# CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels

Ronghui Gu Zhong Shao Hao Chen Xiongnan (Newman) Wu
Jieung Kim Vilhelm Sjöberg David Costanzo

*Yale University*

## Abstract

Complete formal verification of a non-trivial concurrent OS kernel is widely considered a grand challenge. We present a novel compositional approach for building certified concurrent OS kernels. Concurrency allows interleaved execution of kernel/user modules across different layers of abstraction. Each such layer can have a different set of observable events. We insist on formally specifying these layers and their observable events, and then verifying each kernel module at its proper abstraction level. To support certified linking with other CPUs or threads, we prove a strong contextual refinement property for every kernel function, which states that the implementation of each such function will behave like its specification under any kernel/user context with any valid interleaving. We have successfully developed a practical concurrent OS kernel and verified its (contextual) functional correctness in Coq. Our certified 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 proof of functional correctness of a complete, 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 in the world. Hence it is highly desirable to formally verify the correctness of these programs [53]. Recent efforts [33, 58, 34, 25, 23, 13, 5, 14] 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 [31, 7], including not only user and I/O concurrency on a single CPU, but also multicore parallelism with fine-grained locking. This severely limits the applicability and power 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 across different abstraction layers; they contain many interdependent components that are difficult to untangle. Several researchers [55, 51] believe that the combination of concurrency and the kernels' functional complexity makes formal verification of functional correctness intractable, and even if it is possible, its cost would far exceed that of verifying a single-core sequential kernel.

Second, concurrent kernels need to support all three types of concurrency (user, I/O, or multicore) and make them work coherently with each other. User and I/O concurrency rely on thread yield/sleep/wakeup primitives or interrupts to switch control and support synchronization; these constructs are difficult to reason about since they transfer control from one thread to another. Multicore concurrency with fine-grained locking requires sophisticated spinlock implementations such as MCS locks [46], which are also hard to verify.

Third, concurrent kernels should also guarantee that each of their system calls eventually returns, but this depends on the progress of the concurrent primitives used in the kernels. Proving starvation-freedom [28] for concurrent objects only became possible recently [40]. Standard Mesa-style condition variables [35] do not guarantee starvation-freedom; this can be fixed by using a FIFO queue of condition variables, but the solution is not trivial and even the popular, most up-to-date OS textbook [7, Fig. 5.14] has gotten it wrong [6].

Fourth, given the high cost of building concurrent kernels, it is important that they can be quickly adapted to support new hardware platforms and applications [8, 45, 20]. One advantage of a certified kernel is the formal specification for all of its components. In theory, this allows us to add certified kernel plug-ins as long as they do not violate any existing invariants. In practice, however, if we are unable to encapsulate interference, even a small edit could incur huge verification overhead.

In this paper, we present a novel compositional approach that tackles all these challenges. We believe that, to control the complexity of concurrent kernels and to provide strong support for 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 meticulously uncovering and specifying these layers, and then verifying each kernel module at its *proper* abstraction level.

The functional correctness of an OS kernel is often stated as a *refinement*. This is shown by building *forward simulation* [44] from the C/assembly implementation of a kernel ($K$) to its abstract functional specification ($S$). Of course, the ultimate goal of having a certified kernel is to reason about programs 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 $K \bowtie P$ also refines $S \bowtie P$. If this fails to hold, the kernel is simply still incorrect since $P$ can observe some difference between $K$ and $S$. Gu et al. [23] advocated proving such a *contextual refinement* property, but they only considered the *sequential* contexts (i.e., $P$ is sequential).

For concurrent kernels, proving the *contextual refinement* property becomes essential. In the sequential setting, the only way that the context code $P$ can interfere with the kernel $K$ is when $K$ fails to encapsulate its private state; that is, $P$ can modify some internal state of $K$ without $K$'s permission. In the concurrent setting, the *environment* context ($\varepsilon$) of a running kernel $K$ could be other kernel threads or a copy of $K$ running on another CPU. With shared-memory concurrency, interference between $\varepsilon$ and $K$ is both necessary and common; the sequential *atomic* specification $S$ is now replaced by the notion of linearizability [29] plus a progress property such as starvation-freedom [28].

In fact, linearizability proofs often require event reordering that preserves the happens-before relation, so $K \bowtie \varepsilon$ may not even refine $S \bowtie \varepsilon$. Contextual refinement in the concurrent setting requires that for any $\varepsilon$, we can find a *semantically related* $\varepsilon'$ such that $K \bowtie \varepsilon$ refines $S \bowtie \varepsilon'$. Several researchers [22, 42, 40] have shown that contextual refinement is precisely equivalent to the linearizability and progress requirements for implementing compositional concurrent objects [28, 29].

Our paper makes the following contributions:

- We present **CertiKOS**—a new extensible architecture for building certified concurrent OS kernels. CertiKOS uses contextual refinement over the "concurrent" *environment contexts* ($\varepsilon$) as the *unifying* formalism for composing different concurrent kernel/user objects at different abstraction levels. Each $\varepsilon$ defines a specific instance on how other threads/CPUs/devices respond toward the events generated by the current running threads. Each abstraction layer, parameterized over a specific $\varepsilon$, is an assembly-level machine extended with a particular set of abstract objects (i.e., abstract states plus atomic primitives). CertiKOS successfully decomposes an otherwise prohibitive verification task into many simple and easily automatable ones.
- We show how the use of an environment context at each layer allows us to apply standard techniques for verifying sequential programs to verify concurrent programs. Indeed, most of our kernel programs are written in a variant of C (called ClightX) [23], verified at the source level, and compiled and linked together using CompCertX [23, 24] — a *thread-safe* version of the CompCert compiler [37, 38]. As far as we know, CertiKOS is the first architecture that can truly build certified concurrent kernels and transfer global properties proved for programs (at the kernel specification level) down to the concrete assembly machine level.
- We show how to impose temporal invariants over these environment contexts so we can verify the progress of various concurrent primitives. For example, to verify the starvation-freedom of ticket locks or MCS locks, we must assume that the multicore hardware (or the OS scheduler) always generates a *fair* interleaving, and those threads/CPUs which requested locks before the current running thread will eventually acquire and then release the lock. In a separate paper [24], we present the formal theory of environment contexts and show how these assumptions can be discharged when we compose different threads/CPUs to form a complete system.
- Using CertiKOS, we have successfully developed a fully certified concurrent OS kernel (called mC2) in the Coq proof assistant [2]. Our kernel supports both fine-grained locking and thread yield/sleep/wakeup primitives, and can run on stock x86 multicore machines. It can also double as a hypervisor and boot multiple instances of Linux in guest VMs running on different CPUs. Our certified hypervisor kernel consists of 6500 lines of C and x86 assembly. The entire proof effort for supporting concurrency took less than 2 person years. To our knowledge, this is the first proof of functional correctness of a complete, general-purpose concurrent OS kernel with fine-grained locking.

The rest of this paper is organized as follows. Section 2 gives an overview of our new CertiKOS architecture. Section 3 shows how we use environment contexts to turn concurrent layers into sequential ones. Section 4 presents the design and development of the mC2 kernel and how we verify various concurrent kernel objects. Section 5 presents an evaluation of CertiKOS. Sections 6-7 discuss related work and then conclude.

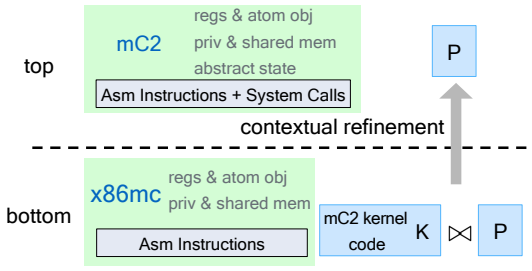



**Figure 1:** Certified OS kernels: what to prove?

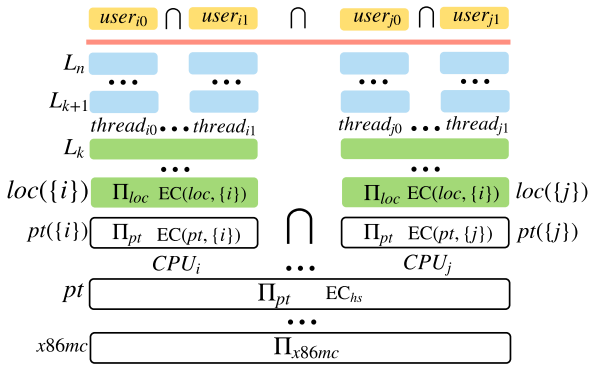



**Figure 2:** Contextual refinement between concurrent layers

## 2 Overview of Our Approach

The ultimate goal of research on building certified OS kernels is not just to verify the functional correctness of a particular kernel, but rather to find the best OS design and development methodologies that can be used to build provably reliable, secure, and efficient computer systems in a cost-effective way. We enumerate a few important dimensions of concerns and evaluation metrics which we have used so far to guide our work toward this goal:

- **Support for new kernel design.** Traditional OS kernels use the hardware-enforced "red line" to define a single system call API. A certified OS kernel opens up the design space significantly as it can support multiple certified kernel APIs at different abstraction levels. It is important to support kernel extensions [9, 20, 45] and novel ring-0 or guest-domain processes [30, 8] so we can experiment and find the best trade-offs.
- **Kernel performance.** Verification should not impose significant overhead on kernel performance. Of course, different kernel designs may imply different performance priorities. An L4-like microkernel [43] focuses on fast inter-process communication (IPC), while a Singularity-like kernel [30] emphasizes efficient support for type-safe ring-0 processes.
- **Verification of global properties.** A certified kernel is much less interesting if it cannot be used to prove global properties of the complete system built on top of the kernel. Such global properties include not only safety, liveness, and security properties of user-level processes and virtual machines, but also resource usage and availability properties (e.g., to counter denial-of-service attacks).
- **Quality of kernel specification.** A good kernel specification should capture precisely the *contextually observable* behaviors of the implementation [23]. It must support transferring global properties proved at a high abstraction level down to any lower abstraction level [16].
- **Cost of development and maintenance.** Compositionality is the key to minimize such cost. If the machine model is stable, verification of each kernel module should only need to be done once (to show that it *implements* its deep functional specification [23]). Global properties (e.g., information flow security) should be derived from the kernel deep specification alone [16].
- **Quality of formal proofs.** We use the term *certified kernels* rather than *verified kernels* to emphasize the importance of third-party machine-checkable proof certificates [53]. Hand-written paper proofs are error-prone [32]. Program verification without explicit machine-checkable proof objects has been subject to significant controversy [17].

**Overview of CertiKOS** Our new CertiKOS architecture aims to address all these concerns and also tackle the challenges described in Section 1. The CertiKOS architecture leverages the new certified programming methodologies developed by Gu et al. [23, 24] and applies them to support building certified concurrent OS kernels.

A *certified abstraction layer* consists of a language construct $(L_1, M, L_2)$ and a mechanized proof object showing that the layer implementation $M$, built on top of the interface $L_1$ (the *underlay*), is a *contextual refinement* of the desirable interface $L_2$ above (the *overlay*). A *deep* specification ($L_2$) of a module ($M$) captures everything *contextually observable* about running the module over its underlay ($L_1$). Once we have certified $M$ with a deep specification $L_2$, there is no need to ever look at $M$ again, and any property about $M$ can be proved using $L_2$ alone.

In Figure 1, we use x86mc to denote an assembly-level multicore machine. Suppose we load such a machine with the mC2 kernel $K$ (in assembly) and user-level assembly code $P$, and we use $[[\cdot]]_{\text{x86mc}}$ to denote the whole-machine semantics for x86mc, then proving any global property of such a complete system amounts to reasoning about the semantic object $[[K \bowtie P]]_{\text{x86mc}}$, i.e., the set of observable behaviors from running $K \bowtie P$ on x86mc.

Reasoning at such a low level is difficult, so we formalize a new mC2 machine that extends the x86mc machine with the (deep) high-level specification of all system calls implemented by $K$. We use $[[\cdot]]_{\text{mC2}}$ to denote its whole-

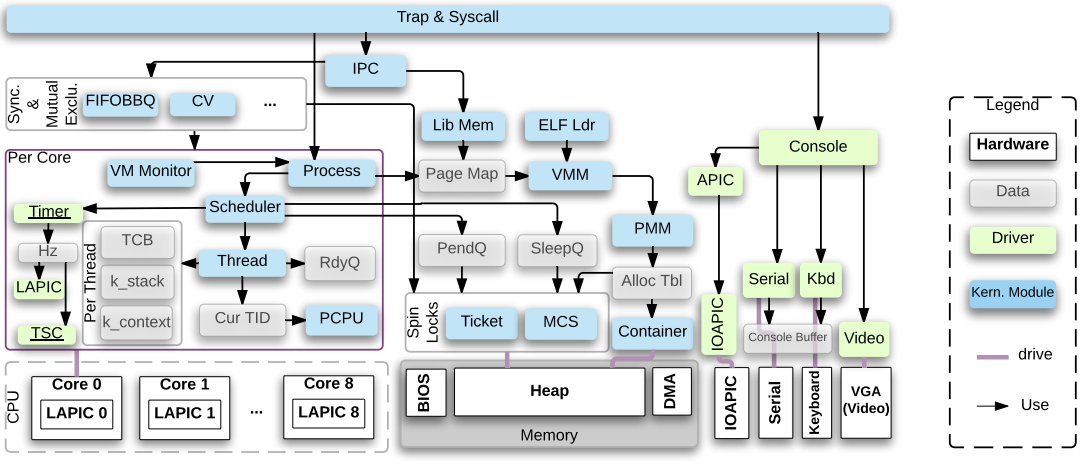



**Figure 3:** System architecture for the mC2 kernel

machine semantics. The contextual refinement property about the mC2 kernel can be stated as:

$$\forall P,\ [[K \bowtie P]]_{\text{x86mc}} \sqsubseteq [[P]]_{\text{mC2}}$$

Hence any global property proved about $[[P]]_{\text{mC2}}$ can be transferred to $[[K \bowtie P]]_{\text{x86mc}}$.

To support concurrency, for each layer interface $L$, we parameterize it with an *active* thread set $A$ and then carefully define its set of valid *environment contexts*, denoted as $\text{EC}(L,A)$. Each environment context $\varepsilon$ captures a specific instance—from a particular run—of the list of events that other threads or CPUs (i.e., those not in $A$) return when responding to the events generated by those in $A$. We can then define a new *thread-modular* machine $\Pi_{\text{L(A)}}(P,\varepsilon)$ that will operate like the usual assembly machine when $P$ switches control to those threads in $A$, but will only obtain the list of events from the environment context $\varepsilon$ when $P$ switches control to those outside $A$. The semantics for a concurrent layer machine $L$ is then:

$$[[P]]_{L(A)} = \{\ \Pi_{\text{L(A)}}(P,\varepsilon)\ \mid\ \varepsilon \in \text{EC}(L,A)\ \}$$

To support parallel layer composition, we carefully design $\text{EC}(L,A)$ so that the following property holds:

$$[[P]]_{L(A\cup B)}\ =\ [[P]]_{L(A)}\ \cap\ [[P]]_{L(B)}\ \ \textit{if}\ A \cap B = \varnothing$$

The formal details for $\text{EC}(L,A)$ and $[[\cdot]]_{L(A)}$ are presented in a separate paper [24]. Note that if $A$ is a singleton, for each $\varepsilon$, $\Pi_{\text{L(A)}}$ behaves like a sequential machine.

With our new compositional layer semantics, we can take a multicore machine like x86mc and zoom into a specific active CPU $i$ by creating a *logical* "single-core" machine layer for CPU $i$, and then apply techniques from Gu et al. [23] to build a collection of certified "sequential" (per-CPU) layers (see Figure 2). When we want to introduce kernel- or user-level threads, we can further zoom into a particular thread (e.g., $i0$) and create a corresponding logical machine layer. We can impose specific invariants over the environment contexts (i.e., the "rely" conditions) and use them to ensure that per-CPU or per-thread reasoning can be soundly composed (when their "rely" conditions are compatible with each other). After we have added all the kernel components and implemented all the system calls, we can combine these per-thread machines into a single concurrent machine.

Under CertiKOS, building a new certified concurrent kernel (or experimenting with a new design) is just a matter of composing a collection of certified concurrent layers, developed in a variant of C (called ClightX) or assembly. Gu et al. [23] have developed a certified compiler (CompCertX) that can compile certified ClightX layers into certified assembly layers. Since all concurrent primitives in CertiKOS are treated as CompCert-style external calls or built-ins, they cannot be reordered or optimized away by the compiler. Memory accesses over these external calls cannot be reordered either. Therefore, each concurrent ClightX module (running over a particular per-thread or per-CPU layer) is compiled by CompCertX as if it were a sequential program performing many external-call events. The correctness of CompCertX guarantees that the generated x86 assembly behaves the same as the source ClightX module. CompCertX can therefore serve as a *thread-safe* version of CompCert.

CertiKOS can thus enjoy the full programming power of both an ANSI C variant and an assembly language to certify any efficient routines required by low-level kernel programming. The layer mechanism allows us to certify most kernel components at higher abstraction levels, even though they all eventually get mapped (or compiled) down to an assembly machine.

**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 re-

search 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 RADL (The Robot Architecture Definition Language [39]) 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 support so that even if attackers take control of one VM, they still cannot break into other VMs to compromise the overall mission of the UGV.

Within mC2, we have various shared objects such as spinlock modules (Ticket, MCS), sleep queues (SleepQ) for implementing queueing 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.

**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 important functional correctness property implies that all system calls and traps will strictly follow the high-level specification and always run *safely* and *terminate* eventually; and 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 $[[K \bowtie P]]_{\text{x86mc}}$ refines $[[P]]_{\text{mC2}}$, we can also derive the important *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 $K \bowtie P$ running on the concrete x86mc machine would indeed behave exactly the same. All global properties proved at the system-call specification level can be transferred down to the lowest assembly machine.

**Assumptions and limitations** The mC2 kernel is obviously not as comprehensive as real-world kernels such as Linux. The main goal of this paper is to show that it is feasible to build certified concurrent kernels with fine-grained locking. We did not try to incorporate all the latest advances for multicore kernels into mC2.

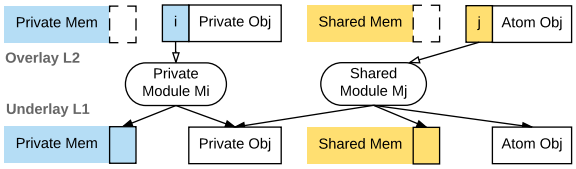



**Figure 4:** Defining concurrent abstraction layers

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 non-atomic 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 [55].

Since our machine does not model TLB, any code for addressing TLB shootdown cannot be verified.

The mC2 kernel currently lacks a certified storage system. We plan to incorporate recent advances in building certified file systems [13, 5] into mC2 in the near future.

Our assembly machine only covers a small part of the full x86 instruction set, so our contextual correctness results only apply to programs in this subset. Additional instructions can be easily added if they have simple or no interaction with our kernel. Costanzo et al. [16, Sec. 6] shows how the fidelity of the CompCert-style x86 machine model would impact the formal correctness or security claims, and how such gap can be closed.

The CompCertX assembler for converting assembly into machine code is unverified. We assume correctness of the Coq proof checker and its code extraction mechanism.

The mC2 kernel also relies on a bootloader, a *PreInit* module (which initializes the CPUs and the devices), and an ELF loader. Their verification is left for future work.

## 3 Layer Design with Environment Context

In this section, we explain the general layer design principles and show how we use environment context to convert a concurrent layer into CPU-local layers.

**Multicore hardware** allows all the CPUs to access the same piece of memory simultaneously. In CertiKOS, we *logically* distinguish the *private memory* (i.e., private to a CPU or a thread) from the *shared memory* (i.e., shared by multiple CPUs or threads). The private memory does not need to be synchronized, whereas non-atomic shared memory accesses need to be protected by some synchronization mechanisms (e.g., locks), which are normally implemented using atomic hardware instructions (e.g., fetch-and-add). With proper protection, each shared memory operation can be viewed as if it were atomic.

**Atomic object** is an abstraction of well-synchronized shared memory, combined with operations that can be performed over that shared memory. 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. Each primitive invocation records a *single* corresponding event in the log. 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.

**Concurrent layer interface** contains both *private objects* (e.g., $i$ in Fig. 4) and *atomic objects* (e.g., $j$ in Fig. 4), along with some invariants imposed on these objects. The verification of a concurrent kernel requires repeatedly building certified abstraction layers. The overlay interface $L_2$ is a new and more abstract interface, built on top of the underlay interface $L_1$, and implemented by module $M_i$ or $M_j$ (cf. Fig. 4). *Private objects* only access private memory and are built following techniques similar to those presented by Gu et al. [23]. *Atomic objects* are implemented by shared modules (e.g., $M_j$ in Fig. 4) that may access existing atomic objects, private objects, and non-atomic shared memory.

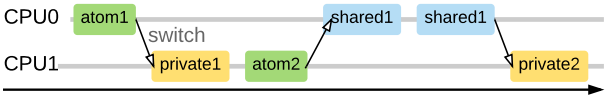
Every atomic primitive in the overlay generates exactly one event (this is why it is really atomic), while its implementation may trigger multiple events (by calling multiple atomic primitives in the underlay).

It is difficult to build certified abstraction layers directly on a multicore, nondeterministic hardware model. To construct an atomic object, we must reason about its implementation under all possible interleavings and prove that every access to shared memory is well synchronized.

In the rest of this section, we first present our x86 multicore machine model ($\Pi_{\text{x86mc}}$), and then show how we gradually refine this low-level model into a more abstract machine model ($\Pi_{\text{loc}}$) that is suitable for reasoning about concurrent code in a CPU-local fashion.

### 3.1 Multicore hardware model

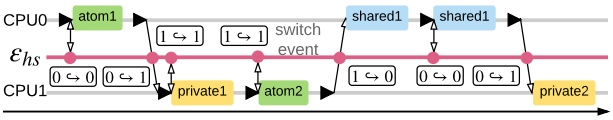
Our fine-grained multicore hardware model ($\Pi_{\text{x86mc}}$) allows arbitrary interleavings at the level of *assembly instructions*. At each step, the hardware *nondeterministically* chooses one CPU and executes the next assembly instruction on that CPU. Each assembly instruction is classified as *atomic*, *shared*, or *private*, depending on whether the instruction involves an atomic object call, a non-atomic shared memory access, or only a private object/memory access. One interleaving of an example program running on two CPUs is as follows:




Since only atomic operations generate events, this interleaving produces the logical log $[0.\texttt{atom}_1, 1.\texttt{atom}_2]$.

### 3.2 Machine model with hardware scheduler

As a first step toward abstracting away the low-level details of the concurrent CPUs, we introduce a new machine model ($\Pi_{\text{hs}}$) configured with a *hardware scheduler* ($\varepsilon_{hs}$) that specifies a particular interleaving for an execution. This results in a deterministic machine model. To take a program from $\Pi_{\text{x86mc}}$ and run it on top of $\Pi_{\text{hs}}$, we insert a *logical switch point* (denoted as "▶") before each assembly instruction. At each switch point, the machine first queries the hardware scheduler and gets the CPU ID that will execute next. All the *switch decisions* made by $\varepsilon_{hs}$ are stored in the log as switch events. The previous example on $\Pi_{\text{x86mc}}$ can be simulated by the following $\varepsilon_{hs}$:




The log recorded by this execution is as follows (a switch from CPU $i$ to $j$ is denoted as $i \hookrightarrow j$):

$$[0 \hookrightarrow 0, 0.\texttt{atom}_1, 0 \hookrightarrow 1, 1 \hookrightarrow 1, 1 \hookrightarrow 1, 1.\texttt{atom}_2, 1 \hookrightarrow 0, 0 \hookrightarrow 0, 0 \hookrightarrow 1]$$

The behavior of running a program $P$ over this model with a hardware scheduler $\varepsilon_{hs}$ is denoted as $\Pi_{\text{hs}}(P, \varepsilon_{hs})$, indicating that it is parametrized over all possible $\varepsilon_{hs}$. Let $\text{EC}_{hs}$ represent the set of all possible hardware schedulers. Then we define the whole-machine semantics:

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

Note this is a special case of the definition in Section 2 for the whole-machine semantics of a concurrent layer machine, where the active set is the set of all CPUs. To ensure correctness of this machine model with respect to the hardware machine model, we prove that $\Pi_{\text{x86mc}}$ *contextually refines* the new model. Before we state the property, we first define *contextual refinement* formally.

**Definition 1** (Contextual Refinement)**.** *We say that layer* $L_0$ contextually refines *layer* $L_1$ *(written as* $\forall P, [[P]]_{L_0} \sqsubseteq [[P]]_{L_1}$*), if and only if for any* $P$ *that does not go wrong on* $\Pi_{L_1}$ *under any configuration, we also have that (1)* $P$ *does not go wrong on* $\Pi_{L_0}$ *under any configuration; and (2) any observable behavior of* $P$ *on* $\Pi_{L_0}$ *under some configuration is also observed on* $\Pi_{L_1}$ *under some (possibly different) configuration.*

**Lemma 1** (Correctness of the hardware scheduler model)**.**

$$\forall P, [[P]]_{x86mc} \sqsubseteq [[P]]_{hs}$$

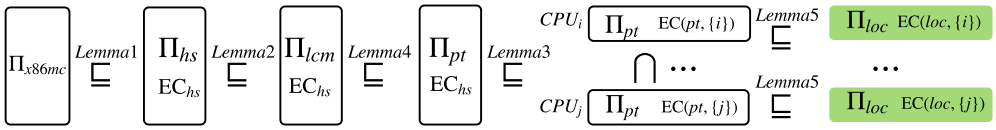



**Figure 5:** The contextual refinement chain from multicore hardware model $\Pi_{\text{x86mc}}$ to CPU-local model $\Pi_{\text{loc}}$

### 3.3 Machine with local copy of shared memory

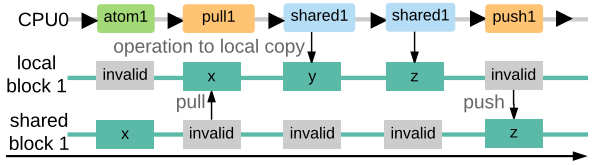
The above machine model does not restrict any access to the shared memory. We therefore abstract the machine model with hardware scheduler into a new model that enforces well-synchronized accesses to shared memory.

In addition to the global shared memory concurrently manipulated by all CPUs, each CPU on this new machine model ($\Pi_{\text{lcm}}$) also maintains a local copy of shared memory blocks along with a *valid bit*. The relation between a CPU's local copy and the global shared memory is maintained through two new *logical* primitives `pull` and `push`.

The `pull` operation over a particular CompCert-style memory block [37] 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, push an `invalid` local block, or access an `invalid` local block, the program goes wrong. 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 correspond to any physical protection mechanisms; thus they do not introduce any performance overhead.

The shared memory updates of the previous example can be simulated on $\Pi_{\text{lcm}}$ as follows:




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

We have shown (in Coq) that $\Pi_{\text{lcm}}$ is correct with respect to the previous machine model $\Pi_{\text{hs}}$ with the EC$_{hs}$.

**Lemma 2** (Correctness of the local copy model)**.**

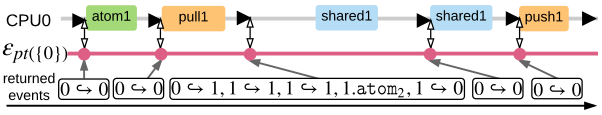
$$\forall P, [[P]]_{hs} \sqsubseteq [[P]]_{lcm}$$

### 3.4 Partial machine with environment context

Although $\Pi_{\text{lcm}}$ provides a way to reason about shared memory operations, it still does not have much support for CPU-local reasoning. To achieve modular verification, the machine model should provide a way to reason about programs on each CPU locally by specifying expected behaviors of the context programs on other CPUs. The model should then provide a systematic way to link the proofs of different local components together to form a global claim about the whole system. To this purpose, we introduce a partial machine model $\Pi_{\text{pt}}$ 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* (i.e., 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 EC$(pt, A)$. Each environment context $\varepsilon_{pt(A)} \in \text{EC}(pt, A)$ is a *response function*, which takes the current log and returns a list of events from the context programs (i.e., 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, i.e., the events returned by the response function include switch events. The execution of CPU 0 in the previous example can be simulated with a $\varepsilon_{pt(\{0\})}$ function:




For example, at the 3rd switch point, $\varepsilon_{pt(\{0\})}$ returns the event list $[0 \hookrightarrow 1, 1 \hookrightarrow 1, 1 \hookrightarrow 1, 1.\text{atom}_2, 1 \hookrightarrow 0]$.

**Composition of partial machine models** Suppose we have verified that two programs, separately running with two *disjoint* active CPU sets $A$ and $B$, produce event lists satisfying invariants $\text{INV}_A$ and $\text{INV}_B$, respectively. If $\text{INV}_A$ is consistent with the environment-context invariant of

$B$, and $\texttt{INV}_B$ 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 $\texttt{INV}_A \wedge \texttt{INV}_B$. Using the whole-machine semantics from Section 2, we express this composition as a contextual refinement.

**Lemma 3** (Composition of partial machine models)**.**

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

After composing the programs on all CPUs, the context CPU set becomes empty and the composed invariant holds on the whole machine. Since 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 $\text{EC}(pt, C) = \text{EC}_{hs}$. By showing that this *composed machine* with the entire CPU set $C$ is refined by $\Pi_{\text{lcm}}$, the proofs can be propagated down to the multicore hardware model.

**Lemma 4** (Correctness of the composed total machine)**.**

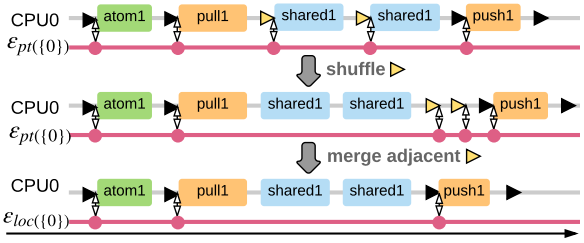
$$\forall P, [[P]]_{lcm} \sqsubseteq [[P]]_{pt(C)}$$

### 3.5 CPU-local machine model

If we focus on a single active CPU $i$, the partial machine model is like a *local* machine 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 (e.g., those between private operations). In this subsection, we introduce a CPU-local machine model (denoted as $\Pi_{\text{loc}}$) 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*.

**Shuffling switch points** In $\Pi_{\text{loc}}$, we introduce a *log cache* — for the switch points before shared and private operations, the query results from the environment context are stored in a temporary log cache. The cached events are applied to the logical log just before the next atomic or `push`/`pull` operation. Thus, when we perform shared or private operations, the observations of the environment context are delayed until the next atomic or `push`/`pull` operation. This is possible 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.

**Merging switch points** 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 CPU-local machine model)**.**

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

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 (cf. Fig. 5). We introduce and verify the mC2 kernel on top of the CPU-local machine model $\Pi_{\text{loc}}$. The refinement proof guarantees that the proved properties can be propagated down to the multicore hardware model $\Pi_{\text{x86mc}}$.

All our proofs (including every step in Fig. 5 and Fig. 2) 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 well the informal intuitions given in this and next sections.

## 4 Certifying the mC2 Kernel

Contextual refinement provides an elegant formalism for decomposing the verification of a complex kernel into a large number of small tractable tasks: we define a series of logical abstraction layers, which serve as increasingly higher-level specifications for an increasing portion of the kernel code. We design these abstraction layers in a way such that complex interdependent kernel components are untangled and converted into a well-organized kernel-object stack with clean specification (cf. Fig. 2).

In the mC2 kernel, the pre-initialization module is the bottom layer that connects to the *CPU-local machine model* $\Pi_{\text{loc}}$, instantiated with a particular *active CPU* (cf. Sec. 3.5). The trap handler contains the top layer that provides system call interfaces and serves as a specification of the whole kernel, instantiated with a particular active thread running on that active CPU. Our main theorem states that any global properties proved at the topmost abstraction layer can be transferred down to the lowest hardware machine. In this section, we explain selected components in more details.

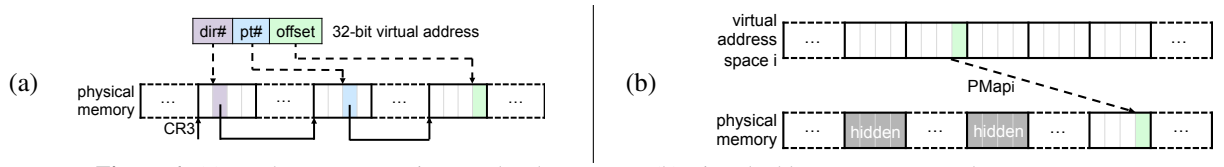
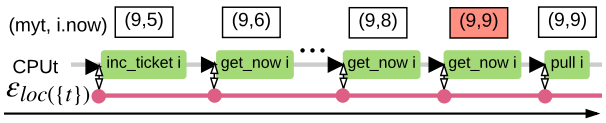



**Figure 6:** (a) Hardware MMU using two-level page map; (b) Virtual address space *i* set up by page map *i*

Each CPU-local pre-initialization machine defines the x86 hardware behaviors including page table walk 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 the data exchange between devices and memory. The hardware memory management unit (MMU) is modeled in a way that mirrors the paging hardware (cf. Fig. 6a). When paging is enabled, memory accesses made by both the kernel and the user programs are translated using the page map pointed to by `CR3`. When a page fault occurs, the fault information is stored in `CR2`, the CPU mode is switched from user mode to kernel mode, and the page fault handler is triggered.

**The spinlock module** provides fine-grained lock objects as the base of synchronization mechanisms.

**Ticket Lock** depends on an *atomic ticket object*, which consists of two fields: `ticket` and `now`. Figure 7 shows one implementation of a ticket lock. Here, `L` is declared as an array of ticket locks; each shared data object can be protected with one lock in the array, identified using a specific lock index (`i`). The atomic increment to the ticket is achieved through the atomic `fetch-and-increment` (FAI) operation (implemented using the `xaddl` instruction with the `lock` prefix in x86). As described in Section 3.5, the *switch points* at this abstraction level have been shuffled and merged so that there is exactly one switch point before each atomic operation. Thus, the lock implementations generate a list of events; for example, when CPU *t* acquires the lock *i* (stored in `L[i]`), it continuously generates the event "`t.get_now i`" (line 10) until the latest `now` is increased to the ticket value returned by the event "`t.inc_ticket i`" (line 9), and then followed by the event "`t.pull i`" (line 11):




The event list is as below:

$$[\blacktriangleright, \texttt{t.inc\_ticket i}, \blacktriangleright, \texttt{t.get\_now i}, \cdots, \blacktriangleright, \texttt{t.get\_now i}]$$

Verifying the linearizability and starvation-freedom of the ticket lock object is equivalent to proving that under a *fair* hardware scheduler $\mathcal{E}_{hs}$, the ticket lock implementation is a *termination-sensitive* contextual refinement of its atomic specification [42, 40]. There are two main proof

```
1  typedef struct {
2    volatile uint ticket;
3    volatile uint now;
4  } ticket_lock;
5  ticket_lock L[NUM_LOCK];
6
7  void acq_lock (uint i) {
8    uint t;
9    t=▶FAI(&L[i].ticket);
10   while( ▶L[i].now!=t){}
11   ▶pull (i);
12 }
13 void rel_lock (uint i) {
14   ▶push (i);
15   ▶L[i].now ++;
16 }
```

**Figure 7:** Pseudocode of the ticket lock implementation

obligations: (1) the lock guarantees *mutual exclusion*, and (2) the `acq_lock` operation eventually succeeds.

*Mutual exclusion* is straightforward for a ticket lock. At any time, only the thread whose ticket is equal to the current serving ticket (i.e., `now`) can hold the lock. Furthermore, each thread's ticket is unique as the `fetch-and-increment` operation is atomic (line 9). Thanks to this *mutual exclusion* property, it is safe to *pull* the shared memory associated with the lock *i* to the local copy at line 11. Before releasing the lock, the local copy is *pushed* back to the shared memory at line 14.

To prove that `acq_lock` eventually succeeds, from the fairness of $\mathcal{E}_{hs}$, we assume that between any two consecutive events from the same thread, there are at most *m* events generated by other threads (for some *m*). We also impose the following invariants on the environment:

**Invariant 1** (Invariants for ticket lock)**.** *An environment context that holds the lock i (1) never acquires lock i again before releasing it; and (2) always releases lock i within k steps (for some k).*

**Lemma 6** (Starvation-freedom of ticket lock)**.** *Acquiring ticket-lock in the mC2 kernel eventually succeeds.*

*Proof.* The full proofs are mechanized in Coq; here we highlight the main ideas. Let *n* be the maximum number of the total threads. Then (1) there are at most *n* threads waiting before the current one; (2) the thread holding the lock releases the lock within *k* steps, which generates at most *k* events; and (3) the environment context generates at most *m* events between each step of the lock holder. Hence there are at most $n \times m \times k$ events generated by the *context* of the threads waiting before the current one. Since the current thread belongs to this "context" and each read to the `now` field generates one `get_now` event, there are at most $n \times m \times k$ loop iterations at line 10 in Fig. 7. Thus, acquiring lock always succeeds. □

After we abstract the lock implementation into an atomic specification, each acquire-lock call in the higher layers only generates a single event "`t.acq_lock i`." We can compose such per-CPU specification with those of its environment CPUs as long as they all follow Invariant 1.

**MCS Lock** is known to have better scalability than ticket lock over machines with a larger number of CPUs. In mC2, we have also implemented a version of MCS locks [46]. 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 acquire and release.

**Physical memory management** introduces the page allocation table `AT` (with `nps` denoting the maximum physical page number). Since `AT` is shared among different CPUs, we associate it with a lock `lock_AT`. The page allocator is then refined into an atomic object where the implementation for each of its methods (e.g., `palloc` in Fig. 8) is proved to satisfy an atomic interface, with the proof that lock utilization for `lock_AT` satisfies Inv. 1. Once the atomic allocator is introduced, lock acquire and release for `lock_AT` are *not allowed to be invoked* at higher layers. Thus, in this layered approach, it is not possible that a thread holding a lock defined at a lower layer tries to acquire another lock introduced at a higher layer, i.e., the order that a thread acquires different locks is guided by the layer order that the locks are introduced. This implicit order of lock acquisitions prevents *deadlocks* in mC2.

Another function of the physical memory management is to dynamically track and bound the memory usage of each thread. A *container* object is used to record information for each thread (array `cn` in Fig. 8); one piece of information tracked is the thread's *quota*. Inspired by the notions of containers and quotas in HiStar [59], a thread in mC2 is spawned with some quota specifying the maximum number of pages that the thread will ever be allowed to allocate. As can be seen in Fig. 8, `palloc` returns an error code if the requesting thread has no remaining quota (lines 2 and 3), and the quota is decremented when a page is successfully allocated (line 13). Quota enforcement allows the kernel to prevent a denial-of-service attack, where one thread repeatedly allocates pages and uses up all available memory (thus denying other threads from allocating pages). From a security standpoint [16], it also prevents the undesirable information channel between different threads that occurs due to such an attack.

**Virtual memory management** provides consecutive virtual address spaces on top of physical memory management (see Fig. 6b), We prove that the primitives manipulating page maps are correct, and the *initialization procedure* sets up the two-level page maps properly in terms of the hardware address translation.

```
1  int palloc (uint tid) {
2    if (cn[tid].quota < 1)
3      return ERROR;
4    ▶acq_lock (lock_AT);
5    uint i=0,fp=nps;
6    while(fp==nps&&i<nps){
7      if (!AT[i].free)
8        fp = i;
9      i++; }
10   if (fp != nps) {
11     AT[i].free = 0;
12     AT[i].ref = 1;
13     cn[tid].quota --;
14   }
15   else fp = ERROR;
16   ▶rel_lock (lock_AT);
17   return fp;
18 }
```

**Figure 8:** Pseudocode of `palloc`

**Invariant 2.** *(1) paging is enabled only after all the page maps are initialized; (2) pages that store kernel-specific data must have the kernel-only permission in all page maps; (3) the kernel page map is an identity map; and (4) non-shared parts of user processes' memory are isolated.*

By Inv. 2, we show that it is safe to run both the kernel and user programs in the virtual address space when paging is enabled. In this way, memory accesses at higher layers operate on the basis of the high-level, abstract descriptions of address spaces rather than concrete page directories and page tables stored in the memory itself.

**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*.

**The shared queue library** abstracts the queues implemented as *doubly-linked lists* into *abstract queue states* (i.e., Coq lists). The local *enqueue* and *dequeue* operations are specified over the abstract lists. As usual, we associate each shared queue with a lock. The atomic interfaces for shared queue operations are represented by queue events "`t.enQ i e`" and "`t.deQ i`", which can be replayed to construct the shared queue. For instance, starting from an empty initial queue, if the current log of the $i$-th shared queue is $[\blacktriangleright, t_0.\texttt{enQ i 2}, \blacktriangleright, t_0.\texttt{deQ i}]$, and the event lists generated by the *environment context* at two switch points are $[t_1.\texttt{enQ i 3}]$ and $[t_1.\texttt{enQ i 5}]$, respectively, then the complete log for the queue $i$ is:

$$[t_1.\texttt{enQ i 3}, t_0.\texttt{enQ i 2}, t_1.\texttt{enQ i 5}, t_0.\texttt{deQ i}]$$

By replaying the log, the shared queue state becomes $[2,5]$, and the last atomic dequeue operation returns 3.

**Thread management** introduces the thread control block and manages the resources of dynamically spawned threads (e.g., quotas) and their meta-data (e.g., children, thread state). For each thread, one page (4KB) is allocated for its *kernel stack*. We use an external tool [12] to show

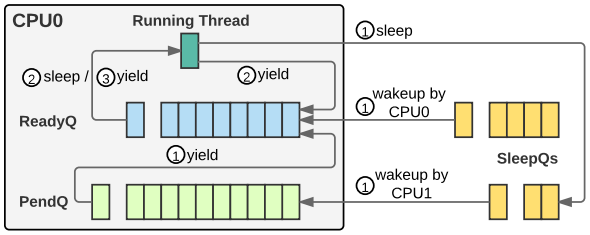



**Figure 9:** Scheduling routines `yield`, `sleep`, and `wakeup`

that the stack usage of our compiled kernel is less than 4KB, so stack overflows cannot occur inside the kernel.

One interesting aspect of the thread module is the context switch function. This assembly function saves the register set of the current thread and restores the register set from the kernel context of another thread on the same CPU. Since the instruction pointer register (`EIP`) and stack pointer register (`ESP`) are saved and restored in this procedure, this kernel context switch function is verified at the assembly level, and linked with other code that is verified at the C level and then compiled by CompCertX.

The thread scheduling is done by three primitives: `yield`, `sleep`, and `wakeup`. They are implemented using the shared queue library (cf. Fig. 9). Each CPU has a *private ready queue* ReadyQ and a *shared pending queue* PendQ. The context CPUs can insert threads to the current CPU's pending queue. The mC2 kernel also provides a set of shared *sleeping queues* SleepQs. As shown in Fig. 9, `yield` moves a thread from the pending queue to the ready queue and then switches to the next ready thread. The `sleep` primitive simply adds the running thread to a sleeping queue and runs the next ready thread. The `wakeup` primitive contains two cases. If the thread to be woken up belongs to the current CPU, then the primitive adds the thread to its ready queue. Otherwise, `wakeup` adds the thread to the pending queue of the CPU it belongs to. Except for the ready queue, all the other thread queue operations are protected by *fine-grained* locks.

**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 $\varepsilon_{ss}$. The scheduling primitives generate the `yield` and `sleep` events and $\varepsilon_{ss}$ responds with the next thread ID to execute. One invariant we impose on $\varepsilon_{ss}$ is that a sleeping thread can be rescheduled only after a `wakeup` event is generated. The second step is to introduce the *active thread set* to represent the *active* threads on the *active CPU*, and extend the $\varepsilon_{ss}$ with the *context threads*, i.e., 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 model with a single active thread on the active CPU (cf. Fig. 2).

```
1  struct fifobbq {
2    Queue insrtQ, rmvQ;
3    int n_rmv, n_insrt;
4    int front, next;
5    int T[MAX]; lock l;
6  } q;
7
8  void remove(){
9    uint cv, pos, t;
10   ▶acq_lock (q.l);
11   pos = q.n_rmv ++;
12   cv = my_cv ();
13   ▶enQ (q.rmvQ, cv);
14   while(q.front < pos ||
15       q.front == q.next)
16     ▶wait (cv, q.l);
17
18   t = q.T[q.front % MAX]
19   q.front ++;
20
21   cv=▶peekQ (q.insrtQ);
22   if (cv != NULL)
23       ▶signal (cv);
24   ▶deQ (q.rmvQ);
25   cv = ▶peekQ (q.rmvQ);
26   if (cv != NULL)
27       ▶signal (cv);
28   ▶rel_lock (q.l);
29   return t;
30 }
```

**Figure 10:** Pseudocode of the remove method for FIFOBBQ

**Starvation-free condition variable** A *condition variable* (CV) is a synchronization object that enables a thread to wait for a change to be made to a shared state (protected by a lock). Standard Mesa-style CVs [35] 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 version of CV using condition queues as shown by Anderson and Dahlin [7, Fig. 5.14]. However, we have found a bug in the FIFOBBQ implementation shown in that textbook: in some cases, their system can get stuck by allowing all the signaling and waiting threads to be asleep simultaneously, or the system can arrive at a dead end where the threads on the remove queue (rmvQ) can no longer be woken up. We fixed this issue by postponing the removal of the CV of a waiting thread from the queue, until the waiting thread finishes its work (cf. Fig. 10); the remover is now responsible for removing itself from the rmvQ (line 24) and waking up the next element in the rmvQ (line 27). Here, `peekQ` reads the head item of a queue; and `my_cv` returns the CV assigned to the current running thread.

## 5 Evaluation

**Proof effort and the cost of change** We take the certified sequential mCertiKOS kernel [23], and extend the kernel with various features such as dynamic memory management, container support for controlling resource consumption, Intel hardware virtualization support, shared memory IPC, single-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. During this development process, many of our certified layers (including their implementation, their functional specification, and the layer refinement proofs) have undergone many rounds of modifications and extensions.

CertiKOS makes such evolution process much easier. For example, all certified layers in the sequential kernel can be directly ported to the concurrent setting if they do not use any synchronization. We have also merged the work by Chen et al. [14] on the interruptible kernel with device drivers using our multicore model.

Overall, our certified mC2 kernel consists of 6500 lines of C and x86 assembly. We have also developed a general linking theorem for composing multiple threads running on the same CPU, and another theorem for combining programs running on different CPUs. Our team completed the verification of the new concurrency framework and features in about 2 person years.

Regarding specification, there are 943 lines of code used to specify the lowest layer axiomatizing the hardware machine model, and 450 lines of code for the specification of the abstract system call interfaces. These are in our trusted computing base. We keep these specifications small to limit the room for errors and ease the review process. Outside the trusted computing base, there are 5249 lines of additional specifications for the various kernel functions, and about 40K lines of code used to define auxiliary definitions, lemmas, theorems, and invariants. Additionally, there are 50K lines of Coq proof scripts for proving the newly-added concurrency features. At least one third of these auxiliary definitions and proof scripts are redundant and semi-automatically generated, which makes our proof a little verbose. For example, many invariant proofs get duplicated across the layers whenever there is a minor change to the entire set of invariants. We are currently working on a new layer calculus to minimize redundant definitions and proofs.

**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 10 in Fig. 7) was implemented as `while(▶L[i].now<t){}`. This passed all our tests, but during the verification, we found that it did not satisfy the atomic specification since the ticket field might overflow. For example, if `L[i].ticket` is $(2^{32}-1)$, `acq_lock` will cause an overflow (line 9 in Fig. 7) and the returned ticket `t` equals 0. In this case, `L[i].now` is not less than `t` and `acq_lock` returns immediately, which violates the order implied by the ticket. We fixed this bug by changing the loop body to "`while(▶L[i].now!=t){}`"; we completed the proof by showing that the maximum number of concurrent threads is far below $2^{32}$.

**Performance evaluation** Although the performance is not the main emphasis of this paper, we have run a number of micro and macro benchmarks to measure the speedup and overhead of mC2 and to compare mC2 to existing systems such as KVM and seL4. All experiments have been performed on an Intel Core i7-2600S (2.8GHz, 4 cores)

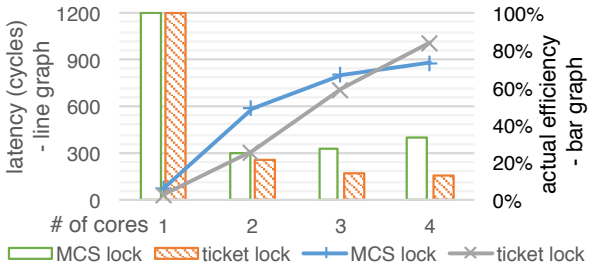



**Figure 11:** The comparison between actual efficiency of ticket lock and MCS lock implementations in mC2

with 8 MB L3 cache, 16 GB memory, and a 120 GB Intel 520 SSD. Since the power control code has not been verified, we disabled the turbo boost and power management features of the hardware during experiments.

**Concurrency overhead** The run-time overhead introduced by concurrency in mC2 mainly comes from *the latency of spinlocks* and *the contention of the shared data*.

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 10,000 consecutive lock acquires and releases (transactions) on each round.

Figure 11 shows the results of our latency measurement. In the single core case, ticket locks impose 34 cycles of overhead, while MCS locks impose 74 cycles (line chart). As the number of cores grows, the latency increases rapidly. However, note that all transactions are protected by the same lock. Thus, it is expected that the slowdown should be proportional to the number of cores. In order to show the actual efficiency of the lock implementations, we normalize the latency against the baseline (single core) multiplied by the number of cores ($\frac{n*t_1}{t_n}$). As can be seen from the bar chart, efficiency remains about the same for MCS lock, but decreases for ticket lock.

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

To reduce contention, all shared objects in mC2 are carefully designed and pre-allocated with a fine-grained lock. We design a benchmark with server/client pairs to evaluate the speedup of the system as more cores are introduced. We run a pair of server/client processes on each core, and we measure the total throughput (i.e., the number of transactions that servers make in each millisecond) across all available cores. A server's transaction consists of first performing an IPC receive from a channel $i$, then executing a payload (certain number of '`nop`' instructions), and finally sending a message to channel $i+1$. Correspondingly, a client executes a constant payload of 500 cycles, sends an IPC message to channel $i$, and then

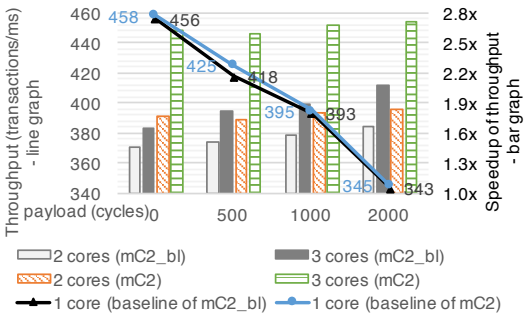



**Figure 12:** Speedup of throughput of mC2 vs. mC2-bl in a client/server benchmark under various server payloads (0-2,000)

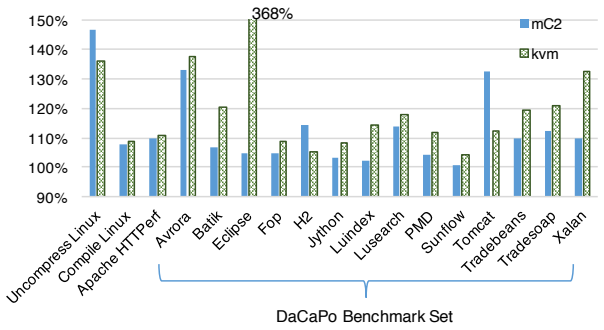



**Figure 13:** Normalized performance for macro benchmarks running over Linux on KVM vs. Linux on mC2; the baseline is Linux on bare metal; a smaller ratio is better

receives its server's message through channel $i+1$. When the client has to wait for a reply from the server, the control is switched to a special system process which then immediately yields back to the server process.

Figure 12 shows this server/client benchmark, comparing mC2 against a big-kernel-lock version of mC2 (mC2-bl). We insert a pair of lock acquire and release at the top-most layer by hand, and replace all fine-grained locks with an empty function. This does not introduce bias because the speedup is normalized against its own baseline (single core throughput) for each kernel version separately. From the figure, we can see that the speedup rate for big-kernel-lock is about 1.45x ~ 1.66x with 2 cores and 1.64x ~ 2.07x with 3 cores. On the other hand, the fine-grained locks of mC2 yield better speedup as the number of cores increases (roughly 1.77x ~ 1.84x and 2.62x ~ 2.71x with 2 and 3 cores, respectively). Note that the server/client pairs are distributed into different CPUs, and there is no cross core communication; therefore, one might expect perfect scaling as the number of cores increases. We did not quite achieve this because each core must execute some system processes which run at constant rates and consume CPU resources, and we did not align kernel data structures against cache-line size.

**IPC performance** We measure the latency of IPC send/recv in mC2 against various message sizes, and compare the result with seL4's IPC implementation.

A comparison of the performance of seL4 and mC2 is not straightforward since the verified mC2 kernel runs on a multicore x86 platform, while the verified seL4 kernel runs on ARMv6 and ARMv7 hardware and only supports single-core. Thus, we use an unverified, single-core version of seL4 for comparison. Moreover, the synchronized IPC API in seL4 (`Call/ReplyWait`) has a different semantics from mC2's send/recv: it uses a round-trip message passing protocol (with a one-off reply channel created on the fly) while trapping into the kernel twice, and it does not use any standard sleep or wakeup procedures. To have a meaningful comparison with respect to the efficiency of implementing system calls, we compare $(send + recv) \times 2$ of mC2 with $(Call + ReplyWait) + Null \times 2$ of seL4, where *Null* is the latency of a null system call in seL4.

We measure seL4's performance using seL4's IPC benchmark sel4bench-manifest [3] with processes in different address spaces and with identical scheduler priorities, both in *slowpath* and *fastpath* configurations. We consulted the seL4 team [27] and used 158 cycles as the cost of each null system call (*Null*) in seL4. To measure mC2's performance, we simply replace seL4's *Call* and *ReplyWait* system calls with mC2's synchronous *send* and *receive* calls. We found that, when the buffer size is zero, mC2 takes about 3800 cycles to perform a round trip IPC, while seL4's fastpath IPC takes roughly 1200 cycles, and seL4's slowpath IPC takes 1800 cycles. When the message size is larger than 2 words, the fastpath IPC of seL4 falls back to the slowpath; in the 10-words IPC case, mC2's round trip IPC takes 3820 cycles, while seL4 takes 1830 cycles. Note that seL4 follows the microkernel design philosophy, and thus its IPC performance is critical. IPC implementations in seL4 are highly optimized and heavily tailored to specific hardware platforms.

**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 13 contains a compilation of standard macro benchmarks: unpacking of the Linux 4.0-rc4 kernel, compilation of the Linux 4.0-rc4 kernel, Apache HTTPerf [47] (running on loopback), and DaCaPo Benchmark 9.12 [11]. 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 expose the raw disk interface to the guest via VirtIO [52] (instead of doing the pass-through), and its disk driver does not provide good buffering support.

## 6 Related Work

Dijkstra [18, 19] proposed to "realize" a complex program by decomposing it into a hierarchy of linearly ordered abstract machines. Based on this idea, the PSOS team at SRI [48] developed the Hierarchical Development Methodology (HDM) and applied it to design and specify an OS using 20 hierarchically organized modules. HDM was later also used for the KSOS system [50]. Gu et al. [23] developed new languages and tools for building certified abstraction layers with *deep* specifications, and showed how to apply the layered methodology to construct fully certified (sequential) OS kernels in Coq.

Costanzo et al. [16] showed how to prove sophisticated global properties (e.g., information-flow security) over a deep specification of a certified OS kernel and then transfer these properties from the specification level to its correct assembly-level implementation. Chen et al. [14] extended the layer methodology to build certified kernels and device drivers running on multiple *logical* CPUs. They treat 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 [33, 34] 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. [51] and von Tessin [55] 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 [55] further showed how to convert the single-core seL4 proofs into proofs for a BKL-based clustered multikernel.

The Verisoft team [49, 36, 4] applied the VCC framework [15] to formally verify Hyper-V, which is a widely deployed multiprocessor hypervisor by Microsoft consisting of 100 kLOC of concurrent C code and 5 kLOC of assembly. However, only 20% of the code is verified [15]; it is also only verified for function contracts and type invariants, not the full functional correctness property. There is a large body of other work [10, 58, 25, 13, 26, 56, 5, 54] showing how to build verified OS kernels, hypervisors, file systems, device drivers, and distributed systems, but they do not address the issues on concurrency.

Xu et al. [57] developed a new verification framework by combining rely-guarantee-based simulation [41] with Feng et al.'s program logic for reasoning about interrupts [21]. They have successfully verified key modules in the $\mu$C/OS-II kernel [1]. Their work supports preemption but only on a single-core machine. They have not verified any assembly code nor connected their verified C-like source programs to any certified compiler so there is no end-to-end theorem about the entire kernel. They have not proved any progress properties so even their verified kernel modules or interrupt handlers could still diverge.

## 7 Conclusion

We have presented a novel extensible architecture for building certified concurrent OS kernels that have not only 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 Jérémie Koenig, Tahina Ramananandro, Shu-Chun Weng, Liang Gu, Mengqi Liu, Quentin Carbonneaux, Jan Hoffmann, Hernán Vanzetto, Bryan Ford, Haozhong Zhang, Yu Guo, and Joshua Lockerman. We also want to thank our shepherd Gernot Heiser and anonymous referees for helpful feedbacks that improved this paper significantly. This research is based on work supported in part by NSF grants 1065451, 1521523, and 1319671 and DARPA grants FA8750-12-2-0293, FA8750-16-2-0274, and FA8750-15-C-0082. Hao Chen's work is also supported in part by China Scholarship Council. 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] The real-time kernel: µC/OS-II. http://micrium.com/rtos/ucosii, 1999 – 2012.

[2] The Coq proof assistant. http://coq.inria.fr, 1999 – 2016.

[3] The seL4 benchmark. https://github.com/smaccm/sel4bench-manifest, 2015.

[4] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Proc. 3rd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 40–54, 2010.

[5] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. CoGENT: Verifying high-assurance file system implementations. In *Proc. 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, 2016.

[6] T. Anderson. Private communication, Apr. 2016.

[7] T. Anderson and M. Dahlin. *Operating Systems Principles and Practice*. Recursive Books, 2011.

[8] A. Belay, A. Bittau, A. Mashtizadeh, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, 2012.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, pages 267–284, 1995.

[10] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[11] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. 21st ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190. ACM Press, Oct. 2006.

[12] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Proc. 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281, 2014.

[13] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *Proc. 25th ACM Symposium on Operating System Principles (SOSP)*, pages 18–37, 2015.

[14] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *Proc. 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 431–447, 2016.

[15] 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 (TPHOLs)*, pages 23–42, 2009.

[16] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proc. 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, 2016.

[17] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. In *Proc. 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 206–214, Jan. 1977.

[18] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, pages 341–346, May 1968.

[19] E. W. Dijkstra. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured programming*, pages 1–82. Academic Press, 1972. ISBN 0-12-200550-3.

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Dec. 1995.

[21] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. 2008 ACM SIGPLAN*

*Conference on Programming Language Design and Implementation (PLDI)*, pages 170–182, 2008.

[22] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.

[23] 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 (POPL)*, pages 595–608, 2015.

[24] 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.

[25] 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 *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, 2014.

[26] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *Proc. 25th ACM Symposium on Operating System Principles (SOSP)*, pages 1–17, 2015.

[27] G. Heiser. Private communication, September 2016.

[28] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Apr. 2008.

[29] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[30] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41 (2):37–49, 2007.

[31] M. F. Kaashoek. Parallel computing and the OS. In *Proc. SOSP History Day*, pages 10:1–10:35, 2015.

[32] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. McCarthy, J. Rafkind, S. Tobin-stadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL)*, pages 285–296, 2012.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.

[34] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), Feb. 2014.

[35] B. W. Lampson. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23 (2), Feb. 1980.

[36] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proc. 2nd World Congress on Formal Methods*, pages 806–809, 2009.

[37] X. Leroy. The CompCert verified compiler. http://compcert.inria.fr/, 2005–2016.

[38] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[39] W. Li, L. Gerard, and N. Shankar. Design and verification of multi-rate distributed systems. In *Proc. 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 20–29, 2015.

[40] H. Liang and X. Feng. A program logic for concurrent objects under fair scheduling. In *Proc. 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 385–399, 2016.

[41] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. 39th ACM Symposium on Principles of Programming Languages (POPL)*, pages 455–468, 2012.

[42] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *Proc. 24th International Conference on Concurrency Theory (CONCUR)*, pages 227–241. Springer-Verlag, 2013.

[43] J. Liedtke. On micro-kernel construction. In *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, 1995.

[44] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[45] A. Madhavapeddy, R. Mortier, C. Rostos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 461–472, 2013.

[46] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[47] D. Mosberger and T. Jin. Httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, Dec. 1998. ISSN 0163-5999.

[48] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: its system, its applications, and proofs. Technical Report CSL-116, SRI Computer Science Laboratory, May 1980.

[49] W. Paul, M. Broy, and T. In der Rieden. The Verisoft XT Project. http://www.verisoftxt.de, 2010.

[50] T. Perrine, J. Codd, and B. Hardy. An overview of the kernalized secure operating system (KSOS). In *Proc. 7th DoD/NBS Computer Security Initiative Conference*, pages 146–160, Sep 1984.

[51] S. Peters, A. Danis, K. Elphinstone, and G. Heiser. For a microkernel, a big lock is fine. In *APSys '15 Asia Pacific Workshop on Systems, Tokyo, Japan*, 2015.

[52] R. Russell. VirtIO: Towards a de-facto standard for virtual I/O devices. *Operating System Review*, 42 (5):95–103, July 2008.

[53] Z. Shao. Certified software. *Communications of the ACM*, 53(12):56–66, December 2010.

[54] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *Proc. 25th USENIX Security Symposium*, pages 87–104, 2016.

[55] M. von Tessin. *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, March 2013.

[56] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proc. 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, 2015.

[57] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. In *Proc. 28th International Conference on Computer-Aided Verification (CAV), Part II*, pages 59–79, 2016.

[58] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, 2010.

[59] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.