Provable Multicore Schedulers with Ipanema: Application to Work Conservation

Baptiste Lepers
baptiste.lepers@sydney.edu.au
University of Sydney

Redha Gouicem
Damien Carver
first.last@lip6.fr
Sorbonne Université, LIP6, Inria

Jean-Pierre Lozi
jean-pierre.lozi@oracle.com
Oracle Labs

Nicolas Palix
nicolas.palix@imag.fr
Université Grenoble Alpes

Maria-Virginia Aponte
aponte@cnam.fr
CNAM

Willy Zwaenepoel
willy.zwaenepoel@sydney.edu.au
University of Sydney

Julien Sopena
julien.sopena@lip6.fr
Sorbonne Université, LIP6, Inria

Julia Lawall, Gilles Muller
first.last@inria.fr
Inria, Sorbonne Université, LIP6

Abstract

Recent research and bug reports have shown that work conservation, the property that a core is idle only if no other core is overloaded, is not guaranteed by Linux's CFS or FreeBSD's ULE multicore schedulers. Indeed, multicore schedulers are challenging to specify and verify: they must operate under stringent performance requirements, while handling very large numbers of concurrent operations on threads. As a consequence, the verification of correctness properties of schedulers has not yet been considered.

In this paper, we propose an approach, based on a domain-specific language and theorem provers, for developing schedulers with provable properties. We introduce the notion of concurrent work conservation (CWC), a relaxed definition of work conservation that can be achieved in a concurrent system where threads can be created, unblocked and blocked concurrently with other scheduling events. We implement several scheduling policies, inspired by CFS and ULE. We show that our schedulers obtain the same level of performance as production schedulers, while concurrent work conservation is satisfied.

CCS Concepts: • **Software and its engineering → Formal software verification**.

EuroSys '20, April 27–30, 2020, Heraklion, Greece
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387544

Keywords: scheduling, formal verification, Linux

ACM Reference Format:
Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, and Julia Lawall, Gilles Muller. 2020. Provable Multicore Schedulers with Ipanema: Application to Work Conservation. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387544

1 Introduction

An OS-kernel thread scheduler[1] decides which thread runs at a given moment on a given core. A scheduler is a key OS service since any bad decision that it makes may lead to cores wasting cycles and may increase application response times [36]. Recent work has revealed bugs related to the violation of the work conservation property (no core remains idle when work is ready to be scheduled) in Linux's CFS scheduler [36], as well as a bug in recent versions of FreeBSD's ULE scheduler [6]. Indeed, developing a production scheduler is very challenging. A scheduler must operate under stringent performance constraints [28], support concurrency, adapt to complex hardware features, such as NUMA, and address not only scheduling of the threads on individual cores but also balancing the load across cores. As a result, while schedulers were services of a few hundred lines of code twenty years ago when machines were single core, they have grown into highly complex pieces of software in the era of multicore NUMA machines. The Linux kernel `kernel/sched` directory and associated scheduler header files contain more than 23,000 lines of code in Linux 4.19 (October 2018). The file `fair.c`, containing the code specific to the CFS scheduler, amounts to over 5,500 lines of code.

[1] We use *thread* to refer to a unit of scheduling, *i.e.*, either a lightweight *thread* or a single-threaded process. In Linux, the term *task* is used for this.

The conventional wisdom is that abnormal OS behavior can only be found by testing. Unfortunately, testing may miss scheduling issues, because schedulers are very sensitive to specific workloads and machine configurations. In fact, one of the Linux work conservation bugs remained undetected for more than 5 years and the ULE bug remained undetected for 2.5 years. It has been reported that Google incurred performance issues due to the scheduler on 25% of its disk servers [50]. These issues remained unnoticed for 3 years and cost millions of dollars. Furthermore, as part of this work we have found a new work conservation issue in CFS that occurs when unblocking a thread.

Recent work on seL4 [29] and CertiKOS [24] has opened an alternative to testing, showing that it is possible to prove an OS formally correct at design time. However, these approaches are made tractable by constrained concurrency models. SeL4 forbids concurrency in the kernel, while CertiKOS requires all accesses to shared variables, including reads, to be performed in critical sections. These constraints, however, do not match the requirements of a production multicore scheduler. Indeed, for performance and scalability, CFS is fully concurrent and allows cores to observe the instantaneous state of other cores without locks held, even if doing so may lead to decisions based on outdated values. Taking locks in order to read a consistent core state would severely degrade the performance of the whole system.

In this paper, we define a methodology for developing concurrent multicore schedulers that can be proved work conserving. Using our methodology, we have developed work-conserving schedulers that are inspired by those of Linux and FreeBSD. Our goal is to offer the same performance as production schedulers with the additional guarantee that cores are never wasted. There are several challenges in proving work conservation that require advances beyond the state of the art in proving OS code. First, we want to focus on proving properties of the scheduling algorithm itself, while still verifying source code that can be used in a real OS. Second, we want to reduce the proving effort required as compared to when starting from a low-level language such as C in which abstractions are hidden in low-level optimized code. Third, we need a concurrency model that allows a high degree of parallelism between scheduling events, and allows reasoning about work conservation properties of a scheduler while allowing threads to be concurrently created, blocked, unblocked or terminated.

To address the above challenges, we first propose a novel approach based on the identification of key scheduling abstractions and the realization of these abstractions as a Domain-Specific Language (DSL), Ipanema. While DSLs are typically proposed to facilitate programming in a particular domain [39], this aspect is secondary in our work. Instead, we rely on the fact that our scheduling abstractions have a clean semantics that is enforced by the DSL design and the DSL compiler. Expressing a scheduling policy in terms of these abstractions, via the Ipanema DSL, then implies that we can assemble the properties guaranteed by the abstractions into proofs of properties of the complete scheduling algorithm. Second, we define a "non-synchronized read" concurrency model that relies on the mutually exclusive execution of scheduling events locally on a core, but that still permits reading the state of other cores without requiring locks. Third, we show that work conservation cannot be ensured in a system where threads can be created, unblocked, blocked, or terminated concurrently with the execution of load balancing. We introduce *concurrent work conservation* (CWC), a property that is provable even in a concurrent system, under our "non-synchronized read" concurrency model. We find that an Ipanema CWC CFS-like scheduler is able to achieve the same performance as the original CFS scheduler.

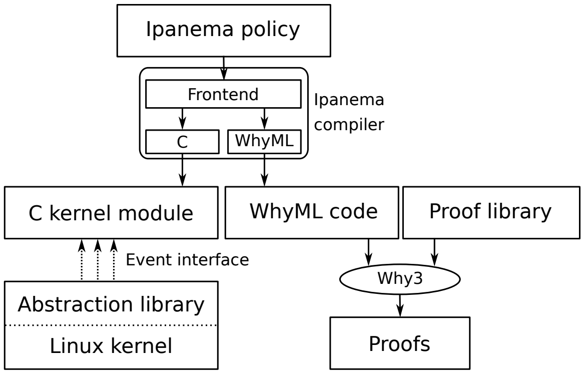


Figure 1. Ipanema overview: compilation to execution

Figure 1 gives an overview of our approach. First, the scheduler developer implements a policy using the Ipanema DSL. This policy is then subjected to two compiler backends: one generating efficient C code for execution as a Linux kernel module, and another generating code in WhyML, the ML-like imperative language supported by the state-of-the-art Why3 program verification platform [5]. The imperative features of WhyML allow a line-by-line correspondence between the WhyML code and the C code in the Linux kernel module. The C code is linked with a small library implementing our scheduling abstractions, while the WhyML code is used with Why3 to prove that the algorithm represented by the scheduling policy is CWC. The DSL enforces restrictions on the code structure to facilitate the proof of this property. Our DSL approach is inspired in part by Cogent [1], which targets filesystem development, and aims at simplifying the proving effort by generating part of the proof from the source code. We go one step further since our proofs exploit the domain-specific abstractions used in the scheduling code.

The contributions of this paper are:

- A methodology to ease the proving effort of CWC thanks to our scheduling abstractions and the Ipanema DSL.

- A methodology and compiler support to construct CWC proofs in the presence of unprotected read accesses to shared variables.
- A complete tool chain for compiling an Ipanema scheduling policy as a Linux 4.19 kernel module.
- Proved CWC ULE-like and CFS-like policies written in Ipanema.
- Identification of a new work conservation issue in CFS during thread unblocking that impacts highly parallel applications such as NAS.
- An evaluation of the performance of Ipanema on a set of established Linux scheduling benchmarks and on applications that stress the scheduler. On a large multicore machine with 160 hardware threads, we show that (i) the Ipanema scheduler improves performance on the NAS parallel benchmarks by up to 36% over Linux CFS because it achieves CWC and CFS does not, and (ii) on other workloads for which CFS does not exhibit any work-conservation issues, Ipanema schedulers perform similarly to CFS.

The rest of this paper is organized as follows. Section 2 presents issues in proving work conservation and introduces concurrent work conservation. Section 3 presents our scheduling abstractions and the DSL. Section 4 presents the proof techniques. Section 5 evaluates the performance of Ipanema schedulers. Section 6 presents related work, and Section 7 concludes.

2 Work Conservation

To optimize the utilization of resources, production schedulers for general-purpose multicore systems try to even the load across cores. In particular, schedulers try to assign newly runnable threads to the least loaded cores and perform load balancing, periodically and when a core becomes idle. *Work conservation* is the property that after such scheduling events, if a core on the machine is overloaded, then no core is idle.

We first describe some work-conservation bugs that have recently been identified in CFS and ULE. Then, we formally define work conservation and describe the challenges in implementing and proving a work-conserving scheduler in the context of a fully concurrent system. Finally, we introduce the concurrent work conservation property that addresses the concurrency challenges.

2.1 Work-conservation bugs

In a EuroSys 2016 study, Lozi et al. [36] showed four bugs in CFS that broke work conservation. CFS is a concurrent scheduler, and takes into account the NUMA hierarchy of the machine. Each level of the hierarchy defines a set of scheduling domains that share a common architecture feature (NUMA node, cache level). Each scheduling domain contains a set of groups of cores that are the scheduling domains for the next level below. Load balancing is carried out within a given scheduling domain. To minimize overhead, CFS frequently runs load balancing between cores on the same NUMA node, but only infrequently runs load balancing between cores located on different NUMA nodes. CFS also limits thread placement on thread creation or unblock to cores within a NUMA node. These optimizations contributed to the identified bugs.

The first bug stemmed from the fact that CFS only looks at the average loads of scheduling domains when performing load balancing instead of looking at the load of individual cores. CFS might consider that a NUMA node has a high load because it runs a high priority thread, even if most of its cores are idle. This bug caused a 13× slowdown on the NAS LU HPC benchmark. While the bug has recently been fixed, it had been present in the scheduler for more than five years (since Linux 2.6.37) when the study was published. Two other bugs were also caused by algorithmic and implementation errors, and had existed for 2-3 years (since Linux 3.9 and 3.18, respectively). The fourth bug was caused by the fact that periodic load balancing mistakenly moved long-running threads across nodes because of transient threads that happened to be scheduled at the moment the load balancer ran. The resulting overloading caused threads to block to wait for slower ones, hiding the node overload. To favor locality CFS does not migrate threads to idle cores on other nodes on an unblock, thus leaving the node in an overloaded state. To our knowledge, this bug has not been fixed at the time of writing and has been present since at least 2009.

As for ULE, a bug report in Dec. 2017 [6] showed that since the version released in Feb. 2015, no periodic load balancing was performed. The load balancing function was called by the kernel, but returned before doing any useful work because a variable was incorrectly initialized.

While the aim of Ipanema is not to find bugs in existing schedulers, but rather to present a way to produce correct schedulers, we found a work conservation issue in the process creation code of CFS when reimplementing it in Ipanema (Section 4.3).

These issues suggest that OS testing is not sufficient to find work conservation bugs, which can exist for a long time. Indeed, such bugs affect performance but do not cause a crash. Still, the consequences are important since the infrastructure is under-used, energy is wasted to power idle hardware, and users may observe a degraded response time.

2.2 Definitions

Multicore schedulers assign threads to cores as part of scheduling events. Events may place a single thread on a chosen core, as in the case of an `unblock` or `new` event, or may re-place any of the threads in the system, as is the case of a load balancing event. For `unblock` or `new`, we are concerned with whether the chosen core becomes overloaded, and we refer to the property as *local work conservation* (LWC). For load balancing, we are concerned with whether any core on the system becomes overloaded, and we refer to the property as *global work conservation* (GWC). We now define these properties more formally.

We say that a thread is *runnable* when it is scheduled or waiting to be scheduled. A core c is *idle* when it has no runnable threads, and *overloaded* when it has more than one runnable thread. Local and global work conservation are then defined as follows:

Definition 1 (Work Conservation (LWC, GWC))**.** For a given scheduling event, for any core c, let $O(c)$ hold iff core c is overloaded at the end of the scheduling event, and let $I(c)$ hold iff core c is idle at the end of the scheduling event. Then, a scheduling policy's implementation of the scheduling event is *local work conserving* for some core c iff at the end of the scheduling event:

$$O(c) \implies (\forall c'.\neg I(c'))$$

and is *global work conserving* iff at the end of the scheduling event:

$$(\exists c.O(c)) \implies (\forall c'.\neg I(c'))$$

2.3 Concurrent Work Conservation

In a fully concurrent scheduler such as CFS, work conservation may be impossible to achieve. Indeed, concurrent executions of the scheduler may cause a core to become idle or overloaded, and therefore make it impossible to achieve or prove WC as defined in the previous section. For instance, a core that is not overloaded when it is observed by a scheduling event may become overloaded before the end of the scheduling event due to a concurrent `unblock` or `new`. Likewise, in the case of load balancing, a thread selected for migration may block or terminate before the migration is performed. As the threads to migrate must be runnable, such a block or terminate may leave no thread available to migrate. After a scheduling event has observed a core, or even after it has placed threads on the core, all of the threads on the core can subsequently block or terminate, leaving the core idle. Finally, a concurrent schedule or yield event that may reposition a thread in the runqueue may also cause a core to appear to be idle (empty runqueue) if the reposition operation overlaps with the end of load balancing. We now introduce revised definitions of idle and overloaded that account for the occurrence of concurrent scheduling events. We then use these new definitions to craft a definition of work conservation that is achievable and provable.

Definition 2 (Concurrent Overloaded (CO) and Concurrent Idle (CI))**.** For a given scheduling event, for any core c, let $I(c)$ hold iff c is idle at the end of the scheduling event. Let $O(c)$ hold iff c is overloaded at the end of the scheduling event. Let $B(c)$ hold iff a thread `block` or `terminate` event that removes a thread from c overlaps with the scheduling event. Let $U(c)$ hold iff a thread `unblock` or `new` event that places a thread on c overlaps with the scheduling event, or, in the case of load balancing, if a thread is stolen for c from a core c' such that $U(c')$ (intuitively, in that situation we set $U(c)$ because the stolen thread might be the thread that was unblocked). Let $E(c)$ hold iff another scheduling event is in progress on core c at the end of the scheduling event. Then, for any core c,

$$\begin{aligned} CO(c) &\equiv O(c) \wedge \neg U(c) \\ CI(c) &\equiv I(c) \wedge \neg B(c) \wedge \neg E(c) \end{aligned}$$

We then propose the property of *Concurrent Work Conservation*:

Definition 3 (Concurrent Work Conservation (LCWC, GCWC))**.** A scheduling policy's implementation of a scheduling event is *local concurrent work conserving* for some core c iff at the end of the scheduling event:

$$CO(c) \implies (\forall c'.\neg CI(c'))$$

and is *global concurrent work conserving* iff at the end of the scheduling event:

$$(\exists c.CO(c)) \implies (\forall c'.\neg CI(c'))$$

In summary, CWC adapts WC by removing from consideration cores that may become idle or overloaded due to scheduling events unrelated to load balancing.

2.4 Hierarchical Concurrent Work Conservation

As described in Section 2.1, to reduce the cost of load balancing, CFS does not balance the load across the entire machine at every balancing operation, but rather periodically balances each domain in the hierarchy at different intervals. To capture this behavior, we provide a definition of hierarchical concurrent work conservation that reasons about each domain individually. Each group in a domain may comprise one or more cores. We generalize Definition 3 as follows:

Definition 4 (Hierarchical Concurrent Work Conservation (HCWC))**.** Let k be the number of cores in a group g. Let $O_k(g)$ hold iff g has more than k threads at the end of load balancing. Let $I_k(g)$ hold iff g has fewer than k threads at the end of load balancing. Let B_k, U_k, and E_k be defined like B, U, and E, but generalized from cores to groups. For example, $B_k(g)$ holds iff a thread block or terminate operation on any core in g overlaps with load balancing. Then, for a group g of k cores:

$$\begin{aligned} CO_k(g) &\equiv O_k(g) \wedge \neg U_k(g) \\ CI_k(g) &\equiv I_k(g) \wedge \neg B_k(g) \wedge \neg E_k(g) \end{aligned}$$

and a load balancer is *hierarchical concurrent work conserving* iff at the end of load balancing:

$$(\exists g.CO_k(g)) \implies (\forall g'.\neg CI_k(g'))$$

3 The Ipanema DSL

To achieve CWC, a scheduler's load balancer and its thread placement strategy for `unblock`/`new` must search for idle cores across the machine. To avoid blocking the machine, which would incur a high performance overhead, the state of other cores should be observed without taking locks, even though doing so may result in slightly inaccurate thread placement decisions. Linux's CFS is indeed based on such a "non-synchronized read" concurrency model to provide performance and scalability on large multicore machines.

Proving CWC with such non-synchronized reads requires enforcing constraints on the scheduler code to limit the kinds of inconsistencies and out-of-date information that may impact thread placement decisions. To address this issue, we propose to express a scheduling policy using a Domain-Specific Language (DSL). A DSL traditionally provides abstractions appropriate to a particular domain to make programming in the domain easier or more robust for domain experts [38, 39]. In our setting, however, the DSL makes it possible to introduce constraints on the structure of the code that make desired properties easier to verify and enforce. In particular, our DSL limits access to shared data structures in such a way as to make it possible to generate proof obligations that respect the concurrency semantics while incorporating sufficient invariants to make CWC provable on a range of scheduling policies.

In the rest of this section, we present our DSL, Ipanema, in terms of its main abstractions, its syntax, and its expressiveness. Ipanema captures the hierarchical load balancing algorithm of CFS, which is the most complex one of which we are aware and deals with cache affinity on NUMA machines. Additionally, we have been able to reimplement policies similar to ULE [7], lottery scheduling [53], and EDF [8].

3.1 Abstractions and concurrency

The main objects relevant to multicore scheduling are threads and cores. Threads have a state, indicating whether they are running, ready to run, blocked, or terminated. Ready threads wait to run on a specific core, and are stored in that core's runqueue. Threads may also be associated with policy-specific attributes, such as their expected load on the system or their priority. Cores may likewise be associated with attributes that summarize the attributes of the threads running or waiting on them, such as the total load of these threads.

Threads are affected by a series of scheduling events, such as the blocking of a running thread, the unblocking of a thread that is waiting for a resource, the need to select a new thread, and the balance of the load across cores. A scheduling policy must provide handlers for these events. To ensure the integrity of the scheduling state, at the core-local scope, events are executed in mutual exclusion. At the machine scope, Ipanema allows a core to observe the state of another core at any time. Not enforcing any synchronization on reads improves performance, but implies that the observed core can be in the middle of updating its state, potentially exposing inconsistent information.

To ensure the mutual exclusion of events at the core-local scope, a lock is associated with each core. We refer to this lock as the *core lock*. To reduce the set of properties that have to be proved, as well as easing the task of the policy designer, the Ipanema DSL does not provide any constructs for manipulating these locks. Instead, all locking actions are generated by the Ipanema compiler. To enable the reasoning needed to prove CWC, Ipanema puts substantial restrictions on the variables that can be observed during thread placement without holding the associated core lock. Such observations may only be performed by handlers of designated events: load balancing, thread unblock, and thread creation. Only four Ipanema-defined variables can be observed from another core without holding specific locks: the core attribute `cload`, which is the sum of the loads of the runnable threads on a given core, a core-specific counter of the number of runnable threads on a given core that is accessible via the function `count`, and the global variables `idle_cores` and `active_cores`, which contain the set of idle and non-idle cores in the system, respectively. Finally, the first two variables cannot be explicitly modified by the policy, but instead are updated by code inserted by the Ipanema compiler into the middle of the thread state transitions that impact their values. These restrictions ensure that it is possible to define invariants that describe the possible relationships between the observed values and the actual scheduling state, *e.g.*, the relationship between the value of a core's `cload` variable and the actual number of runnable threads on the given core. We assume a weak memory model similar to that of ARM [14]: reads and writes can be reordered by the underlying C compiler or by the processor, unless they are separated by a memory barrier; these are added as needed by the Ipanema compiler. Our assumptions would also hold on architectures with a stricter memory model, such as Intel's TSO.

Finally, for the purpose of placing lock operations, the Ipanema compiler decomposes the event handlers for load balancing, thread unblock, and thread creation into *observing* and *updating* phases. Observing phases examine shared attributes of other cores, but do not take the lock of those cores. Updating phases modify variables that can be observed from other cores. Such phases take the lock of the core associated with the modified variable. Updating phases are kept as short as possible to minimize the performance overhead.

3.2 Ipanema

Ipanema is based on a previous DSL, Bossa [40], which targets scheduling properties in a unicore environment. Bossa provides abstractions for defining scheduling properties of threads, thread states, and scheduling-event handlers. It does not provide abstractions for thread placement on `new` or `unblock` events, or for defining the load-balancing policy, as these are specific to a multicore setting.

We present Ipanema using a CWC variant of FreeBSD's ULE scheduling policy (Listing 1). We first describe thread and core definitions, and the scheduling-event handlers. We then give an overview of the notation used to express the

Listing 1 A CWC ULE-like scheduler

```
1  const int INTERRUPT = 1; const int REGULAR = 2;
2  const int INTERACTIVE = 4;
3  thread = { int load=1, prio, slice; core last_core; }
4  core = {
5      threads = {
6          shared RUNNING thread current;
7          shared READY set<thread> realtime:order = {highest prio};
8          shared READY set<thread> timeshare:order = {highest prio};
9      ...}
10     system shared int cload;
11 }
12 steal = {
13     can_steal_core(core src, core dst) {
14         src.cload − dst.cload >= 2
15     } => stealable_cores
16     do {
17         select_core() { first(stealable_cores order = {highest cload}) } => src
18         steal_thread(core dst, thread t) {
19             if (src.cload − dst.cload >= 2) {
20                 if (t.prio == REGULAR) t => dst.timeshare;
21                 else t => dst.realtime;
22             }
23         }
24     } until (runnable(dst) != 0)
25 }
26 handler (thread_event e) {
27     On tick {
28         e.target.slice−−;
29         if (e.target.slice <= 0) {
30             update_realtime(e.target);
31             if (t.prio == REGULAR) e.target => timeshare;
32             else e.target => realtime;
33         }
34     }
35     On yield { ... }
36     On block {
37         e.target => blocked;
38     }
39     On unblock {
40         thread t = e.target;
41         core idlest = choose_wakeup_core(e.target);
42         if (update_realtime(t)) t => idlest.realtime;
43         else t => idlest.timeshare;
44     }
45     On schedule {
46         thread t = first(realtime);
47         if (!valid(t)) t = first(timeshare);
48         t => current;
49         t.last_core = self;
50         t.slice = get_slice(t);
51     }
52     On new { ... } On detach { ... }
53 }
54 handler (core_event e) {
55     On synchronized balancing {
56         foreach (c in system_cores() order = {lowest cload}) {
57             steal_for(c);
58     } } }
59 void update_realtime(thread t) { ... }
60 int runnable(core c) { return count(c.realtime) + count(c.timeshare) + ...; }
61 int get_slice(thread t) { ... }
62 core choose_wakeup_core(thread t) {
63     /* Run interrupt threads on their core */
64     if (t.prio == INTERRUPT) { return t.last_core; }
65     foreach (g in t.last_core.d.groups) { /* Pick an idle cpu that shares a L2 */
66         if ((g.sharing_level & L2_CACHE) != 0) {
67             foreach (c in g.cores) { if (c.cload == 0) return c; }}}
68     return first(system_cores() order = { lowest cload }); /* Default */
69 }
```

load-balancing policy and the associated support provided by the runtime system. Finally, we present the support for a scheduling domain hierarchy. A BNF describing the main features of the language is shown in Appendix A.

Lines 1-3 declare thread attributes, comprising the thread load (always 1 for ULE), the thread priority (INTERRUPT, INTERACTIVE, or REGULAR for ULE), and the core on which the thread was last run. Lines 4-9 declare core attributes, including the thread states that are associated with the core and `cload`, a measure of the load on the core. Thread states are characterized by a state class: RUNNING for the state of the thread that is running on the core, READY for the state of the threads that are waiting in the core's runqueue, BLOCKED for the state of the threads that have blocked while running on the core, and TERMINATED for the state of the threads that have terminated while running on the core. The ULE policy maintains two runqueues, `realtime` (line 7) for INTERRUPT and INTERACTIVE threads and `timeshare` (line 8) for REGULAR threads. BLOCKED and TERMINATED states are not represented by any data structure.

The remainder of the scheduling policy defines the handlers for load balancing (lines 12-25 and 55-58) and the handlers and associated functions for the scheduling events (lines 26-53 and 59-69). The handlers are written in a C-like syntax, with a specific operator => for making thread state transitions, bounded loops, and a few other scheduling-specific constructs that help ensure the validity of the scheduler code.

Event handlers. To illustrate the structure of event handlers, we consider the ULE handlers for the `block` and `unblock` events (lines 36-44 of Listing 1). The `block` handler consists of a single DSL instruction that changes the state of the blocking thread `e.target` to `blocked` (the ULE state in the BLOCKED state class). The `unblock` handler consists of a series of DSL instructions that first select a new core (`choose_wakeup_core` call, line 41) for the unblocking thread, and then add the thread to the `timeshare` or `realtime` queue of the chosen core according to the thread's priority.

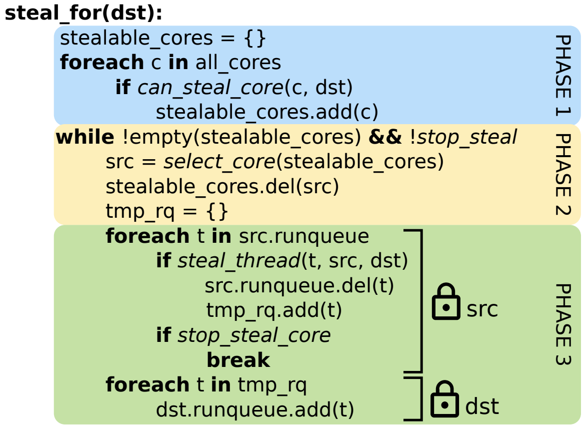
These simple specifications hide a more complex implementation. The state change shown in the `block` handler requires removing the thread from the data structure associated with its current state, updating any shared core attributes, *i.e.*, `cload` and the number of runnable threads on the core, and changing the state of the thread to `blocked`. Unblocking requires first observing the states of other cores to select a new core (`choose_wakeup_core`, line 41), and then performing a series of operations to change the state of the thread and install it in the chosen runqueue of the chosen core. By placing the updates to the shared variables after removing the thread from any data structure associated with its current state and before adding it to any data structure associated with its new state, the code generated by the Ipanema compiler respects the invariant that the shared variables always contain a value that overestimates the load on the core by at most the load of a single thread. This property is essential to reasoning about work conservation in the presence of concurrent scheduling events, as described in Section 4.

Load balancing. Load balancing involves observing the state of all of the cores of the system, to steal threads from more loaded cores and allow them to run on less loaded cores. The ULE-CWC load balancer tries to steal exactly one thread for each core that has fewer threads than some other core, whether or not the former core is idle. ULE-CWC, like ULE itself, has a single load balancer than steals for all of the cores on the machine. Indeed, all our CWC policies allow only one core to perform load balancing for the entire system, by initiating load balancing from the `synchronized balancing` event handler (lines 55-58). This strategy prevents one balancing core from stealing threads placed by another concurrent balancing core.

For a given destination core, Ipanema abstracts the load balancing strategy into three phases, triggered by a call to `steal_for(c)` (line 57). The first phase, `can_steal_core()` (lines 13-15 of Listing 1), collects a list of stealable cores. The second phase, `select_core()` (line 17), chooses a source core from which threads may be stolen from the list of stealable cores (if any). Finally, the third phase, `steal_thread()` (lines 18-23), migrates one or more threads `t` from the source core chosen by `select_core()` to the destination core.

Figure 2 shows the structure of the code generated by the Ipanema compiler from the three phases. `can_steal_core()` and `select_core()` are observing since they examine shared attributes of other cores. `steal_thread()` is updating, as it removes and adds threads in the runqueues of other cores. To minimize the performance overhead, the Ipanema compiler structures the code generated for `steal_thread()` such that all threads are removed from the selected core first, while holding only the lock of the selected core, and then all removed threads are moved to the destination core, while holding only the destination core's lock. As `select_core()` is executed without holding any other core's lock, the core it selects may no longer have stealable threads by the time of reaching `steal_thread()`. Our CWC policies perform the second and third phases in a loop (lines 16 and 24), so that balancing only fails if none of the cores identified by `can_steal_core()` has any threads to offer. Such a loop continues until the specified condition is satisfied (here, `dst` is not idle) or `select_core()` has considered all of the cores selected by `can_steal_core()`.

The expression of the load balancing policy in terms of the three phases allows the compiler to generate the optimized locking code found in Phase 3. It also factorizes the proof effort, as shown in Section 4, because the proof of the code shown in Figure 2 can be reused for all policies.



```
steal_for(dst):
    stealable_cores = {}
    foreach c in all_cores
        if can_steal_core(c, dst)
            stealable_cores.add(c)
    while !empty(stealable_cores) && !stop_steal
        src = select_core(stealable_cores)
        stealable_cores.del(src)
        tmp_rq = {}
        foreach t in src.runqueue
            if steal_thread(t, src, dst)
                src.runqueue.del(t)
                tmp_rq.add(t)
            if stop_steal_core
                break
        foreach t in tmp_rq
            dst.runqueue.add(t)
```

Figure 2. Structure of load balancing

Scheduling domain hierarchy. The Linux kernel supports the collection of cores into *scheduling domains* [46], that are typically defined according to the properties of the hardware caches and that are organized into a hierarchy. A scheduling policy may use the domains to maintain locality when choosing a new core for a given thread. The Ipanema DSL allows a scheduling policy to declare attributes of domains, such as the set of cores in the domain and the set of children of the domain (*groups* in CFS). The Ipanema DSL also provides looping operators that allow the load balancing, `unblock`, and `new` event handlers to iterate over the domains containing a given core, from the smallest, *e.g.*, containing the core itself, to the largest, *e.g.*, containing all of the cores on the machine.

3.3 Experience

The main limitations of the DSL are the constraints on which event handlers may contain an observing phase, *i.e.*, may observe shared variables, and on what shared variables may be observed (Section 3.1), and the restriction of loops to iterate over lists and thus be bounded. These limitations facilitate proofs, while still making it possible to express a wide range of scheduling policies.

A policy is compiled into a Linux kernel module, and loaded at runtime into the kernel. The frontend of the compiler amounts to around 37k lines of OCaml code, including the code for parsing and for validating generic and scheduler-specific correctness properties. The backend generating C code amounts to 4,200 lines of OCaml code.

Table 1 gives the size of the four Ipanema policies used in our evaluation. CFS-CWC is a CWC CFS-like scheduler that both exploits Linux domains and satisfies the HCWC property for load balancing. For load balancing, CFS-CWC achieves HCWC by iterating over all of the cores identified by `can_steal_core()` if necessary to find a thread to steal, rather than abandoning the search if the first selected core turns out to have no thread available, as done by CFS. CFS-CWC also removes the rather high (10-25%) imbalance thresholds imposed by CFS, instead stealing whenever any imbalance is observed. For `unblock` and `new`, CFS searches for idle cores

Table 1. Expansion ratio of the generated code

Policy	DSL LOC	C LOC	Ratio
CFS-CWC	362	1,526	x4.22
CFS-CWC-FLAT	222	1,267	x5.71
ULE-CWC	243	1,404	x5.78
ULE	260	1,382	x5.32

only within a single domain to prioritise locality. CFS-CWC, on the other hand, starts the search following the locality heuristics of CFS, but, if none is found, searches for an idle core elsewhere on the machine before resorting to a core that is already running a thread. CFS-CWC-FLAT is a simplified version of CFS-CWC that considers a flat topology in which all cores belong to the same domain. ULE is a port in Ipanema of the FreeBSD ULE scheduler. Its `new` and `unblock` handlers are already work conserving, but its load balancer steals from a core at most once per balance and thus can leave cores idle when other cores are overloaded. ULE-CWC is a CWC version of our ULE scheduler that removes the latter restriction. These policies rely on the Ipanema abstraction library, consisting of 1,900 lines of C code, and on the existing scheduling and context-switch infrastructure of `kernel/sched`. In contrast to the 23,000 lines of code found in the `kernel/sched` of Linux 4.19, Ipanema isolates the scheduling policy in a single, easily identifiable place.

The CFS-like policies implemented in Ipanema aim to reproduce the core scheduling and placement heuristics used in CFS, but do not reimplement all of CFS's heuristics and features. Most notably, we have not implemented cgroups and NUMA-aware data migration. The generated C files are 3 to 5 times larger than the Ipanema ones (see Table 1). Part of the explanation is that we care about the human readability of the generated C code which helps debugging. Still this ratio gives an estimation of the robustness and productivity gain in using a DSL.

4 Proving CWC

Our goal is to prove concurrent work conservation properties of Ipanema code implementing load balancing, unblocking, and placement of new threads. The key challenge in our proofs is the lockless interaction between the observing phases of load balancing, `unblock`, and `new`, and the updating phases of the event handlers running on other cores. The proofs benefit from the constraints on the scheduling code imposed by the DSL and the DSL compiler.

Our proofs are carried out using the Why3 [5] platform for deductive verification. Why3 reasons about code written in WhyML, an ML variant containing imperative features and notations for expressing invariants. Given WhyML code annotated with pre- and post-conditions, loop invariants, and any needed assertions, Why3 uses weakest pre-conditions to generate conditions that must be verified to ensure that the post-conditions hold. The user can prove these verification conditions using a large selection of off-the-shelf theorem provers; we use Alt-Ergo [15] and CVC4 [4].

To interface with Why3, in addition to generating C code for execution, a separate backend of the Ipanema compiler also generates WhyML code (see Figure 1). Due to the imperative features of WhyML, the WhyML code generated by the Ipanema compiler closely tracks the generated C code, with typically a line-by-line correspondence. The Ipanema compiler also generates the pre- and post-conditions that correspond to CWC.

Using a DSL and WhyML code allows us to reduce the proving effort compared to proving C code directly. The C code works on complex data structures and makes heavy use of aliases, pointer logic and external code. Proving properties of such code requires substantial reasoning about raw memory, to show well-definedness properties of accesses to data structures, even though such properties are already guaranteed by our DSL. Furthermore, the DSL restricts the usage of shared variables to specific places in specific event handlers, which makes the proofs in the presence of concurrent scheduling events tractable.

Our approach relies on a trusted computing base consisting of Why3, its underlying theorem provers, and the Ipanema compiler. This trusted computing base can be reduced by relying on recent advances in certified compilers [17, 33]. Furthermore, we assume that the rest of the kernel obeys the model that we have developed for its interaction with the scheduler. In particular, we assume that the kernel initiates load balancing periodically.

We first present how we model the behavior of the updating phases of the event handlers, and then consider how to prove CWC.

4.1 Modeling scheduling event-handler behavior

The event-handler code is executed concurrently on multiple cores, with no locks ensuring synchronization across the different cores. Accordingly, CWC proofs must consider possible interleavings of the individual updates and reads to shared variables executed by these events, according to our targeted weak ARM-like memory model or a stricter model.

To create a model of the possible interleavings between the event handlers, the Ipanema compiler first extracts the updating phase of each handler, consisting of the state change operations performed by the handler, any updates to the designated shared variables `cload`, the thread count, `idle_cores`, and `active_cores`, as well as their control-flow and data-flow dependencies. It then subdivides each updating phase into a sequence of *updating fragments*, each implemented as a function containing a single thread-state or shared-variable update. To respect the control-flow dependencies within a given event handler, each such function returns a *continuation* [54], *i.e.*, a data structure encapsulating the identity of the next instruction to execute and information about any local state. Characterizing the effect of concurrent execution of the event handlers then amounts to characterizing the effect of all interleavings of the updating fragments that respect the control-flow represented by the continuations. To simulate these interleavings, we implement a loop that randomly chooses a core and a continuation and allows the scheduling state on that core to advance according to the function indicated by the continuation, if the randomly chosen continuation is the same as the one currently associated with the core. The characterization of the effect of the concurrent execution of the event handlers is then derived from the invariants of this loop.

The interleavings that must be considered depend on the locks that are held by the different phases of the load balancing, `unblock` and `new` events. The compiler generates three WhyML functions that simulate different kinds of interleavings:

- `others()` simulates interleavings of the updating fragments of all event handlers on all cores. The post-condition of this function represents what can be observed when not holding the lock of any other core.
- `others_and_synchronize(`c`)` simulates interleavings of the updating fragments of all event handlers on all cores and ensures that core c ends up outside of any event handler. The post-condition of this function represents what can be observed after taking the lock of c.
- `others_except(`c`)` simulates interleavings of the updating fragments of all event handlers on all cores except core c. The post-condition of this function represents what can be observed while holding the lock of c.

The Ipanema compiler then places calls to `others()`, `others_and_synchronize()`, and `others_except()` in the WhyML code before operations that observe the scheduling state of other cores, according to the held locks. For example, for load balancing, a call is added to `others()` in Phase 1 (Figure 2), to `others_and_synchronize(`*src*`)` just before the first `foreach` in Phase 3, to `others_except(`*src*`)` at the start of the body of the first `foreach` in Phase 3, to `others_and_synchronize(`*dst*`)` just before the second `foreach` in Phase 3, and to `others_except(`*dst*`)` at the start of the body of the second `foreach` in Phase 3.

The principle challenge in reasoning about the concurrent behavior of the event handlers is that the update of the `cload` variable and the thread-count variable (that can be read from another core by the load balancing, `unblock`, and `new` handlers of the policy) and the movement of threads into and out of the runqueue (which determines whether a core is actually idle) are not done atomically. Thus, the value of `cload` and/or the thread-count variable can be stale. As noted in Section 3.2, the Ipanema compiler ensures that the code for a thread state change operation is always generated such that the `cload` or thread-count variable of core c either accurately reflects the set of runnable threads on these cores, or has a greater value. We define the predicate $synch(c)$ such that $synch(c)$ is satisfied if and only if the former case holds. In the latter case, when $\neg synch(c)$, `cload` is greater than the actual load of the runnable threads by the load of the thread that is changing state and/or the thread-count variable is greater than the number of runnable threads by 1. For example, on the state change operation in a block event, the code generated by the Ipanema compiler first removes the blocking thread from the runqueue, and then reduces the core's thread count and `cload` accordingly; the compiler enforces this ordering with a barrier. With respect to our definition of CWC, if $\neg synch(c)$ holds at the end of load balancing, then the predicate $E(c)$ in the definition of CWC is true.

Beyond the need to take into account the impact of concurrent updates on non-synchronized reads, verifying CWC requires knowing whether `block` (or `terminate`) or `unblock` (or `new`) events have occurred on other cores since the beginning of the load balancing, `unblock`, or `new` operation being verified. To maintain this information, the Ipanema compiler generates *ghost code* [22]. Such code does not map to any C code but rather is included in the WhyML code only to enable a proof. Ghost code is available in a number of proof tools in addition to Why3, such as Dafny [31] and Leon [30]. Concretely, the Ipanema compiler instruments the generated WhyML code to maintain boolean maps B and U that record the set of cores on which threads have blocked or unblocked, respectively, since the start of the execution of the CWC operation.

Given the above generated code, the scheduler developer must prove that the relations shown in Figure 3 hold between the state before and after calling each of `others()`, `others_and_synchronize()`, and `others_except()`. In this figure, $|c|$ is the number of runnable threads on the core after the call, and $|c|^-$ is the number of such threads before the call. Continuing with the example of a thread blocking on core c, when the event handler starts, $synch(c)$ holds. After removing the thread from the runqueue $\neg synch(c)$, $B(c)$, and $|c| < |c|^-$ hold. After updating `cload` and the thread-count variable, $synch(c)$ should hold again, but $|c| < |c|^-$. As $B(c)$ is still true, this situation satisfies $synch(c)$. Finally, changing the state of the blocking thread to indicate that it has blocked has no effect on $synch(c)$, $B(c)$, and the value of $|c|$. Further blocking events on the same core will further reduce the number of runnable threads on the core. `unblock` events may raise the number of threads on the core as compared to the original number, but $U(c)$ will also be set in this case. The Ipanema compiler generates post-conditions expressing these relations in each of the functions generated from the updating fragments of the event handlers and in the definitions of `others()`, etc.

The scheduler developer must also prove that the interleaving of events does not lose or gain any threads, except by `terminate` and `new` events, respectively. The Ipanema compiler includes post-conditions expressing this property in the generated functions.

		Ending state	
		$synch(c)$	$\neg synch(c)$
Starting state	$synch(c)$	$\lvert c\rvert = \lvert c\rvert^{-} \lor$ $(B(c) \land \lvert c\rvert < \lvert c\rvert^{-}) \lor$ $(U(c) \land \lvert c\rvert > \lvert c\rvert^{-})$	$\lvert c\rvert = \lvert c\rvert^{-} - 1 \lor$ $(B(c) \land \lvert c\rvert < \lvert c\rvert^{-}) \lor$ $(U(c) \land \lvert c\rvert \geq \lvert c\rvert^{-})$
	$\neg synch(c)$	$\lvert c\rvert = \lvert c\rvert^{-} + 1 \lor$ $(B(c) \land \lvert c\rvert \leq \lvert c\rvert^{-}) \lor$ $(U(c) \land \lvert c\rvert > \lvert c\rvert^{-})$	$\lvert c\rvert = \lvert c\rvert^{-} \lor$ $(B(c) \land \lvert c\rvert < \lvert c\rvert^{-}) \lor$ $(U(c) \land \lvert c\rvert > \lvert c\rvert^{-})$

Figure 3. Relation between *synch*(c) and the number of threads on core c before and after an updating fragment

4.2 Load balancing, unblock and new

Load balancing. For load balancing, the Ipanema policy developer only provides the strategies for collecting relevant cores (`can_steal_core()`), selecting a core from which to steal (`select_core()`), and the operations to perform when stealing (`steal_thread()`). The Ipanema compiler generates WhyML functions and associated lemmas describing this code and its expected behavior. The main part of the load balancing algorithm is represented by the code in Figure 2. From this code, the Ipanema compiler generates a *proof skeleton*, parameterized by the `can_steal_core()`, etc. operations. According to the definition of global CWC (Definition 3), the proof skeleton proves as a post-condition the following property of the system state `p` at the end of load balancing:

```
(exists c:int. overloaded p c ∧ not(u[c])) →
    (forall c':int. not(idle p c') ∨ b[c'] ∨ not(synch p c'))
```

Proving the result of instantiating the proof skeleton with the definitions of our CFS and ULE-like CWC policies is straightforward using the automation provided by Why3.

Unblock and new. For `unblock` and `new` events, the Ipanema policy developer provides the complete implementation, and thus no proof skeleton can be provided. The Ipanema compiler translates the provided handlers into WhyML code, annotated with a post-condition expressing the LCWC property (Definition 3), where `p` represents the system state at the end of the event and `c` represents the chosen core:

```
overloaded p c ∧ not(u[c]) →
    (forall c':int. not(idle p c') ∨ b[c'] ∨ not(synch p c'))
```

The `unblock` code of ULE-CWC (lines 62-69 of Listing 1) performs two searches, first within the hierarchy to find an idle core among the cores sharing an L2 cache with the core on which the thread previously blocked, and then if the first search fails, across the entire system to find the least loaded core. To represent this algorithm, the Ipanema compiler generates around 250 lines of code. Currently, the scheduler developer must place loop invariants in this code manually in order to carry out the proof. As thread placement algorithms typically involve simple searches over sets of cores, we are exploring whether the Ipanema compiler can further generate some of the required loop invariants [2]. The treatment of the `new` event-handler code is similar.

4.3 CWC bugs found while writing policies

In developing our Ipanema CWC scheduling policies, we made a few errors, for example, in the criteria used by `can_steal_core()` in the load-balancing algorithm. Trying to verify the Ipanema load-balancing proof skeleton based on these incorrect definitions showed that some of the verification conditions generated by Why3 could not be proved by the available solvers. While no counterexamples were provided, knowing which verification condition fails helped us find examples showing that our definitions were not CWC.

We additionally tried to prove the local CWC property of CFS's `new` and `unblock` thread placement algorithms. These proofs also fail, because CFS's algorithms never check for idle cores across the entire machine, but focus on the domain of the core of the parent and the domain of the core where the thread ran previously, respectively. As a consequence, a NUMA node might become overloaded even when the rest of the machine is mostly idle. Again, examining the verification conditions that were not provable helped to understand the problem. This issue in `unblock` is the root cause of the fourth WC bug described by Lozi et al. [36] that remains unfixed. In Section 5.2, we see the practical impact of the issue in `new` on the NAS benchmarks, where a large number of threads are created at the same time and have to be placed on a large number of cores.

4.4 Proof assessment

In summary, proving a scheduling policy CWC requires (i) proving that the interleavings of the updating phases of the event handlers respect the starting and ending state properties defined at the end of Section 4.1, (ii) proving that the definitions in the load balancing algorithm allow proving the load-balancing proof skeleton, and (iii) proving that the code provided for `unblock` and `new` satisfies LCWC. Our proofs are built on a library of useful operations on threads and states, amounting to around 2000 lines of WhyML code, developed part time over 1 person-year, while learning how to use Why3. The load balancing proof skeleton is around 700 lines of code and was developed in 4 person-months of full time effort. Why3 is normally run interactively, in a dedicated IDE, making it difficult to estimate the proving time experienced while developing the proof. Each completed CWC proof for the policies we considered (ULE-CWC and CFS-CWC) could be replayed in around 10 minutes on a 44-core server (2.20GHz, 256GB RAM). This time represents the time for generating the verification conditions and invoking and running the solvers.

5 Performance Evaluation

In this section, we evaluate the performance of Ipanema schedulers with respect to production schedulers such as CFS. We aim to evaluate the potential overhead of CWC schedulers. There are two potential sources of overhead: (i) the use of locks in the updating phases of events; as locks are placed automatically, their use may be less efficient than in the hand-optimized CFS code and (ii) the fact that CWC implies suppressing load balancing concurrency, in contrast to vanilla CFS which supports concurrent load balancing.

Our results show that CWC policies improve performance on benchmarks that exhibit a lack of work conservation with CFS. On other benchmarks, our DSL-based approach is on par with CFS and ULE, and the potential sources of overhead do not negatively impact performance, even on a machine with a large number of cores.

5.1 Experimental setup

We perform our evaluation on a 4-socket Xeon E7-8870 v4 machine (80 cores/160 hardware threads) with 512GB of RAM, running the Debian Buster OS. In the rest of this section, for simplicity, we refer to hardware threads as cores. Experiments were done with the Linux governor in *performance* mode to remove the effects of dynamic frequency scaling. We run a modified Linux 4.19 kernel that supports Ipanema schedulers. Modifications include adding a softirq used for the load balancing event, which is called every 4ms (the default duration of a tick in Linux), and modifying fastpaths that assume that if CFS has no thread to run, then the machine is idle. In total, fewer than 20 lines of code had to be modified. The modifications do not impact the performance of CFS or Ipanema.

We evaluate five schedulers, CFS, CFS-CWC, CFS-CWC-FLAT, ULE, and ULE-CWC. CFS is Linux's vanilla scheduler of the 4.19 kernel, used as a baseline comparison. CFS is written in C. The Ipanema schedulers are those described in Section 3.3. As workloads, we use benchmarks from the NAS benchmark suite [3], as well as Kbuild and Sysbench. These benchmarks stress schedulers by heavily creating and/or blocking and unblocking threads.

The NAS benchmark suite is composed of parallel scientific kernels. We exclude I/O-based kernels because they exhibit a high standard deviation on our machine, and keep all of the applications (BT, CG, EP, FT, IS, LU, MG, SP, and UA) that are dominated by computations and synchronizations (*e.g.*, barriers). The NAS benchmarks are challenging from a work conservation point of view, because they create and unblock many threads at once. Indeed, they exhibit a work conservation issue (see Section 4.3) with CFS that our CWC policies are able to resolve almost completely.

The other benchmarks in this evaluation do not exhibit work conservation issues, but they are useful to evaluate potential overheads of CWC policies relative to production OS schedulers such as CFS and ULE. Sysbench [51] is a scriptable benchmark tool that includes OLTP benchmarking. We run Sysbench on two databases, MySQL 8.0.15, and MongoDB 4.1.8, using a mix of read/write OLTP queries to evaluate request latency and throughput. Sysbench and the database threads share all the machine cores, and the database is stored in memory using a `ramfs` partition. Kbuild is a parallel batch application that builds the Linux kernel using `make` according to the configuration obtained with `make defconfig` with different numbers of jobs. The kernel source tree is placed in a `ramfs` partition in order to avoid physical I/O. For all experiments, the results presented are the mean of 12 runs.

5.2 NAS: solving work-conservation issues

We first assess the efficiency of thread placement in terms of work conservation. For this, we use the NAS benchmarks, which are run with 160 threads, *i.e.*, the same number of threads as cores. Overall, all Ipanema schedulers perform better than vanilla CFS as shown in Figure 4. The geometric mean improvement is 15.5% for ULE, 11.8% for CFS-CWC, 14.0% for CFS-CWC-FLAT, and 14.4% for ULE-CWC, showing that the implementation strategies used by the Ipanema compiler are sufficient to give good performance.

The lower performance of CFS is due to the work conservation issue on a `new` event highlighted in Section 4.3. To identify the issue, we recorded and compared the size of the runqueues using the scheduler profiling tools SchedLog and SchedDisplay that we developed in previous work [9]. Figure 5a shows the beginning of the execution of FT under CFS which is representative of the placement problem. Gray lines mean that there is only one thread on the core and the thread is running, while red lines mean that the core is overloaded, *i.e.*, one thread is running and at least one other thread is runnable. CFS places all new threads on the same NUMA node as the thread that performed the thread creation. As a consequence, initially, all threads are created on a single NUMA node (cores 120-159), and the cores of this NUMA node become overloaded while the rest of the machine is idle. It is then up to the periodic load balancer to balance the load, which is done in multiple rounds, one per scheduling domain of the hierarchy. Therefore, it is only at 0.5s that all of the cores get a thread, with the load balancer sometimes making placement errors later. The fact that cores are overloaded causes some threads to reach barriers late, which increases the time that all threads must spend in them. Therefore, the barriers are clearly visible in Figure 5a. Figure 5b shows the same execution with CFS-CWC. There are only two placement errors at startup which are due to concurrent unblocks choosing the same core, which can introduce a WC issue but is tolerated by CWC. Subsequently, no core is overloaded, so the system is work conserving. Threads execute in a much more synchronized way and spend less time in barriers. Note that the problem of concurrent unblocks potentially occurs

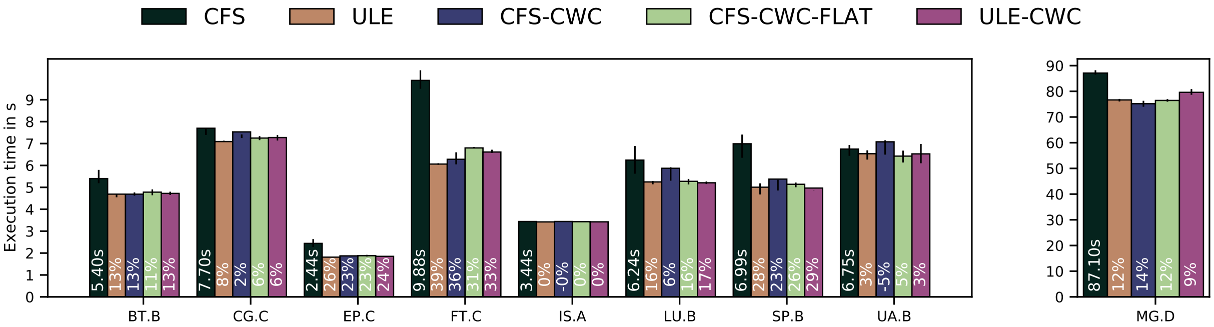
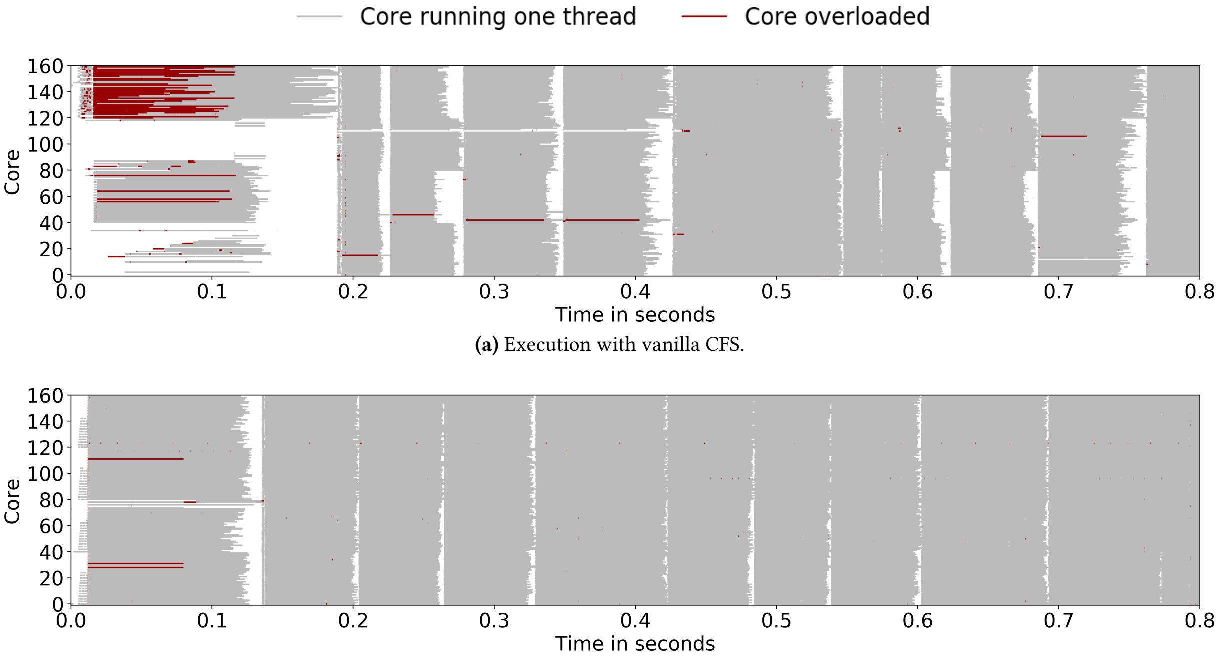


Figure 4. Performance of NAS with 160 threads



(a) Execution with vanilla CFS.

(b) Execution with CFS-CWC.

Figure 5. First 0.8s of the execution of NAS FT.C

whenever multiple threads are woken simultaneously. Therefore, it impacts workloads that have many threads waiting on locks.

On UA, CFS-CWC is to 5% slower than CFS even though CFS again suffers from the same bad placement problem on the first unblock, delaying threads at barriers. Profiling revealed that load balancing was not the issue and we investigated locality since some NAS applications are known to be placement sensitive [16]. To confirm this hypothesis, we ran the NAS applications, pinning threads to cores and varying the initial placement of threads. We observe performance differences of up to 33% between placements.

Heuristics that lead to work conservation violations are hard to detect using standard profiling tools. They do not cause the system to crash or hang, but eat away at performance. We found the work conservation issue exhibited by FT when reimplementing CFS in Ipanema. Our proofs indicated a possible work conservation problem in the code of the `unblock` and `new` events, which was easy to understand once it was pointed out.

5.3 Performance impact on other workloads

We now evaluate the performance impact of our DSL-based approach on workloads that are scheduling intensive but do not exhibit work conservation issues. We first run Kbuild with up to 256 concurrent jobs (see Figure 6a). On our machine with 160 cores, for all schedulers, there is an increase of performance up to 128 jobs. With more jobs, performance worsens a little and then stays stable. While CFS is less efficient, the maximum difference compared to Ipanema schedulers is less than 6%. The small gains in performance are explained by the thread placement strategy when a thread is unblocked. Our CWC policies use a more aggressive strategy, trying to place threads on idle cores. As seen in NAS, CFS favors preserving cache locality over work conservation and delays thread migrations.

We then benchmark the MySQL and MongoDB databases, which are highly demanding in terms of locking. Figures 6b and 6d present the performance in terms of throughput for the OLTP benchmarks, while Figures 6c and 6e present the 95^{th} percentile of response times. For MySQL with 64 clients or more, CFS performs a little worse than all of our Ipanema schedulers in terms of throughput, with the highest difference being 8.2% (compared to CFS-CWC with 128 clients). For MongoDB, all schedulers perform similarly, with performance differences reaching at most 3%. In terms of latency, all schedulers also exhibit small differences, with at most 11% for MySQL. We believe that the differences are again due to the placement strategy in `unblock` events, which occur at a high rate in these benchmarks.

A potential overhead of our CWC policies is that, unlike in CFS, load balancing is not concurrent: one core performs load balancing for the entire machine. The CFS-CWC-FLAT policy pushes this to an extreme; as it considers that all cores belong to a single domain, each load balancing tries to steal for each core from all of the other cores. This thus represents a worst case bound on the Ipanema load balancing cost. In our experiments, we find that CFS-CWC-FLAT is sometimes better and sometimes worse than the others. On the one hand, the single domain slows down load balancing as compared to CFS(-CWC) since all of the cores are observed during load balancing. On the other hand, load balancing is done in a single round, while CFS(-CWC) may take several rounds (one per hierarchy domain) to distribute thread across the entire machine. The results across all of the benchmarks show no clear winning strategy on our 160 core machine.

6 Related Work

Kernel correctness. Testing is the conventional approach to improve kernel correctness. Linux relies on various test suites [34] and community testing to detect bugs. Initiatives have also been deployed to more comprehensively assess kernel performance. For instance, the Linux Kernel Performance project [13] has been created to detect performance regressions in the Linux kernel, and work has been done to automatically detect system calls that take an “abnormal” amount of time [43, 48]. While obvious design flaws can be detected, more subtle bugs or bugs that happen on certain hardware may be easily missed.

Model checking has also been used to improve kernel correctness by finding bugs that lead to crashes [41], errors in network control planes [19] and code paths that lead to deadlocks [57]. Model checking of schedulers is challenging due to the combinatorial blow up of the state space due to interactions between a potentially large number of threads and cores.

Recent work has focused on constructing formal specifications of operating systems and proving that the implementation follows the specification. SeL4 was the first fully specified and verified micro-kernel running on a single-core machine [29]. CertiKOS [24] is a verified modular micro-kernel that supports concurrency. However, these two systems have not verified their schedulers to be work conserving. CertiKOS also forbids concurrent accesses to shared variables. Hyperkernel [42] provides a push-button verification system for operating systems, but does not offer any support for concurrency. Other work has been done in better specifying individual behaviors, such as avoiding buffer overflows, accesses to shared variable outside of critical sections, or deadlocks [18, 20, 21, 35, 37, 44, 45, 52]. Like these approaches, we do proofs on concurrent code by verifying invariants on concurrent events.

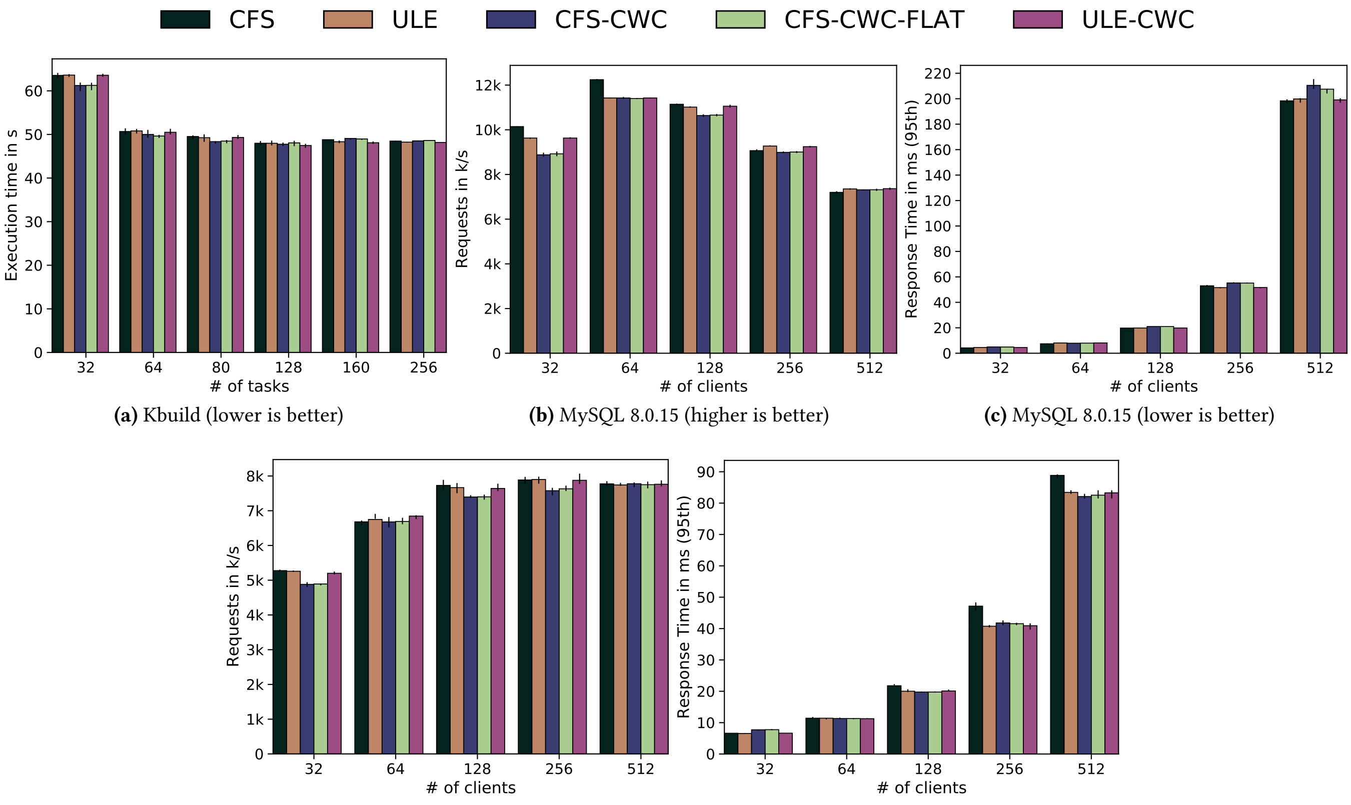
Recent work has also focused on specifications that cover high-level properties of some subsystems of an OS. For instance, file system implementations have been proved resistant to crashes [1, 11, 12, 49], and Frost et al. [23] have formalized file-system dependencies (*e.g.*, a read must be done after a write). Finally, other work has targeted the verification of complex properties in distributed systems [25–27, 55]. In this work, we focus on the challenges of proving properties when reads to shared variables are allowed outside of critical sections on a single machine.

Scheduling. PROSA [10] explored proofs of schedulability analysis for real-time systems. Xu et al. [56] formalized the speed of convergence of various load-balancing algorithms. Their manual proofs could be reused in our system to prove that the implementations of their load balancing algorithms are work conserving.

DSLs. Schüpbach et al. have designed a DSL to abstract the topology of multicore machines [47]. Muller et al. have previously used a DSL to develop schedulers in the context of the Bossa framework [40]. In the case of Bossa, the guarantees of the DSL were used to prove low-level scheduling properties of single-core schedulers, for instance that a core never executes a blocked thread. We built upon this work for the design of Ipanema.

In a preliminary work [32], we introduced abstractions for proving WC (Section 2.2); the proofs require the absence of concurrent scheduling events during load balancing. No implementation was described. This paper adds concurrent scheduling events, and provides a complete solution for programming CWC scheduling policies and proving them thanks to our DSL.

Delaware et al. [17] also exploit the use of a DSL to ensure correctness properties. They provide a framework for developing optimizing compilers for DSLs in which each step of the compiler is implemented as a transformation that is



(a) Kbuild (lower is better)

(b) MySQL 8.0.15 (higher is better)

(c) MySQL 8.0.15 (lower is better)

(d) MongoDB 4.1.8 (higher is better)

(e) MongoDB 4.1.8 (lower is better)

Figure 6. Performance of vanilla CFS, ULE, CFS-CWC, CFS-CWC-FLAT and ULE-CWC

formally verified in Coq. In contrast, we exploit a DSL to restrict the structure of the generated code, making it possible to verify the latter code's algorithmic properties.

7 Conclusion

Writing a scheduler for a multicore system is error-prone. In this paper, we have presented a methodology to write multicore schedulers with provable correctness properties. We have shown how to prove work conservation for a scheduler that reads the instantaneous state of other cores without holding locks, and thus might take decisions based on out-of-date information.

We believe our approach could be leveraged to ease the development of novel scheduling policies, in existing operating systems, or in OS courses. In future work, we want to extend our approach to prove other properties such as thread liveness. Our code and proofs are publicly available at https://gitlab.inria.fr/ipanema-public.

8 Acknowledgements

This work was supported in part by the Swiss National Science Foundation Grants No. 513954 and 514009, and by Oracle donation CR 1930. We thank the anonymous reviewers and our shepherd, Gernot Heiser, for their feedback.

A Simplified BNF grammar of Ipanema

```
entry       ::= scheduler id = {topdecl* body*}
topdecl     ::= constdef | typedef | scheddef | pstatedef | cstatedef | globaldef
constdef    ::= system? const id = exp;
typedef     ::= type id = struct {vardecl+} ; | type id = enum {id+} ;
scheddef    ::= (domain | group | thread) = {vardecl+}
pstatedef   ::= threads = {pstatedecl+} | coredef
cstatedef   ::= cores = {cstatedecl+}
coredef     ::= core = {coredecl+
coredecl    ::= vardecl | pstatedef | steal
globaldef   ::= vardecl | system? type id(param*);
pstatedecl  ::= system?
cstatedecl  ::= (active | sleeping) core id;
              | (active | sleeping) set<core> id;
steal       ::= steal(group id, core id) = {stealgrp}
              | steal((core id))? = {stealthd}
stealgrp    ::= filtergrp selectgrp stealthd until?
              | filtergrp do {selectgrp stealthd stmt} until
stealthd    ::= filter select migrcond
              | filter do {select migrcond stmt} until?
filtergrp   ::= can_steal_group (group id, group id) {exp} => id
selectgrp   ::= select_group() {exp} => id
filter      ::= can_steal_core((core id,)? core id) {exp} => id

select      ::= select_core() {exp} => id
migrcond    ::= steal_thread((group id,)? core id, thread id) stmt until?
until       ::= until (exp)
body        ::= handler(type id) {event*} | interface = {fctdef*} | fctdef*
fctdef      ::= type id (param*) stmt
event       ::= On synchronized? eventid stmt
eventid     ::= schedule | unblock | block | new | terminate | yield | tick
vardecl     ::= system? type id | lazy? type id = exp
stmt        ::= foreach( id in exp order? ) stmt | loc => exp; | ...
exp         ::= self | exp in loc | first(exp order?) | valid(exp)
              | empty(exp) | syscores() | ...
loc         ::= id | state | self | loc.id | first(exp order?)
order       ::= order = {(highest | lowest) id}
```

References

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

[2] Aponte, M., Courtieu, P., Moy, Y., and Sango, M. Maximal and compositional pattern-based loop invariants. In *FM 2012: Formal Methods - 18th International Symposium* (2012), pp. 37–51.

[3] Bailey, D., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing* (Nov. 1991), pp. 158–165.

[4] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *Computer Aided Verification (CAV)* (July 2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, Springer, pp. 171–177. Snowbird, Utah.

[5] Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages* (Wrocław, Poland, August 2011), pp. 53–64. https://hal.inria.fr/hal-00790310.

[6] Bouron, J. [PATCH] Fix bug in which the long term ULE load balancer is executed only once. https://bugs.freebsd.org/bugzilla/show$_{b}$ug.cgi?id=223914, 2017.

[7] Bouron, J., Chevalley, S., Lepers, B., Zwaenepoel, W., Gouicem, R., Lawall, J., Muller, G., and Sopena, J. The battle of the schedulers: FreeBSD ULE vs. Linux CFS. In *USENIX Annual Technical Conference (USENIX ATC)* (2018), pp. 85–96.

[8] Buttazzo, G. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (Third ed.)*. Springer, New York, NY, 2011.

[9] Carver, D., Gouicem, R., Lozi, J.-P., Sopena, J., Lepers, B., Zwaenepoel, W., Palix, N., Lawall, J., and Muller, G. Fork/wait and multicore frequency scaling. In *Workshop on Programming Languages and Operating Systems (PLOS)* (2019), ACM Press.

[10] Cerqueira, F., Stutz, F., and Brandenburg, B. B. PROSA: A case for readable mechanized schedulability analysis. In *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on* (2016), IEEE, pp. 273–284.

[11] Chen, H., Chajed, T., Konradi, A., Wang, S., İleri, A., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Verifying a high-performance crash-safe file system using a tree specification. In *Symposium on Operating Systems Principles (SOSP)* (2017), pp. 270–286.

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

[13] Chen, T., Ananiev, L. I., and Tikhonov, A. V. Keeping kernel performance from regressions. In *Linux Symposium* (2007), vol. 1, pp. 93–102.

[14] Chong, N., and Ishtiaq, S. Reasoning about the ARM weakly consistent memory model. In *Workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008), ACM, pp. 16–19.

[15] Conchon, S., Coquereau, A., Iguernlala, M., and Mebsout, A. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories* (Oxford, United Kingdom, July 2018).

[16] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: a holistic approach to memory placement on NUMA systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), pp. 381–394.

[17] Delaware, B., Pit-Claudel, C., Gross, J., and Chlipala, A. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Principles of Programming languages (POPL)* (2015), pp. 689–700.

[18] Deligiannis, P., Donaldson, A. F., and Rakamaric, Z. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Automated Software Engineering (ASE)* (2015), pp. 166–177.

[19] Dobrescu, M., and Argyraki, K. Software dataplane verification. *Communications of the ACM 58*, 11 (2015), 113–121.

[20] Engler, D., and Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP)* (2003), pp. 237–252.

[21] Erickson, J., Musuvathi, M., Burckhardt, S., and Olynyk, K. Effective data-race detection for the kernel. In *Operating Systems Design and Implementation (OSDI)* (2010), pp. 151–162.

[22] Filliâtre, J.-C., Gondelman, L., and Paskevich, A. The spirit of ghost code. *Formal Methods in System Design 48*, 3 (2016), 152–174.

[23] Frost, C., Mammarella, M., Kohler, E., de los Reyes, A., Hovsepian, S., Matsuoka, A., and Zhang, L. Generalized file system dependencies. In *Symposium on Operating Systems Principles (SOSP)* (2007), pp. 307–320.

[24] Gu, R., Shao, Z., Chen, H., Wu, X. N., Kim, J., Sjöberg, V., and Costanzo, D. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *Operating Systems Design and Implementation (OSDI)* (2016), pp. 653–669.

[25] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. Ironfleet: proving practical distributed systems correct. In *Symposium on Operating Systems Principles (SOSP)* (2015), ACM, pp. 1–17.

[26] Hawblitzel, C., Howell, J., Lorch, J. R., Narayan, A., Parno, B., Zhang, D., and Zill, B. Ironclad apps: End-to-end security via automated full-system verification. In *Operating Systems Design and Implementation (OSDI)* (2014), vol. 14, pp. 165–181.

[27] Hermenier, F., and Henrio, L. Trustable virtual machine scheduling in a cloud. In *Symposium on Cloud Computing (SOCC)* (2017), ACM, pp. 15–26.

[28] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. In *International Symposium on Computer Architecture (ISCA)* (2015), pp. 158–169.

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

[30] Kuncak, V. Developing verified software using Leon. In *NASA Formal Methods (NFM)* (2015), pp. 12–15.

[31] Leino, K. Rustan, M. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010), pp. 348–370.

[32] Lepers, B., Zwaenepoel, W., Lozi, J., Palix, N., Gouicem, R., Sopena, J., Lawall, J., and Muller, G. Towards proving optimistic multicore schedulers. In *Workshop on Hot Topics in Operating Systems (HotOS)* (2017), pp. 18–23.

[33] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming languages (POPL)* (2006), pp. 42–54.

[34] Linux test project. https://linux-test-project.github.io/, 2012.

[35] Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M. F., and Zhang, Z. D^{3}S: debugging deployed distributed systems. In *Networked Systems Design and Implementation (NSDI)* (2008), pp. 423–437.

[36] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. The Linux scheduler: a decade of wasted cores. In *European Conference on Computer Systems (EuroSys)* (2016), ACM, pp. 1:1–1:16.

[37] Mai, H., Pek, E., Xue, H., King, S. T., and Madhusudan, P. Verifying security invariants in ExpressOS. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), pp. 293–304.

[38] Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. Devil: An IDL for hardware programming. In *Operating Systems Design and Implementation (OSDI)* (2000), pp. 17–30.

[39] Muller, G., Consel, C., Marlet, R., Barreto, L. P., Merillon, F., and Reveillere, L. Towards robust OSes for appliances: A new approach based on domain-specific languages. In *ACM SIGOPS European workshop* (2000), ACM, pp. 19–24.

[40] Muller, G., Lawall, J. L., and Duchesne, H. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *High-Assurance Systems Engineering (HASE)* (2005), IEEE, pp. 56–65.

[41] Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. CMC: a pragmatic approach to model checking real code. In *Operating Systems Design and Implementation (OSDI)* (2002), pp. 75–88.

[42] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-button verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP)* (2017), pp. 252–269.

[43] Perl, S. E., and Weihl, W. E. Performance assertion checking. In *Symposium on Operating Systems Principles (SOSP)* (1993), pp. 134–145.

[44] Sahoo, S. K., Criswell, J., Geigle, C., and Adve, V. Using likely invariants for automated software fault localization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013), pp. 139–152.

[45] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS) 15*, 4 (Nov. 1997), 391–411.

[46] Scheduler domains. https://www.kernel.org/doc/html/latest/scheduler/sched-domains.html.

[47] Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., and Isaacs, R. Embracing diversity in the Barrelfish manycore operating system. In *Workshop on Managed Many-Core Systems* (2008), vol. 27.

[48] Shen, K., Zhong, M., and Li, C. I/O system performance debugging using model-driven anomaly characterization. In *File and Storage Technologies (FAST)* (2005), pp. 309–322.

[49] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. Push-button verification of file systems via crash refinement. In *Operating Systems Design and Implementation (OSDI)* (2016), pp. 1–16.

[50] Sites, D. Data center computers: Modern challenges in CPU design, 2015. https://www.youtube.com/watch?v=QBu2Ae8-8LM(56:32).

[51] Sysbench. https://github.com/akopytov/sysbench.

[52] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., and Vogler, R. Static race detection for device drivers: The Goblint approach. In *Automated Software Engineering (ASE)* (2016), IEEE, pp. 391–402.

[53] Waldspurger, C. A., and Weihl, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation (OSDI)* (1994), pp. 1–11.

[54] Wand, M. Continuation-based multiprocessing. In *LISP and Functional Programming* (1980), pp. 19–28.

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

[56] Xu, C., and Lau, F. C. M. *Load balancing in parallel computers: theory and practice*, vol. 381. Springer Science & Business Media, 1996.

[57] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS) 24*, 4 (Nov. 2006), 393–423.