] argued that for an seL4-like microkernel, concurrent data accesses across multiple CPUs can be reduced to a minimum, so a single *big kernel lock (BKL)* might be good enough for achieving good performance on multicore machines. von Tessin[23] further showed how to convert the single-core seL4 proofs into proofs for a BKL-based clustered multikernel.

The Verisoft team[1] applied the VCC framework[7] to formally verify Hyper-V, which is a widely deployed multiprocessor hypervisor consisting of 100 kLOC of C code and 5 kLOC of assembly. However, only 20% of the code is verified[7]; it is also only verified for function contracts and type invariants, rather than the full functional correctness property. CIVL[14] uses the state-machine approach with support for atomic actions and movers to reduce the proof burden for concurrent programs. It is implemented as an extension to Boogie and has been used to verify a concurrent garbage collector. However, CIVL can only be used to reason about safety rather than liveness. There is a large body of other work[6, 13, 24] showing how to build verified OS kernels,

Figure 8. (a) The comparison between actual efficiency of ticket lock and MCS lock implementations in mC2; (b) normalized performance for macro benchmarks running over Linux on KVM versus Linux on mC2; the baseline is Linux on bare metal; a smaller ratio is better.



hypervisors, file systems, device drivers, and distributed systems, but they do not address the issues of shared memory concurrency.

7. CONCLUSION

We have presented a novel extensible architecture for building certified concurrent OS kernels that not only have an efficient assembly implementation, but also machine-checkable contextual correctness proofs. OS kernels developed using our layered methodology also come with a clean, rigorous, and layered specification of all kernel components. We show that building certified concurrent kernels is not only feasible but also quite practical. Our layered approach to certified concurrent kernels replaces the hardware-enforced "red line" with a large number of abstraction layers enforced via formal specification and proofs. We believe this will open up a whole new dimension of research efforts toward building truly reliable, secure, and extensible system software.

Acknowledgments

We would like to acknowledge the contribution of many former and current team members on various CertiKOS-related projects at Yale, especially Tahina Ramananandro, Shu-Chun Weng, Liang Gu, Mengqi Liu, Quentin Carbonneaux, Jan Hoffmann, Hernán Vanzetto, Bryan Ford, Haozhong Zhang, and Yu Guo. We thank Xupeng Li, John Zhuang Hui, Xuguang Patrick Dai, members of the VeriGu lab at Columbia, and anonymous referees for helpful comments and suggestions that improved this research and the implemented tools. This research is based on the work supported in part by NSF grants 1065451, 1521523, and 1715154 and DARPA grants FA8750–12–2–0293, FA8750–16–2–0274, and FA8750–15-C-0082. It is also supported in part by Qtum Foundation and Baidu USA. 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. C

References

1. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E. Automated verification of a small hypervisor. In *Proceedings of 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)* (2010), 40–54.
2. Anderson, T., Dahlin, M. *Operating Systems Principles and Practice*. Recursive Books, 2011 (Figure 5.14).
3. Belay, A., Bittau, A., Mashtizadeh, A., Mazières, D., Kozyrakis, C. Dune: Safe user-level access to privileged CPU features. In *Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)* (2012), 335–348.
4. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z. End-to-end verification of stack-space bounds for C programs. In *Proceedings of 2014 ACM Conference on Programming Language Design and Implementation (PLDI'14)* (2014), 270–281.
5. Chen, H., Wu, X., Shao, Z., Lockerman, J., Gu, R. Toward compositional verification of interruptible OS kernels and device drivers. In *Proceedings of 2016 ACM Conference on Programming Language Design and Implementation (PLDI'16)* (2016), 431–447.
6. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of 25th ACM Symposium on Operating System Principles (SOSP)* (2015), 18–37.
7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S. VCC: A practical system for verifying concurrent C. In *Proceedings of 22nd International Conference on Theorem Proving in Higher Order Logics* (2009), 23–42.
8. Costanzo, D., Shao, Z., Gu, R. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of 2016 ACM Conference on Programming Language Design and Implementation (PLDI'16)* (2016), 648–664.
9. Dijkstra, E.W. The structure of the "THE"-multiprogramming system. *Commun. ACM*, (1968), 341–346.
10. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X., Weng, S.-C., Zhang, H., Guo, Y. Deep specifications and certified abstraction layers. In *Proceedings of 42nd ACM Symposium on Principles of Programming Languages (POPL'15)* (2015), 595–608.
11. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D. Certikos: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (2016), 653–669.
12. Gu, R., Shao, Z., Kim, J., Wu, X.N., Koenig, J., Sjöberg, V., Chen, H., Costanzo, D., Ramananandro, T. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018), ACM, 646–661.
13. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (2014), 165–181
14. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification* (2015), Springer, 449–465.
15. Herlihy, M., Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
16. Kim, J., Sjöberg, V., Gu, R., Shao, Z. Safety and liveness of MCS lock—Layer by layer. In *Asian Symposium on Programming Languages and Systems* (2017), Springer, 273–297.
17. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009), ACM, 207–220.
18. Lampson, B.W. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (1980).
19. Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (2009), 107–115.
20. Liang, H., Feng, X. A program logic for concurrent objects under fair scheduling. In *Proceedings of 43rd ACM Symposium on Principles of Programming Languages (POPL'16)* (2016), 385–399.
21. Mellor-Crummey, J.M., Scott, M.L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM T. Comput. Syst. 9*, 1 (1991), 21–65.
22. Peters, S., Danis, A., Elphinstone, K., Heiser, G. For a microkernel, a big lock is fine. In *APSys '15 Asia Pacific Workshop on Systems, Tokyo, Japan* (2015).
23. von Tessin, M. *The clustered multikernel: An approach to formal verification of multiprocessor operating-system kernels*. PhD thesis, School of Computer Science and Engineering, The University of New South Wales (2013).
24. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z. A practical verification framework for preemptive OS kernels. In *Proceedings of 28th International Conference on Computer-Aided Verification (CAV), Part II* (2016), 59–79.

Ronghui Gu (ronghui.gu@columbia.edu), Columbia University, New York, NY, USA..

Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, and David Costanzo ([zhong.shao,hao.chen,jieung.kim,jeremie.koenig,xiongnan.wu,david.costanzo]@yale.edu), Yale University, New Haven, CT, USA.

Vilhelm Sjöberg (vilhelm.sjoberg@certik.org), CertiK, Cambridge, MA, USA.