Towards Proving Optimistic Multicore Schedulers

Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix, Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller

Towards Proving Optimistic Multicore Schedulers

Baptiste Lepers, Willy Zwaenepoel
EPFL
first.last@epfl.ch

Jean-Pierre Lozi
Université Nice Sophia-Antipolis
jplozi@unice.fr

Nicolas Palix
Université Grenoble Alpes
nicolas.palix@univ-grenoble-alpes.fr

Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller
Sorbonne Universités, Inria, LIP6
first.last@lip6.fr

Abstract

Operating systems have been shown to waste machine resources by leaving cores idle while work is ready to be scheduled. This results in suboptimal performance for user applications, and wasted power.

Recent progress in formal verification methods have led to operating systems being proven safe, but operating systems have yet to be proven free of performance bottlenecks. In this paper we instigate the first effort in proving performance properties of operating systems by designing a multicore scheduler that is proven to be work-conserving.

1. Introduction

Operating system schedulers are a central part of the resource management in multicore machines, yet they often fail to fully leverage available computing power. The default Linux scheduler (CFS) has been shown to leave cores idle while threads are waiting in runqueues (Lozi et al. 2016). This is not an intended behavior: CFS was designed to be work-conserving, meaning it tries to keep cores busy when work is ready for execution. In practice, leaving cores idle results in suboptimal performance and wasted power: we have observed many-fold performance degradation in the case of scientific applications, and up to 25% decrease in throughput for realistic database workloads.

Recent efforts have allowed operating systems to be formally specified and proven correct according to their specification (Gu et al. 2016; Klein et al. 2009; Chen et al. 2015; Amani et al. 2016). Yet, performance properties are largely missing from these specifications: no general-purpose operating system is proven to be work-conserving, fair between threads, or reactive (i.e., to have a bound on the delay to schedule ready threads). Particularly challenging is that, unlike safety properties that either always hold or are broken, performance properties are more loosely defined. For instance, it is perfectly acceptable for a core to become temporarily idle (e.g., after an application exits). Temporary idleness must therefore not be treated as a violation of the work-conserving property.

Another challenge in proving performance properties of multicore schedulers comes from uncoordinated concurrent operations. In CFS, for instance, two cores might decide to steal work from a third core at the same time. In this situation, one of the two cores might fail to steal work because the third core no longer has enough threads to give. Introducing locks to avoid failures is not a desirable option: locking the runqueue of the third core prevents that core from scheduling work and may impact the whole system performance. We think that it is desirable to allow cores to look at the other cores' states and take *optimistic* decisions based on these observations, without locks. Occasional failure to steal threads based on these optimistic decisions should not be considered to be a work-conservation violation. The proper correctness criterion is rather that over time every idle core will manage to steal work from overloaded cores. Handling optimistic decisions and failures is a significant difference with previous attempts at certifying concurrent operating systems, which forced all operations to succeed.

In this paper, we investigate a first effort at better defining and proving performance properties of an optimistic multicore scheduler. More precisely, we present the design and correctness proof of the load balancing part of a multicore scheduler with respect to work conservation. We target schedulers that could be used in practice, which implies that the scheduler should scale to a large number of cores, and implement the complex scheduling heuristics used on modern hardware such as NUMA-aware thread placement.

The backbone of our solution is a set of abstractions that express the fundamental building blocks of a scheduler. These abstractions are exposed to kernel developers via a domain-specific language (DSL), which is then compiled to C code that can be integrated as a scheduling class into the Linux kernel, and to Scala code that is verified by the

HotOS '17, May 08-10, 2017, Whistler, BC, Canada
Copyright © 2017 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-5068-6/17/05... $15.00
DOI: http://dx.doi.org/10.1145/3102980.3102984

Leon toolkit (Leon System for Verification 2010). The design of our abstractions was driven by three constraints: (i) providing sufficient expressiveness, (ii) enabling verification of scheduling policy behavior, and (iii) incurring low overhead. One main technique was to design abstractions that would decompose the scheduler into multiple operations that can be verified in isolation, thus simplifying the proving effort.

This paper makes the following contributions: (i) We formalize work-conserving properties for multicore schedulers in the presence of concurrent, and possibly conflicting, work-stealing attempts. (ii) We present abstractions that allow one to design an optimistic multicore scheduler and allow reducing the proving effort. (iii) We investigate the design of proofs of work-conserving properties in a concurrent environment.

2. Related Work

Kernel correctness Avoiding race conditions and deadlocks in operating systems has been studied in previous works (Engler and Ashcraft 2003; Savage et al. 1997; Erickson et al. 2010; Vojdani et al. 2016; Deligiannis et al. 2015). These systems are restricted to detecting simple ordering properties: shared variables are checked to be accessed only in critical sections, and a strict ordering of locks is checked to prevent deadlocks. Recently, a full multicore micro-kernel has been certified functionally correct (Gu et al. 2016). We follow these works to prove the correctness of critical sections in our code, but allow a more relaxed concurrency model that takes into account *optimistic decisions*.

D3S (Liu et al. 2008), Likely Invariants (Sahoo et al. 2013), and ExpressOS (Mai et al. 2013) ensure that no overflow or underflow happens in the kernel. These works allow expressing invariants that a system must follow, but are currently limited to verifying low level properties of systems (ordering of locks, boundaries of variables).

Another approach to improve the correctness of systems is to formally specify their intended behavior and verify that their implementation matches the specification. SeL4 was the first attempt at specifying a full micro-kernel (Klein et al. 2009). Frost et al. (Frost et al. 2007) present abstractions to make file-system dependencies explicit (e.g., a read must be done after a write). Formal specifications of file systems (Chen et al. 2015; Amani et al. 2016) are a recent breakthrough in the formal specification of operating systems. Like these works, we aim to prove high level properties of systems, with the added difficulty of handling concurrent operations.

Besides formal proofs, model checking has also been used to check kernel properties. CMC (Musuvathi et al. 2002) finds implementation bugs that can lead to crashes. Yang et al. (Yang et al. 2006) found deadlocks in file systems. A scheduler is particularly challenging to model check due to the large number of possible workloads and concurrent operations.

Detecting performance issues Performance regression testing has been done for a long time in operating systems. The Linux Kernel Performance project (Chen et al. 2007) was initiated in 2005 to test for performance regressions. The project uses standard workloads (scientific applications, databases) and compares their performance on every major kernel version. Applications are tested in isolation to avoid performance fluctuations due to non deterministic scheduling decisions in multi-application workloads. While these tests might reveal flagrant flaws in the scheduler design, they are unlikely to find complex bugs that happen when multiple applications are scheduled together.

Work has been done to check that the time taken by various kernel functions is within “reasonable” bounds (Perl and Weihl 1993; Shen et al. 2005). These works could be used to detect performance bugs due to inefficient code paths in the kernel. While these works are an essential first step in understanding abnormal kernel behaviors, they cannot be used to detect high-level issues, such as cores staying idle while threads are ready to be scheduled. The same principles have been applied in distributed systems with the same limitations (Aguilera et al. 2003; Chanda et al. 2007; Barham et al. 2004; Attariyan et al. 2012).

Formalizing OS behavior Xu et al. (Xu and Lau 1996) have studied the speed of convergence of various load balancing algorithms. We plan to build upon this work to prove latency limits on the work-conserving property of our scheduler. DSLs have been proposed to make OSes more robust (Muller et al. 2000). They have been used to enable manipulating devices safely (Schüpbach et al. 2011; Ryzhyk et al. 2009; Mérillon et al. 2000) and to write schedulers (Muller et al. 2005). For Bossa (Muller et al. 2005), the authors proved low-level scheduling properties, e.g., a core never executes a thread that is blocked. We build upon this work to design abstractions for multi-core schedulers.

3. Defining Work Conservation for an Optimistic Multicore Scheduler

Informally, a work-conserving scheduler guarantees that the CPU resource will not be wasted indefinitely. In this section, we provide a formal definition of a work-conserving multicore scheduler. First, we define a model of an optimistic scheduler and explain how threads are assigned to cores in that model. Second, we define work conservation in that context.

3.1 Scheduler model

A scheduler is defined with reference to, for each core of the machine, the *current* thread, if any, that is running on that core, and a *runqueue* containing threads waiting to be scheduled. Cores can only schedule threads that are waiting in their own runqueue. This model is used by most general-purpose operating systems (Linux, FreeBSD, Solaris, Windows, etc.) since having a runqueue per core avoids contention issues (Multiple run-queues for BFS 2012).

The downside of having each core schedule work from its own runqueue is that threads may need to be migrated between runqueues to keep cores busy. We define an *idle* core as a core that has no current thread and no thread in its runqueue. We define an *overloaded* core as a core that has two or more threads, including the current thread.

Periodically, the amount of work waiting in the cores' runqueues is balanced. We call each occurrence of this periodic process a *load balancing round*. The simplest load balancers try to balance the number of threads in runqueues, but realistic schedulers usually adopt more complex load balancing strategies. For instance, CFS considers some threads more important (different niceness), and gives them a higher share of CPU resources. In this context, the load balancer tries to balance the number of threads weighted by their importance. We make no assumption on the criteria used to define how the load should be balanced. We only aim to verify that the criteria do not lead to wasted CPU resources.

The operations of a load balancing round might be performed simultaneously on multiple cores, both idle and non-idle. For instance, in CFS, load balancing operations are performed simultaneously on *all* cores every 4ms. In a load-balancing round, each core independently chooses another core from which to steal, and then performs the stealing operation. When load balancing operations happen simultaneously on multiple cores, some of them may conflict. For example, if two cores simultaneously try to steal a thread from a third core that has only one thread waiting in its runqueue, then one of the two cores will fail to steal a thread.

Our scheduler model integrates potential failures of the load balancing round operations. In a load balancing round, a core (i) tries to find a core from which to steal (selection phase), and then (ii) tries to steal from the core it has chosen (stealing phase). As the selection phase is lock-less, a core is not guaranteed to be able to steal everything that it has selected. In our model, the selection phase may not modify runqueues, and all accesses to shared variables must be read-only. The stealing phase must be done atomically for correctness (*i.e.*, no two cores should be able to steal the same thread).

In order to simplify the proofs, we break the load balancing down into three steps, as shown in Figure 1. First, a core uses a filter function to create a list of other cores that it can steal from. Second, it chooses a core from this list (if any). Third, the core steals thread(s) from the chosen core. Having a filter and then a choice is leveraged in the proofs. In fact, the choice step can mostly be ignored in the work-conserving proof: if the filter defined in the first step is correct then, no matter which core is chosen in the second step, threads can be stolen from the chosen core (assuming no other core is stealing threads from that core concurrently). The exact choice of the core does not matter for the correctness proof. This provides a notable simplification of the proving effort as the counterpart of the choice step in legacy OSes usually contains all the complex heuristics used to perform smart thread placement (e.g., giving priority to some core to improve cache locality, NUMA-aware decisions, etc.).

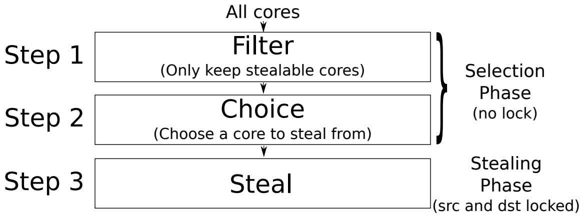

Figure 1: The three steps performed by a core during a load balancing round. First, the core decides from which cores threads can be stolen. Second, the core decides from which core to steal amongst the selected cores (if any). Third, the core performs the stealing operation. During the third step, the runqueue of the core that initiated the stealing operation and the runqueue of the targeted core are both locked.

3.2 Work conservation

We now formalize the work-conserving property of a scheduler. An ideal work-conserving scheduler would make sure that at any given instant no thread is waiting in a runqueue if a core is idle. In practice, idleness is expected and inevitable: because stealing attempts can fail, a core may remain idle even though it tried to steal work from an overloaded core. The correctness criterion presented below instead requires that this situation does not persist forever.

Definition:

Let $\{1, ..., n\}$ be a list of cores and c_i be the state of core i.

Let s be a scheduler.

Let $s.lb(c_1, ..., c_n)$ be the execution of a load balancing round, producing a new set of core states $c'_1, ..., c'_n$.

Then, s is work-conserving

$$\iff$$

$$\left(\begin{array}{c} \forall c_1, ..., c_n, \exists N, c'_1, ..., c'_n \mid \\ s.lb^N(c_1, ..., c_n) = (c'_1, ..., c'_n) \wedge \\ \forall i, j \in \{1, ..., n\}, idle(c'_i) \Rightarrow ! overloaded(c'_j) \end{array} \right)$$

A scheduler is work-conserving iff there exists an integer N such that after N load balancing rounds no core is idle while a core is overloaded.

4. Load Balancing Abstractions and Proofs

In this section, we present a simple load balancer and a sketch of the proof that the load balancer is work-conserving. Our proofs are under the assumption that that no thread enters or leaves the runqueues (e.g., no thread is created or terminated). This is an acceptable assumption since changes in the runqueues could perpetually prevent the load balancing rounds from stealing threads (one could imagine that threads always terminate before being stolen). We begin with a proof that a simple load balancer is work-conserving in the case of no concurrency. Then we show that concurrency significantly complicates the proving effort.

4.1 Simple load balancer

Listing 1 presents the Leon code of a simple load balancer that tries to balance the number of threads between cores. The code of the load balancer follows the abstractions presented in Section 3.1. As indicated by the filter function, (step 1 of the abstraction, line 6 of Listing 1), a core A only steals tasks from a core B if A has at least two fewer threads than B. If this is the case, then, as indicated by the stealing function (step 3, line 11) A steals one task from B. We do not describe the choice of the core performed during the second step (line 9), as it has no influence on the work-conserving proof: it suffices to ensure that the selected core is in the list of stealable cores (line 10).

4.2 Simple context - No concurrency

We now explain how to leverage the three step structure in work-conserving proofs in a simplified setup. In this setup, in each load-balancing round the load-balancing operations do not overlap (i.e., core 0 first does all three load-balancing steps in isolation, then core 1 does all three steps, etc.). We also assume that no thread is added or removed from the runqueues outside of load balancing operations. These constraints simplify proofs, because load balancing operations cannot fail. Still, these proofs are important to show that the filter and `stealCore` functions are sound, meaning that (i) an idle core wants to steal from overloaded cores (and only them), and (ii) during the stealing phase (third step), the idle core actually steals threads from an overloaded core, and does not steal too much from that overloaded core (i.e., in our load-balancing algorithm, the overloaded core should not end up idle after the load-balancing operation).

Listing 2 shows a Leon proof that, when work stealing is initiated from an idle core, and the machine has at least one overloaded core, then the idle core must not filter out all cores of the machine. Using Leon, Lemma1 is expressed as a function that must hold true (`.holds`) for all values of `thief` and `cores` that match the requirements on line 8. Since we only care about idle cores being able to steal from other cores, line 8 specifies that the lemma only applies when `thief` is idle (recall that non-idle cores also perform load balancing operations in our model). The proof is simple: we want to check that a core with 0 threads tries to steal from a core with 2 or more threads. In our non-concurrent setting, Leon can automatically prove that this property holds, even for relatively complex filter functions. For instance, we have found that the proof is still automatically verified for a load balancer that tries to balance the number of threads weighted by their importance. In a sequential setting, this proof is sufficient to ensure that, after one round of load balancing operations on an idle core, if the system had an overloaded core, then the idle core has successfully stolen a thread. Proving that stealing threads cannot make the affected cores idle is then sufficient to prove that the scheduler is work-conserving.

The apparent simplicity of the proof is a direct result of using the abstractions. Most of the complex load-balancing logic (step 2 in Figure 1) is ignored: we only need to prove a property of the filter (step 1) and then ensure that the second step returns a core contained in the result of the first step.

4.3 More realistic context - Adding concurrency and failures

The properties of Section 4.2 are not sufficient when multiple cores perform load balancing operations simultaneously, because in this case work-stealing attempts can fail. Failed attempts imply that idle cores might remain idle at the end of a round. In this section, we give an intuition on how we prove work conservation in the presence of failures.

We begin with an example that illustrates the challenges in designing a work-conserving load-balancing algorithm for a concurrent setting. Suppose that we replace the filter function in Listing 1 by the following, which allows any core to steal from any other overloaded core:

```
def canSteal(stealee) = { stealee.load() >= 2 }
```

This filter makes our algorithm incorrect in the presence of failures. For example, consider a three-core system where core 0 is idle, core 1 has 1 thread and core 2 has 2 threads. During the first load-balancing round, cores 0 and 1 both want to steal a thread from core 2; core 1 succeeds and core 0 fails. During the second round, the same situation could happen, inverting cores 1 and 2, which brings the machine back to the initial state. Core 0 might fail to steal threads forever. A correct load-balancing algorithm must prevent this sort of infinite ping-pong of threads between non-idle cores.

Thus, to show work conservation, we have to show that after a finite number of load-balancing rounds either all of the idle cores have successfully stolen tasks, or that there no longer exist any overloaded cores. Here, we still assume that no thread is added or removed from runqueues outside of load balancing operations. In that context, to prove work conservation we need to prove two properties: first, if a work-stealing attempt fails, it is because another work-stealing attempt performed by another core succeeded, and second, the number of successful work stealing attempts is bounded. Combining the two properties we can prove that the number of failed work-stealing attempts is bounded.

The proof effort is again simplified by the use of the abstractions defined in Section 3.1. For instance the first proof can rely on the fact that failed work-stealing attempts only happen when a core that was marked as stealable during the selection phase is no longer stealable during the stealing phase; this means that the filter implemented in the first step

Listing 1 A simple load balancer that follows the 3 steps of Figure 1.

```
1  case class Core(id: BigInt, current: Option[Task], ready: List[Task], ...) {
2    def load():BigInt = {     # User-defined code
3       self.ready.size + self.current.size
4    }
5    def canSteal(stealee: Core):Boolean = {
6       stealee.load() - self.load() >= 2  # Step 1, user-defined filter
7    }
8    def selectCore(cores: List[Core]): Core = {
9      [...] # Step 2, user-defined code to choose a core from the list
10   } ensuring(res => cores.contains(res))
11   def stealCore(stealee:Core):List[Core] = {
12     if(self.canSteal(stealee)) # Check that the filter of step 1 still holds
13        self.stealOneThread(stealee) # Step 3, user-defined code to steal threads from stealee
14   }
15 }
```

Listing 2 Lemma1: an idle core wants to steal an overloaded core

```
def isOverloaded(core:Core): Boolean = {
   if(core.current.size == 1) core.ready.size >= 1
   else core.ready.size >= 2
}
def Lemma1(thief: Core, cores: List[Core]): Boolean = {
   @require(thief.ready.size == 0 && !thief.current.isDefined); # thief is idle
   ( cores.exists(c => isOverloaded(c)) ==> cores.exists(c => thief.canSteal(c)) ) &&
   ( cores.forall(c => thief.canSteal(c) ==> isOverloaded(c)) )
}.holds;
```

switched from true to false (line 6 of Listing 1). It then suffices to find which lines of code in the load balancer can change the value of the filter – since the selection phase is not allowed to modify the runqueues, the only lines of code that modify the state of the runqueues are in the `stealCore` function that migrates threads; the proof is thus easy to derive based on the results obtained in the previous section.

For the second proof, we show that the absolute "load difference" between cores, computed as follows, decreases with every successful stealing attempt:

$$d(c_1, ..., c_n) = \sum_{i=1}^{n} \sum_{j=1}^{n} |c_i.load - c_j.load|$$

If d always decreases when a core steals threads then, because $d \geq 0$, the number of successful work-stealing operations is bounded. Intuitively this property holds if the `stealCore` function reduces the absolute load difference between the core that initiated the work-stealing operation and the core from which threads are stolen. Combined with the first proof, this implies that the number of load balancing rounds during which cores want to steal from each other is bounded, and thus that the number of failures is bounded.

5. Conclusion and Remaining Challenges

Currently multicore operating systems are developed without performance guarantees. The main goal of this work is to provide abstractions that are expressive enough to write complex load balancing policies while still allowing these policies to be broken down into simple building blocks for which performance properties can be proven with minimal effort. Particularly challenging is the need to be able to deal with concurrency, without incurring lock overhead or situations that can fail indefinitely.

The abstractions presented in this paper make some progress in that direction. For instance, it is possible to implement cache-aware or NUMA-aware thread placements in the second step of the load balancing without adding any complexity to the proofs. However, we have yet to show that the presented abstractions are sufficient for complex load balancing strategies, such as that of Linux. We aim to extend these abstractions to include hierarchical load balancing, for instance to allow balancing load between groups of cores, and then inside groups, instead of balancing load directly between individual cores.

The presented abstractions relax the concurrency requirements of traditional certified concurrent systems, by allowing shared variables to be accessed concurrently outside of critical sections. We believe that this is an essential step in designing systems that scale to a large number of cores. However it also significantly increases the proving effort: we have to prove that all operations will eventually complete despite the presence of failures. To the best of our knowledge, such proofs have never been done on large-scale systems.

References

M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03*, pages 74–89, 2003.

S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. CoGENT: Verifying high-assurance file system implementations. In *ASPLOS*, pages 175–188, 2016.

M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, pages 307–320, 2012.

P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, pages 259–272, 2004.

A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *EuroSys'07*, pages 17–30, 2007.

H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37, 2015.

T. Chen, L. I. Ananiev, and A. V. Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *Automated Software Engineering (ASE'15)*, pages 166–177. IEEE, 2015.

D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI'10*, pages 151–162, 2010.

C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *SOSP '07*, pages 307–320, 2007.

R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *OSDI*, pages 653–669, 2016.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.

Leon System for Verification, Aug. 2010. `https://leon.epfl.ch/`.

X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: debugging deployed distributed systems. In *NSDI*, pages 423–437, 2008.

J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: a decade of wasted cores. In *EuroSys'16*, pages 1:1–1:16. ACM, 2016.

H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *ASPLOS*, pages 293–304, 2013.

F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *OSDI'00*, 2000.

G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Merillon, and L. Reveillere. Towards robust OSes for appliances: A new approach based on domain-specific languages. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 19–24. ACM, 2000.

G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*, pages 56–65. IEEE, 2005.

Multiple run-queues for BFS , Dec. 2012. `https://lwn.net/Articles/529280/`.

M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. In *OSDI'02*, pages 75–88, 2002.

S. E. Perl and W. E. Weihl. Performance assertion checking. In *SOSP '93*, pages 134–145, New York, NY, USA, 1993.

L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *EuroSys*, pages 275–288. ACM, 2009.

S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *ASPLOS '13*, pages 139–152, 2013.

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

A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter. A declarative language approach to device configuration. In *ASPLOS'11*, pages 119–132. ACM, 2011.

K. Shen, M. Zhong, and C. Li. I/O system performance debugging using model-driven anomaly characterization. In *FAST'05*, pages 309–322, 2005.

V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: The goblint approach. In *Automated Software Engineering (ASE'16)*, pages 391–402. IEEE, 2016.

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

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