Provable Multicore Schedulers with Ipanema: Application to Work Conservation

Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, Gilles Muller

To cite this version:

HAL Id: hal-02554342
https://hal.inria.fr/hal-02554342
Submitted on 25 Apr 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.


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

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 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
We then use these new definitions to craft a definition of concurrency in scheduling events, or, in the case of load balancing, if a thread is stolen for 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 $c$,

$$CO(c) \equiv O(c) \land \neg U(c)$$

$$CI(c) \equiv I(c) \land \neg B(c) \land \neg E(c)$$

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 $c$ if at the end of the scheduling event:

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

and is global concurrent work conserving if 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 if a thread block or terminate operation on any core in $g$ overlaps with load balancing. Then, for a group $g$ of $k$ cores:

$$CO_k(g) \equiv O_k(g) \land \neg U_k(g)$$

$$CI_k(g) \equiv I_k(g) \land \neg B_k(g) \land \neg E_k(g)$$

and a load balancer is hierarchical concurrent work conserving if 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 run-queue. 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 \( \text{cload} \), 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 \( \text{count} \), and the global variables \( \text{idle_cores} \) and \( \text{active_cores} \), 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 \( \text{cload} \) 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

```c
const int INTERRUPT = 1; const int REGULAR = 2;
const int INTERACTIVE = 4;
thread = { int load=1, prio, slice; core last_core; }
core = {
    threads = {
        shared RUNNING thread current;
        shared READY set thread -> realtime.order = [highest prio];
        shared READY set thread -> timeshare.order = [highest prio];
    }
    system shared int cloud;
}
steal = {
    can_steal_core(core src, core dst) {
        stealable_cores
        do {
            select_core() { first(stealable_cores order = [highest cloud]) } => src
            steal_thread(core dst, thread t) {
                if (src.cloud - dst.cloud >= 2) {
                    if (t.prio == REGULAR) dst.realtime;
                    else t = dst.realtime;
                }
            }
        } until (runnable(dst)!= 0)
    }
}
handler (thread_event e) {
    On tick {
        e.target.slice --;
        if (e.target.slice == 0) {
            update_realtime(e.target);
            if (t.prio == REGULAR) e.target => timeshare;
            else t = target.realtime;
        }
    }
    On yield { ... }
    On block {
        e.target => blocked;
    }
    On unblock {
        thread t = e.target;
        core idlest = choose_wakeup_core(e.target);
        if (update_realtime(t)) t = idlest.realtime;
        else t = idlest.timebase;
    }
    On schedule {
        thread t = first(realtime);
        if (!valid()) t = first(timeshare);
        t = current;
        t.last_core = self;
        t.slice = get_slice(t);
    }
    On new { ... } On detach { ... }
}
handler (core_event e) {
    On synchronized balancing {
        foreach (c in system_cores) order = [lowest cloud] {
            steal_for(c);
        }
    }
    void update_realtime(thread t) {
        int runnable(core c) { return count(c.realtime) + count(c.timeshare) + ...; }
        int get_slice(thread t) { ... }
        core choose_wakeup_core(thread t) {
            // Run interrupt threads on their core
        }
        if (t.prio == INTERRUPT) { return last_core; }
        foreach (g in last_core.d.groups) { // Pick an idle cpu that shares a L2
            if (g.sharing_level & L2_CACHE) {
                foreach (c in g.cores) if (c.cloud == 0) return c; }
        }
        return first(system_cores) order = [lowest cloud]; // Default
    }
}
```

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., cloud 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 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 cloud, 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., cloud 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

---

**Listing 1 A CWC ULE-like scheduler**

```c
const int INTERRUPT = 1; const int REGULAR = 2;
const int INTERACTIVE = 4;
thread = { int load=1, prio, slice; core last_core; }
core = {
    threads = {
        shared RUNNING thread current;
        shared READY set thread -> realtime.order = [highest prio];
        shared READY set thread -> timeshare.order = [highest prio];
    }
    system shared int cloud;
}
steal = {
    can_steal_core(core src, core dst) {
        stealable_cores
        do {
            select_core() { first(stealable_cores order = [highest cloud]) } => src
            steal_thread(core dst, thread t) {
                if (src.cloud - dst.cloud >= 2) {
                    if (t.prio == REGULAR) dst.realtime;
                    else t = dst.realtime;
                }
            }
        } until (runnable(dst)!= 0)
    }
}
handler (thread_event e) {
    On tick {
        e.target.slice --;
        if (e.target.slice == 0) {
            update_realtime(e.target);
            if (t.prio == REGULAR) e.target => timeshare;
            else t = target.realtime;
        }
    }
    On yield { ... }
    On block {
        e.target => blocked;
    }
    On unblock {
        thread t = e.target;
        core idlest = choose_wakeup_core(e.target);
        if (update_realtime(t)) t = idlest.realtime;
        else t = idlest.timebase;
    }
    On schedule {
        thread t = first(realtime);
        if (!valid()) t = first(timeshare);
        t = current;
        t.last_core = self;
        t.slice = get_slice(t);
    }
    On new { ... } On detach { ... }
}
handler (core_event e) {
    On synchronized balancing {
        foreach (c in system_cores) order = [lowest cloud] {
            steal_for(c);
        }
    }
    void update_realtime(thread t) {
        int runnable(core c) { return count(c.realtime) + count(c.timeshare) + ...; }
        int get_slice(thread t) { ... }
        core choose_wakeup_core(thread t) {
            // Run interrupt threads on their core
        }
        if (t.prio == INTERRUPT) { return last_core; }
        foreach (g in last_core.d.groups) { // Pick an idle cpu that shares a L2
            if (g.sharing_level & L2_CACHE) {
                foreach (c in g.cores) if (c.cloud == 0) return c; }
        }
        return first(system_cores) order = [lowest cloud]; // Default
    }
}
```
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.

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

---

**Figure 2. Structure of load balancing**
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.

### Table 1. Expansion ratio of the generated code

<table>
<thead>
<tr>
<th>Policy</th>
<th>DSL LOC</th>
<th>C LOC</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFS-CWC</td>
<td>362</td>
<td>1,526</td>
<td>x4.22</td>
</tr>
<tr>
<td>CFS-CWC-FLAT</td>
<td>222</td>
<td>1,267</td>
<td>x5.71</td>
</tr>
<tr>
<td>ULE-CWC</td>
<td>243</td>
<td>1,404</td>
<td>x5.78</td>
</tr>
<tr>
<td>ULE</td>
<td>260</td>
<td>1,382</td>
<td>x5.32</td>
</tr>
</tbody>
</table>

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 \( \text{synch}(c) \) such that \( \text{synch}(c) \) is satisfied if and only if the former case holds. In the latter case, when \( \neg\text{synch}(c) \), 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 \( \neg\text{synch}(c) \) 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 \(|c|_\text{old}\) 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, \( \text{synch}(c) \) holds. After removing the thread from the runqueue \( \neg\text{synch}(c), B(c), \text{and } |c| < |c|_\text{old} \) hold. After updating cload and the thread-count variable, \( \text{synch}(c) \) should hold again, but \(|c| < |c|\). As \( B(c) \) is still true, this situation satisfies \( \text{synch}(c) \). Finally, changing the state of the blocking thread to indicate that it has blocked has no effect on \( \text{synch}(c), B(c), \text{and } \) 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.
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.
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 fast-paths 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 an 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...
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 95th 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
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 = (topdef body)*
topdef ::= constdef | typedef | scheddef | onetaskdef | c戡edef | globaldef
constdef ::= system | const id = exp,
typedef ::= type id = struct {vardef*} ; | type id = enum {id*} ;
scheddef ::= (domain | group | thread) = {vardef*}
onetaskdef ::= threads = {vardef*} | coredef
c戡edef ::= cores = {vardef*}
coredef ::= core = {vardef*}
coredef ::= vardef | ptaskdef | steal
globaldef ::= vardef | system | type id=param*;
ptaskdef ::= system
c.double ::= (active | sleeping) core id;
steal ::= steal(group id, core id) = (stealgp);
stealgp ::= steal(group id, core id) = steal(g compromises i, id)
until, core = (stealh);
stealh ::= filter select migroand
until, filter do (select migroand stmt) until
filtergp ::= can_steal_group(group id, group id) (exp) => id
select_group gp (exp) => id
filter ::= can_steal_core(core id) (exp) => id
select ::= select_core gp (exp) => id
migroand ::= steal_thread(group id, core id, thread id) stmt until
until, stmt, until
body ::= handler(type id (event*) | interface = {fctdef*} | fctdef
fctdef ::= type id (param*) stmt
event ::= On synchronized event stmt
event ::= schedule | unblock | block | new | terminate | yield | tick
vardef ::= 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)
loc ::= id | self | self | loc.id | first(exp order)
order ::= order = (highest | lowest) id
```

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


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


