The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS

Justinien Bouron, Sébastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, Julien Sopena

The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS

Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, and Willy Zwaenepoel, *EPFL;*
Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena,
Sorbonne University/Inria/LIP6

https://www.usenix.org/conference/atc18/presentation/bouron

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS

Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel
EPFL

Redha Gouicem, Julia Lawall, Gilles Muller, Julien Sopena
Sorbonne University, Inria, LIP6

Abstract

This paper analyzes the impact on application performance of the design and implementation choices made in two widely used open-source schedulers: ULE, the default FreeBSD scheduler, and CFS, the default Linux scheduler. We compare ULE and CFS in otherwise identical circumstances. We have ported ULE to Linux, and use it to schedule all threads that are normally scheduled by CFS. We compare the performance of a large suite of applications on the modified kernel running ULE and on the standard Linux kernel running CFS. The observed performance differences are solely the result of scheduling decisions, and do not reflect differences in other subsystems between FreeBSD and Linux.

There is no overall winner. On many workloads the two schedulers perform similarly, but for some workloads there are significant and even surprising differences. ULE may cause starvation, even when executing a single application with identical threads, but this starvation may actually lead to better application performance for some workloads. The more complex load balancing mechanism of CFS reacts more quickly to workload changes, but ULE achieves better load balance in the long run.

1 Introduction

Operating system kernel schedulers are responsible for maintaining high utilization of hardware resources (CPU cores, memory, I/O devices) while providing fast response time to latency-sensitive applications. They have to react to workload changes, and handle large numbers of cores and threads with minimal overhead [12]. This paper provides a comparison between the default schedulers of two of the most widely deployed open-source operating systems: the Completely Fair Scheduler (CFS) used in Linux, and the ULE scheduler used in FreeBSD. Our goal is *not* to declare an overall winner. In fact, we find that for some workloads ULE is better and for others CFS is better. Instead, our goal is to illustrate how differences in the design and the implementation of the two schedulers are reflected in application performance under different workloads.

ULE and CFS are both designed to schedule large numbers of threads on large multicore machines. Scalability considerations have led both schedulers to adopt per-core runqueues. On a context switch, a core accesses only its local runqueue to find the next thread to run. Periodically and at select times, e.g., when a thread wakes up, both ULE and CFS perform load balancing, i.e., they try to balance the amount of work waiting in the runqueues of different cores.

ULE and CFS, however, differ greatly in their design and implementation choices. FreeBSD ULE is a simple scheduler (2,950 lines of code in FreeBSD 11.1), while Linux CFS is much more complex (17,900 lines of code in the latest LTS Linux kernel, Linux 4.9). FreeBSD runqueues are FIFO. For load balancing, FreeBSD strives to even out the number of threads per core. In Linux, a core decides which thread to run next based on prior execution time, priority, and perceived cache behavior of the threads in its runqueue. Instead of evening out the number of threads between cores, Linux strives to even out the average amount of pending work.

The major challenge in comparing ULE and CFS is that application performance depends not only on the scheduler, but also on other OS subsystems, such as networking, file systems and memory management, which also differ between FreeBSD and Linux. To isolate the effect of differences between CFS and ULE, we have ported ULE to Linux, and we use it as the default scheduler to run all threads on the machine (including kernel threads that are normally scheduled by CFS). Then, we compare application performance between this modified Linux with ULE and the default Linux kernel with CFS.

We first examine the impact of the per-core scheduling decisions made by ULE and CFS, by running applica-

tions and combinations of applications on a single core, We then run the applications on all cores of the machine, and study the impact of the load balancer. We use 37 applications ranging from scientific HPC applications to databases. While offering similar performance in many circumstances, CFS and ULE occasionally behave very differently, even on simple workloads consisting of one application running one thread per core.

This paper makes the following contributions:

- We provide a complete port of FreeBSD's ULE scheduler in Linux and release it as open source [21]. Our implementation contains all the features and heuristics used in the FreeBSD 11.1 version.
- We compare the application performance under the ULE and CFS schedulers in an otherwise identical environment.
- Unlike CFS, ULE may starve threads that it deems non-interactive for an unbounded amount of time. Surprisingly, starvation may also occur when the system executes only a single application consisting of identical threads. Even more surprising, this behavior actually proves beneficial for some workloads (e.g., a database workload).
- CFS converges faster towards a balanced load, but ULE achieves a better load balance in the long run.
- The heuristics used by CFS to avoid migrating threads can hurt performance in HPC workloads that only use one thread per core, because CFS sometimes places two threads on the same core, while ULE always places one thread on each core.

The outline of the rest of this paper is as follows. Section 2 presents the CFS and ULE schedulers. Section 3 describes our port of ULE to Linux and the main differences between the native ULE implementation and our port. Sections 4 presents the machines and the workloads used in our experiments. Section 5 analyzes the impact of per-core scheduling in CFS and ULE. Section 6 analyzes the load balancer of CFS and ULE. Section 7 presents related work and Section 8 concludes.

2 Overview of CFS and ULE

2.1 Linux CFS

Per-core scheduling: Linux's CFS implements a weighted fair queueing algorithm: it evenly divides CPU cycles between threads weighted by their priority (represented by their niceness, high niceness meaning low priority and vice versa) [18]. To that end, CFS orders threads by a multi-factor value called *vruntime*, representing the amount of CPU time a thread has already used divided by the thread's priority. Threads with the same priority and same vruntime have executed the same amount of time, meaning that core resources have been shared fairly between them. To ensure that the vruntime of all threads progresses fairly, when the current running thread is preempted, CFS schedules the thread with the lowest vruntime to run next.

Since Linux 2.6.38 the notion of fairness in CFS has evolved from fairness between threads to fairness between applications. Before Linux 2.6.38 every thread was considered as an independent entity and got the same share of resources as other threads in the system. This meant that an application that used many threads got a larger share of resources than single-threaded applications. In more recent kernel versions, threads of the same application are grouped into a structure called a *cgroup*. A cgroup has a vruntime that corresponds to the sum of the vruntimes of all of its threads. CFS then applies its algorithm on cgroups, ensuring fairness between groups of threads. When a cgroup is chosen to be scheduled, its thread with the lowest vruntime is executed, ensuring fairness within a cgroup. Cgroups can also be nested. For instance, systemd automatically configures cgroups to ensure fairness between different users, and then fairness between the applications of a given user.

CFS avoids thread starvation by scheduling all threads within a given time period. For a core executing fewer than 8 threads the default time period is 48ms. When a core executes more than 8 threads, the time period grows with the number of threads and is equal to $6 * number\ of\ threads$ ms; the 6ms value was chosen to avoid preempting threads too frequently. Threads with a higher priority get a higher share of the time period. Since CFS schedules the thread with the lowest vruntime, CFS needs to prevent a thread from having a vruntime much lower than the vruntimes of the other threads waiting to be scheduled. If that were to happen, the thread with the low vruntime could run for a long time, starving the other threads. In practice, CFS ensures that the vruntime difference between any two threads is less than the preemption period (6ms). It does so at two key points: (i) when a thread is created, the thread starts with a vruntime equal to the maximum vruntime of the threads waiting in the runqueue, and (ii) when a thread wakes up after sleeping, its vruntime is updated to be at least equal to the minimum vruntime of the threads waiting to be scheduled. Using the minimum vruntime also ensures that threads that sleep a lot are scheduled first, a desirable strategy on desktop systems, because it minimizes the latency of interactive applications. Most interactive applications sleep most of the time, waiting for user input, and are immediately scheduled as soon as the user

interacts with them.

CFS also uses heuristics to improve cache usage. For instance, when a thread wakes up, it checks the difference between its vruntime and the vruntime of the currently running thread. If the difference is not significant (less than 1ms), the current running thread is not preempted – CFS sacrifices latency to avoid frequent thread preemption, which may negatively impact caches.

Load balancing: In a multicore setting, Linux's CFS evens out the amount of *work* on all cores of the machine. This is different from evening out the number of threads. For instance, if a user runs 1 CPU-intensive thread and 10 threads that mostly sleep, CFS might schedule the 10 mostly sleeping threads on a single core.

To balance the amount of work, CFS uses a *load* metric for threads and cores. The load of a thread corresponds to the average CPU utilization of a thread: a thread that never sleeps has a higher load than one that sleeps a lot. Like the vruntime, the load of a thread is weighted by the thread's priority. The load of a core is the sum of the loads of the threads that are runnable on that core. CFS tries to even out the load of cores.

CFS takes into account the loads of cores when creating or waking up threads. The scheduler first decides which cores are suitable to host the thread. This decision involves many heuristics, such as the frequency at which the thread that initiated the wakeup wakes up threads. For instance, if CFS detects a 1-to-many producer-consumer pattern, then it spreads out the consumer threads as much as possible on the machine, and most cores of the machine are considered suitable to host woken up threads. In a 1-to-1 communication pattern, CFS restricts the list of suitable cores to cores sharing a cache with the thread that initiated the wakeup. Then, among all suitable cores, CFS chooses the core with the lowest load on which to wake up or create the thread.

Load balancing also happens periodically. Every 4ms every core tries to steal work from other cores. This load balancing takes into account the topology of the machine: cores try to steal work more frequently from cores that are "close" to them (e.g., sharing a cache) than from cores that are "remote" (e.g., on a remote NUMA node). When a core decides to steal work from another core, it tries to even out the load between the two cores by stealing as many as 32 threads from the other core. Cores also immediately call the periodic load balancer when they become idle.

On large NUMA machines, CFS does not compare the load of all cores against each other, but instead balances the load in a hierarchical way. For instance, on a machine with 2 NUMA nodes, CFS balances the load of cores inside the NUMA nodes, and then compares the load of the NUMA nodes (defined as the average load of their cores) to decide whether or not to balance the load between nodes. If the load difference between the nodes is small (less than 25% in practice), then no load balancing is performed. The greater the distance between two cores (or groups of cores), the higher the imbalance has to be for CFS to balance the load.

2.2 FreeBSD ULE

Per-core scheduling: ULE uses two runqueues to schedule threads: one runqueue contains *interactive* threads, and the other contains *batch* threads. A third runqueue called *idle* is used when a core is idle. This runqueue only contains the idle task.

The goal of having two runqueues is to give priority to interactive threads. Batch processes usually execute without user interaction, and thus scheduling latency is less important. ULE keeps track of the *interactivity* of a thread using an *interactivity penalty* metric between 0 and 100. This metric is defined as a function of the time r a thread has spent running and the time s a thread has spent voluntarily sleeping (not including the time spent waiting for the CPU), and is computed as follows:

$$scaling\ factor = m = 50$$

$$penalty(r,s) = \begin{cases} \frac{m}{\frac{s}{r}} & s > r \\ \frac{m}{\frac{r}{s}} + m & otherwise \end{cases}$$

A penalty in the lower half of the range (≤ 50) means that a thread has spent more time voluntarily sleeping than running, while a penalty above means the opposite.

The amount of history kept for the sleep and running times is (by default) limited to the last 5 seconds of the thread's lifetime. On the one hand, having a large amount of history would lengthen the time required to detect batch threads. On the other hand, too little history would induce noise in the classification [15].

To classify threads, ULE first computes a *score* defined as *interactivity penalty* + *niceness*. A thread is considered interactive if its score is under a certain threshold, 30 by default as in FreeBSD11.1. With a niceness value of 0, this corresponds roughly to spending more than 60% of the time sleeping. Otherwise, it is classified as batch. A negative *nice* value (high priority) makes it easier for a thread to be considered interactive.

When a thread is created, it inherits the runtime and sleeptime (and thus the interactivity) of its parent. When a thread dies, its runtime in the last 5 seconds is returned to its parent. This penalizes parents that spawn batch children when being interactive.

Inside the interactive and batch runqueues, threads are further sorted by *priority*. The priority of interactive threads is a linear interpolation of their *score* (i.e., a thread with a penalty of 0 has the highest interactive

priority, while a thread with a penalty of 30 has the lowest interactive priority). Inside the interactive runqueue, there is one FIFO per priority. To add a thread to a runqueue, the scheduler inserts the thread at the end of the FIFO indexed by the thread's priority. Picking a thread to run from this runqueue is simply done by taking the first thread in the highest-priority non-empty FIFO.

The priority of batch threads depends on their runtime: the more a thread runs, the lower its priority. The niceness of the thread is added to get a linear effect on the priority. Inside the batch runqueue, there is also one FIFO per priority. Insertion and removal work as in the interactive case, with a slight difference. To avoid starvation between batch threads, ULE tries to be fair among batch threads by minimizing the difference of runtime between threads, similarly to what CFS does with the vruntime.

When picking the next thread to run, ULE first searches in the interactive runqueue. If an interactive thread is ready to be scheduled, it returns that thread. If the interactive runqueue is empty, ULE searches in the batch runqueue instead. If both runqueues are empty, that means that the core is idle, and no thread is scheduled.

The order in which ULE searches the runqeues effectively gives interactive threads absolute priority over batch threads. Batch threads may potentially starve if the machine executes too many interactive threads. However, it is thought that, as interactive threads by definition sleep more than they execute, starvation does not occur.

A thread runs for a limited period of time defined as a *timeslice*. Contrary to CFS, the rate at which a thread's timeslice expires is the same regardless of its priority. However, the value of a timeslice changes with the number of threads currently running on the core. When a core executes 1 thread, the timeslice is 10 ticks (78ms). When multiple threads are running, this value is divided by the number of threads while being constrained to a lower bound of 1 tick (1/127th of a second). In ULE, full preemption is disabled, meaning that only kernel threads can preempt others.

Load balancing: ULE only aims to even out the number of threads per core. In ULE, the load of a core is simply defined as the number of threads currently runnable on this core. Unlike CFS, ULE does not group threads into cgroups, but rather considers each thread as an independent entity.

When choosing a core for a newly created or awoken thread, ULE uses an affinity heuristic. If the thread is considered cache affine on the last core it ran on, then it is placed on this core. Otherwise, ULE finds the highest level in the topology that is considered affine, or the entire machine if none is available. From there, ULE first tries to find a core on which the minimum priority is higher than that of this thread. If that fails, ULE tries again, but now on all cores of the machine. If this also fails, ULE simply picks the core with the lowest number of running threads on the machine.

ULE also balances threads periodically, every 500-1500ms (the duration of the period is chosen randomly). Unlike CFS, the periodic load balancing is performed only by core 0. Core 0 simply tries to even out the number of threads amongst the cores as follows: a thread from the most loaded core, the (*donor*), is migrated to the less loaded core, the (*receiver*). A core can only be a donor or a receiver once, and the load balancer iterates until no donor or receiver is found, meaning that a core may give away or receive at most one thread per load balancer invocation.

ULE also balances threads when the interactive and batch runqueues of a core are empty. ULE tries to steal from the most loaded core with which the idle core shares a cache. If ULE fails to steal a thread, it tries at a higher level of the topology and so on, until it finally manages to steal a thread. As with the periodic load balancer, the idle stealing mechanism steals at most one thread.

Periodic load balancing in ULE happens less often than in CFS, but more computation is involved in selecting a core during thread placement in ULE. The rationale is that having a better initial thread placement avoids the need for frequently running a global load balancer.[1]

3 Porting ULE to the Linux kernel

In this section we describe the problems encountered when porting ULE to Linux, and the main differences between our port and the original FreeBSD code.

The Linux kernel offers an API to add new schedulers to the kernel. Schedulers must implement the set of functions presented in Table 1. These functions are responsible for adding and removing threads in runqueues, picking threads to be scheduled, and placing threads on cores.

FreeBSD does not offer such an API to schedulers. Instead, it declares prototypes of the functions that must be defined, meaning that only one scheduler can be used at a time, as opposed to Linux, in which multiple scheduling classes can co-exist. Fortunately, functions inside the ULE scheduler can easily be mapped to their counterparts in Linux (see Table 1). In the few cases where the interfaces do not match, it was possible to find a workaround. For instance, Linux uses a single function to enqueue newly created threads and threads that have been woken up, while FreeBSD uses two functions. Linux uses a flag parameter in its function to distinguish between the two cases. It then suffices to use this flag to choose the corresponding FreeBSD function.

[1] In recent versions of FreeBSD, due to a bug, the periodic load balancer never executes [1]. In our ULE code we fixed the bug, and the load balancer is executed periodically.

Linux	FreeBSD equivalent	Usage
`enqueue_task`	`sched_add` for new threads `sched_wakeup` for woken up threads	Enqueue a thread in a runqueue
`dequeue_task`	sched_rem	Remove a thread from a runqueue
`yield_task`	sched_relinquish	Yield the CPU back to the scheduler
`pick_next_task`	sched_choose	Select the next task to be scheduled
`put_prev_task`	sched_switch	Update statistics about the task that just ran
`select_task_rq`	sched_pickcpu	Choose the CPU on which a new (or waking up) thread should be placed

Table 1: Linux scheduler API and equivalent functions in FreeBSD.

Other than interfaces, CFS and ULE also differ in some low-level assumptions. The most notable difference is related to the presence or absence of the current thread in the runqueue data structures. The Linux scheduling class mechanism relies on the assumption that the current thread stays in the runqueue data structure while it runs on a core. In ULE, a thread that runs on a core is removed from the runqueue data structure, and added back when its timeslice is depleted, so that the FIFO property holds. When trying to implement this behavior in Linux, we encountered several showstopper issues, such as kernel crashes for threads that tried to change their scheduling class. We decided instead to adhere to the Linux way of doing things, and leave the currently running thread in the runqueue. Because of that, we had to slightly change the ULE load balancing to avoid migrating a currently running thread.

Furthermore, in ULE, when migrating a thread from one CPU to another, the scheduler acquires the lock on both runqueues. In Linux, this locking mechanism lead to deadlocks. Therefore, we modified the ULE load balancing code to use the same mechanism as that of CFS.

Finally, in FreeBSD, ULE is responsible for scheduling all threads, whereas Linux uses different scheduling policies for different priority ranges (e.g., a realtime scheduler for high priority threads). In this work, we are mainly interested in comparing workloads with priorities falling in the CFS range (100-139). Hence, we scaled down the ULE penalty scores to fit within the CFS range.

4 Experimental environment

4.1 Machines

We evaluate ULE on a 32-core machine (AMD Opteron 6172) with 32GB of RAM. All experiments were performed on the latest LTS Linux kernel (4.9). We also ran experiments on a smaller desktop machine (8-core Intel i7-3770), reaching similar conclusions. Due to space limitations, we omit these results from the paper.

4.2 Workloads

To assess the performance of CFS and ULE, we used both synthetic benchmarks and realistic applications. Fibo is a synthetic application computing Fibonacci numbers. Hackbench [10] is a benchmark designed by the Linux community to stress the scheduler. It creates a large number of threads that run for a short amount of time and exchange data using pipes. We also selected 16 applications from the Phoronix test suite [2] based on their completion time. We excluded Phoronix applications that take more than 10 minutes to complete on a single core, or that were too short to allow reliable time measurements. The 16 Phoronix applications are: compilation benchmarks (build-apache, build-php), compression (7zip, gzip), image processing (c-ray, dcraw), scientific (himeno, hmmer, scimark), cryptography (john-the-ripper) and web (apache). We use the NAS benchmark suite [6] to benchmark HPC applications, the Parsec benchmark suite [7] to benchmark parallel applications, and Sysbench [3] with MySQL and RocksDB [16] as database benchmarks. We use a read-write workload for sysbench and RocksDB to schedule threads with different behaviors.

5 Evaluation of per-core scheduling

In this section, we run applications on a single core to avoid possible side effects introduced by the load balancer. The main difference between CFS and ULE in per-core scheduling is in the handling of batch threads: CFS tries to be fair to all threads, while ULE gives priority to interactive threads. We first analyze the impact of this design decision by co-scheduling a batch and an interactive application on the same core, and we show that under ULE batch applications can starve for an unbounded amount of time. We then show that starvation under ULE can occur even when the system is only running a single application. We conclude this section by comparing the performance of 37 applications, and show how different choices regarding the preemption of threads impact performance.

5.1 Fairness and starvation when co-scheduling applications

In this section, we analyse the behavior of CFS and ULE running a multi-application workload consisting of a compute-intensive thread that never sleeps (fibo, computing numbers), and an application whose threads mostly sleep (sysbench, a database, using 80 threads). Having more than 80 threads per core is not uncommon in datacenters [12]. These threads are never all active at the same time; they mostly wait for incoming requests, or for data stored on disk.

Fibo runs alone for 7 seconds, and then sysbench is launched. Both applications then run to completion. Figure 1(a) presents the runtime accumulated by fibo and sysbench on CFS, and Figure 1(b) presents the same quantities on ULE.

On CFS, sysbench completes in 235s, and then fibo runs alone. Both fibo and sysbench threads share the machine. When sysbench executes, the cumulative runtime of fibo progresses roughly half as fast as when it runs alone, meaning that fibo gets 50% of the core. This is expected: CFS tries to share the core fairly between the two applications. In practice, fibo gets a bit less than half of the CPU due to rounding errors.

On ULE, sysbench is able to complete the same workload in 143s, and then fibo runs by itself. fibo starves while sysbench is running. sysbench threads mainly sleep, so they are classified as interactive and get absolute priority over fibo. Since sysbench uses 80 threads, these threads are able to saturate a core, and prevent fibo from running. Figure 2 presents the evolution of the interactivity penalty of fibo and sysbench over time. Both applications start out as interactive (penalty of 0). The penalty of fibo quickly rises to the maximum value, and then fibo is no longer considered interactive. Sysbench threads, in contrast, remain interactive during their entire execution (penalty below the 30 limit). Thus, sysbench threads get absolute priority over the fibo thread. This situation persists as long as sysbench is running (i.e., the starvation time is not bounded by ULE).

Table 2 presents the total execution time of fibo and sysbench on CFS and ULE, and the latency of requests for sysbench. Sysbench runs 50% slower on CFS, because it shares the CPU with fibo, instead of running in isolation, as it does with ULE (290 transactions/s with CFS vs. 532 with ULE). Fibo is "stopped" during the execution of sysbench on ULE, but then gets to execute by itself, and thus can use the cache more efficiently than when running simultaneously with sysbench on CFS. Thus, fibo runs slightly faster on ULE than on CFS.

We found the strategy used by the ULE scheduler to work well with latency-sensitive applications. These applications are usually correctly classified as interactive and get priority over background threads. To achieve the same result in Linux, the latency-sensitive application would have to be executed by the realtime scheduler, which gets absolute priority over CFS.

	CFS	ULE
Fibo - Runtime	160s	158s
Sysbench - Transactions/s	290	532
Sysbench - Avg. latency	441ms	125ms

Table 2: Execution time of fibo and sysbench using CFS and ULE, average latency of requests in sysbench using CFS and ULE.

5.2 Fairness and starvation within a single application

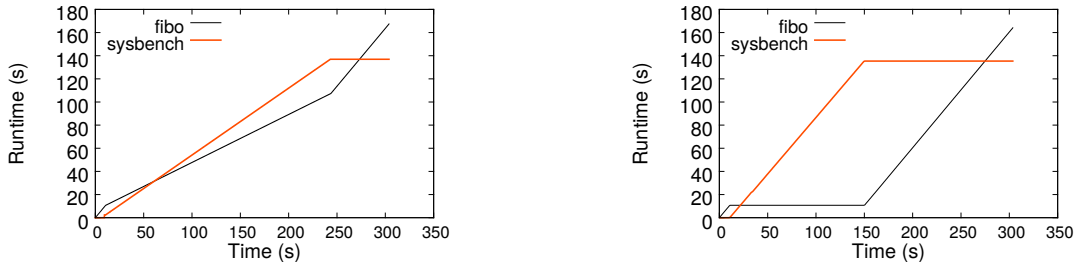
The starvation exhibited by ULE in the multi-application scenario above also occurs in single-application workloads. We now exemplify this behavior using sysbench.

In ULE, newly created threads inherit the interactivity penalty of their parent at the time of the fork. In sysbench, the master thread is created with the interactivity penalty of the bash process from which it was forked. Since bash mostly sleeps, sysbench is created as an interactive process. The sysbench master thread initializes data and creates threads. While doing so, it never sleeps, and its interactivity penalty increases. The first threads are created with an interactivity penalty below the interactive threshold, while the remaining threads are created with an interactivity penalty above it. As a consequence, the first threads get absolute priority over the remaining ones. Since these threads spend most of their time waiting for new requests, their interactivity penalty stays low (it decreases to 0), and their priority remains higher than that of threads that were forked late in the initialization process. The latter threads sysbench may starve forever, if the interactive threads keep the CPU busy at all times.

Figure 3 presents the cumulative runtime of sysbench threads, and Figure 4 presents their interactivity penalty. Sysbench is configured to use 128 threads. The threads created early execute, and their interactivity penalty drops to 0. The threads created later never execute.

Counterintuitively, in this benchmark ULE actually performs better than CFS, because it avoids oversubscription: the machine runs as many threads as it can. As a consequence, ULE has a lower average latency than CFS. In general, we found that this starvation mechanism, seemingly problematic on paper, performs very well in applications where all threads compete to perform the same job.

In contrast to sysbench, the scientific applications we tested are not impacted by starvation, because their threads never sleep. After a short initialization period



(a) (b)

Figure 1: Cumulative runtime of fibo, and sysbench on (a) CFS, and (b) ULE. (a) On CFS, fibo continues to accumulate runtime, albeit more slowly, when sysbench executes, meaning that fibo is not starved. (b) On ULE, when sysbench executes, fibo stops accumulating runtime, meaning that it is starved.

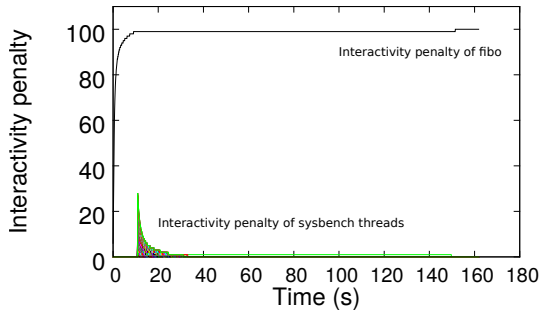


Figure 2: Interactivity penalty of threads over time. Fibo's penalty quickly rises to the maximum value, while the penalty of sysbench threads drops to 0.

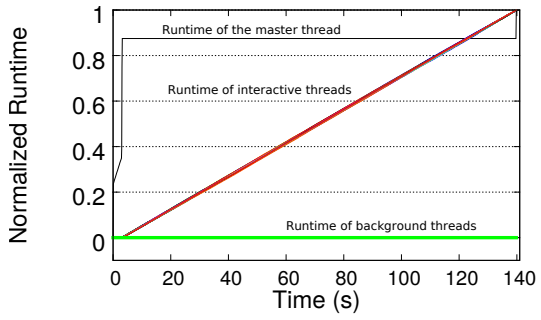


Figure 3: Cumulative runtime of threads of sysbench on ULE. The master thread first spawns 128 threads. 80 threads are classified as interactive and are executed, and 48 threads are classified as background and starve.

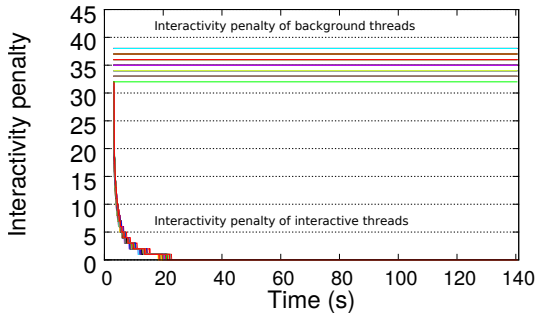


Figure 4: Interactivity penalty of the threads presented in Figure 3. Threads inherit the interactivity penalty of their parent when created. Some are created with a low penalty, and their penalty decreases as they execute (bottom of the graph), while other threads are created with a high penalty and never execute (top of the graph).

all threads are considered as background threads and are scheduled in a fair manner.

5.3 Performance analysis

We now analyze the impact of the per-core scheduling on the performance of 37 applications. We define "performance" as follows: for database workloads and NAS applications, we compare the number of operations per second, and for the other applications we compare "1/execution time". The higher the "performance", the better a scheduler performs. Figure 5 presents the performance difference between CFS and ULE on a single core, with percentages above 0 meaning that the application executes faster with ULE than CFS.

Overall, the scheduler has little influence on most workloads. Indeed, most applications use threads that all perform the same work, thus both CFS and ULE end up scheduling all of the threads in a round-robin fashion. The average performance difference is 1.5%, in favor of

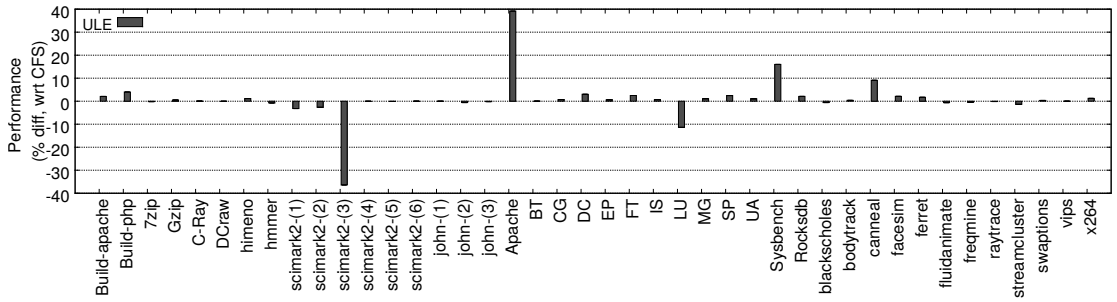


Figure 5: Performance of ULE with respect to CFS on a single core. A number higher than 0 means that the application runs faster on ULE than on CFS.

ULE. Still, scimark is 36% slower on ULE than CFS, and apache is 40% faster on ULE than CFS.

Scimark is a single-threaded Java application. It launches one compute thread, and the Java runtime executes other Java system threads in the background (for the garbage collector, I/O, etc.). When the application is executed with ULE, the compute thread can be delayed, because Java system threads are considered interactive and get priority over the computation thread.

The apache workload consists of two applications: the main server (*httpd*) running 100 threads, and *ab*, a single-threaded load injector. The performance difference between ULE and CFS is explained by different choices regarding thread preemption.

In ULE, full preemption is disabled, while CFS preempts the running thread when the thread that has just been woken up has a vruntime that is much smaller than the vruntime of the currently executing thread (1ms difference in practice). In CFS, *ab* is preempted 2 million times during the benchmark, while it never preempted with ULE. This behavior is explained as follows: *ab* starts by sending 100 requests to the httpd server, and then waits for the server to answer. When *ab* is woken up, it checks which requests have been processed and sends new requests to the server. Since *ab* is single-threaded, all requests sent to the server are sent sequentially. In ULE, *ab* is able to send as many new requests as it has received responses. In CFS, every request sent by *ab* wakes up a *httpd* thread, which preempts *ab*.

6 Evaluation of the load balancer

In this section, we analyze the impact of the load balancing and thread placement strategies on performance. In CFS and ULE, load balancing happens periodically, and thread placement occurs when threads are created or woken up. We first analyze the time it takes for the periodic load balancer to balance a static workload on all cores of the machine. We then analyze design choices made by CFS and ULE when placing threads. Next, we compare the performance of 37 applications running on CFS vs. ULE. Finally, we analyze the performance of multi-application workloads.

6.1 Periodic load balancing

CFS relies on a rather complex load metric. It uses a hierarchical load balancing strategy that runs every 4ms. ULE only tries to even out the number of threads on the cores. Load balancing happens less often (the period varies between 0.5s and 1.5s) and ignores the hardware topology. We now evaluate how these strategies impact the time needed to balance the load on the machine.

To that end, we pin 512 spinning threads on core 0, we launch a taskset command to unpin the threads, and we let the load balancer balance the load between cores. All threads perform the same work (an infinite empty loop), so we expect the load balancer to place 16 threads on each of the 32 cores. Figure 6 presents the evolution over time of the number of threads per core. In the figure, each of the 32 lines represents the number of threads on a given core. The taskset command that unpins threads is launched at 14.5s.

On ULE, as soon as the threads are unpinned, idle cores steal threads (at most one per core) from core 0, thus right after the unpinning, core 0 has $512-31=481$ threads while every other core has 1 thread. Over time, the periodic load balancer is triggered and tries to balance the thread count. However, as the load balancer only migrates one thread at a time from core 0, it takes more that 450 load balancer invocations or about 240 seconds to reach a balanced state.

CFS balances the load much faster. 0.2 seconds after the unpinning, CFS has migrated more that 380 threads from core 0. Surprisingly, CFS never achieves perfect load balance. CFS only balances the load between NUMA nodes when the imbalance between the two nodes is "big enough" (25% load difference in practice). So cores in one node can have 18 threads while cores in another only have 15.

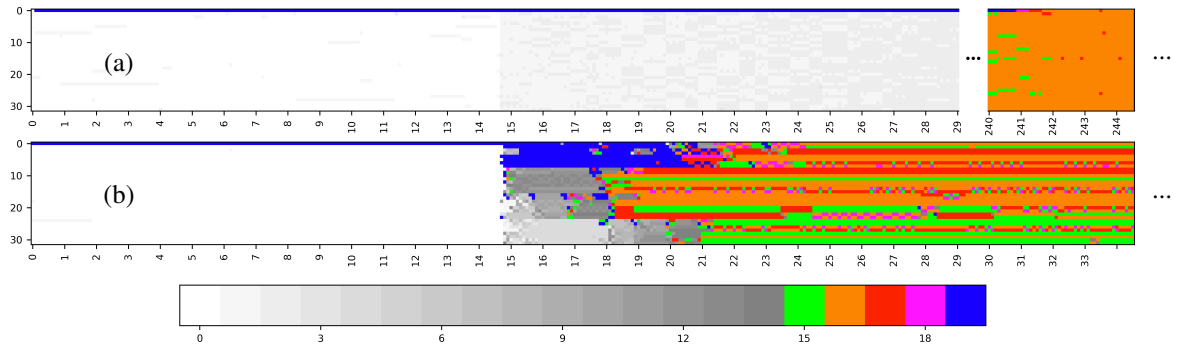


Figure 6: Number of threads per core over time on (a) ULE and (b) CFS. Each line represents a core (32 in total), time passes on the x-axis (in seconds), and colors represent the numbers of threads on the core. Thread counts below 15 are represented in shades of grey. Threads are pinned on core 0 for the first 14.5 seconds of the execution.

While the load balancing strategy used by CFS is well suited for solving a large imbalance loads in the system, it is less suited when a perfect load balance is important for performance.

6.2 Thread placement

We study placement of threads using c-ray, an image processing application from the phoronix benchmark suite. C-ray starts by creating 512 threads. Threads are not pinned at creation time, so the scheduler chooses a core for each thread. Then all threads wait on a barrier before performing the computation. Since all threads behave in the same way, we would expect ULE to perform better than CFS in that configuration: ULE always forks threads on the core with the lowest number of threads, so the load should be perfectly balanced from the start.

Figure 7 presents the evolution in the number of runnable threads per core over time. Load is always balanced in ULE, but surprisingly it takes more than 11 seconds for ULE to have all threads runnable, while it only takes 2 seconds for CFS. This delay is explained by starvation. C-ray uses a cascading barrier in which thread 0 wakes up thread 1, thread 1 wakes up thread 2, etc. Threads are originally created with different interactivity penalties, and some threads are initially interactive, while others are initially batch (same reason as in sysbench, see Section 5.2). When a batch thread is woken up, it might starve until all interactive threads are done, or until their penalty has increased enough for them to be downgraded to the batch runqueue. In practice, in c-ray, threads never sleep after the barrier, so eventually all threads become batch, but, before they do, threads that were initially categorized as batch cannot wake up other threads. Thus, it takes 11 seconds for all threads to be woken up after the barrier.

CFS on the contrary is fair, and all threads are quickly woken up. Then, CFS runs into the imperfect load balancing issue that we explained in Section 6.1.

Despite these load balancing differences, c-ray completes in the same time on CFS and ULE, because c-ray creates more threads than cores, and because both schedulers always keep all cores busy. Preemptions do occur more often with CFS, but do not affect the performance.

6.3 Performance analysis

Figure 8 presents the performance difference between CFS and ULE in a multicore context. The average performance difference between CFS and ULE is small: 2.75% in favor of ULE.

MG, from the NAS benchmark suite, benefits the most from ULE's load balancing strategy: it is 73% faster on ULE than on CFS. MG spawns as many threads as there are cores in the machine, and all threads perform the same computations. When a thread has finished its computation, it waits on a spin-barrier for 100ms and then sleeps if some threads are still computing. ULE correctly places one thread per core, and then never migrates them again. Threads spend very little time waiting for each other in the barriers, and never sleep. In contrast, CFS reacts to micro changes in the load of cores (e.g., due to a kernel thread waking up), and sometimes wrongly places two MG threads on the same core. Since MG uses barriers, the two threads scheduled on the same core end up delaying the whole application. The delay is more than 50% because threads scheduled alone on their cores go to sleep, and then have to be woken up, thus adding latency to the barriers. This suboptimal thread placement also explains the performance difference between CFS and ULE on FT and UA. The simple approach of balancing the number of threads used by ULE works better on HPC-like applications because it ends up placing one thread per core and then never migrates them again.

Sysbench, is slower on ULE due to the overhead of the ULE load balancer. When a thread wakes up, ULE scans

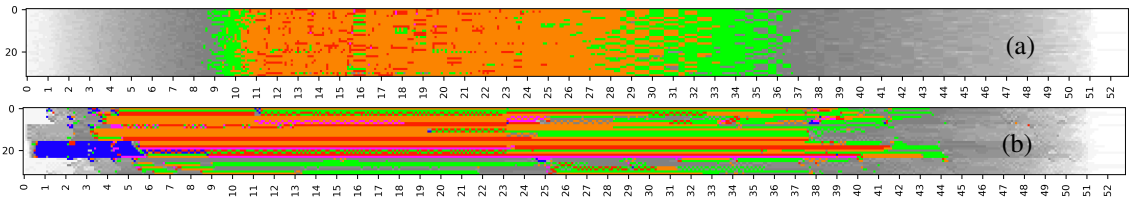


Figure 7: Number of threads per core over time on c-ray on (a) ULE and (b) CFS. Contrary to Figure 6, threads do not start pinned on core 0.

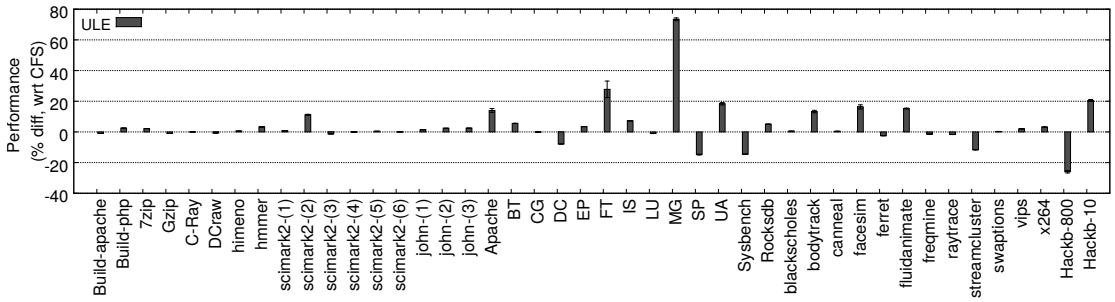


Figure 8: Performance of ULE with respect to CFS on a multicore.

the cores of the machine to find an appropriate core for the thread, and, at worst, may scan all cores three times. This worst case scenario happens on most wakeups in sysbench, resulting in 13% of all CPU cycles being spent on scanning cores. To validate this assumption, we replaced the ULE wakeup function by a simple one that returns the CPU on which the thread was previously running, and then observed no difference between ULE and CFS.

In all the benchmarks we tested, 13% is the highest time spent in the scheduler we observed in ULE, and 2.6% is the highest time spent in the scheduler we observed in CFS. Note that ULE runs into a corner case situation with sysbench, but has a low overhead on other benchmarks, even when they spawn a large number of threads: for instance in hackbench (32 000 threads), the overhead of ULE is 1% (compared to 0.3% for CFS).

6.4 Multi application workloads

Finally, we evaluate the combination of interactive and background workloads using a set of different applications: c-ray + EP (from NAS) is a workload where both applications are considered background by ULE, fibo + sysbench and blackscholes + ferret are workloads where only one application is interactive, and apache + sysbench is a fully interactive workload. Figure 9 shows the performance of CFS and ULE with respect to the performance of the application running alone on the machine (higher is better). Overall, most applications run slower when they are co-scheduled with another application.

When both applications are non-interactive (c-ray + EP), CFS and ULE perform similarly. This is expected, as they schedule background threads in a similar way. EP runs slightly faster on ULE when executed alone, and this performance difference is still present when it is co-scheduled with c-ray. When both applications are interactive, CFS and ULE also perform similarly.

For blackscholes + ferret, ULE gives priority to the interactive application, and ferret is not impacted by being co-scheduled with blackscholes. Blackscholes however runs more than 80% slower. In that context, blackscholes does not fully starve because ferret does not use 100% of all cores. CFS on the contrary shares the CPU fairly, and both applications suffer equally (the impact of co-scheduling on these applications is less than 50% because neither ferret nor blackscholes scales to 32 cores).

Surprisingly when co-scheduled with fibo, sysbench performs worse on ULE than on CFS even though it is correctly categorized as interactive and gets priority over fibo threads. The lack of preemption in ULE explains the performance difference. MySQL does not achieve perfect scaling and, when executed on 32 cores, lock contention forces the threads to sleep when waiting for the locks to be released. Thus, fibo does not starve. When a MySQL lock is released, ULE does not preempt the currently running thread (usually fibo) to schedule a new MySQL thread to enter the critical section. This adds delays (of up to the length of fibo's timeslice, between 7.8ms and 78ms) to the execution of sysbench.

7 Related work

Previous works have compared the design choices made by FreeBSD and Linux. Abaffy et al. [4, 5] compare the

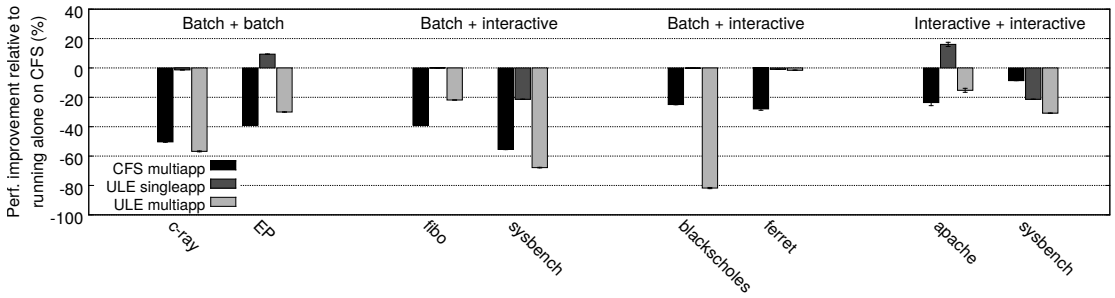


Figure 9: Performance of CFS and ULE on multi application workloads with respect to the performance of the application running alone on CFS.

average waiting time of threads in scheduler runqueues. Schneider et al. [17] compare the networking stack performance of the two operating systems. Design choices made by FreeBSD are also frequently discussed on the Linux kernel mailing list [20]. This study differs in its approach: instead of comparing two complete operating systems, we ported the FreeBSD ULE scheduler to Linux. To the best of our knowledge, this is the first apples-to-apples comparison of the design of ULE and CFS.

The Linux scheduler design has also been discussed in previous works. Torrey et al. [19] compare the latency of the Linux scheduler against a custom implementation of a multilevel feedback queue. Wong et al. compare the fairness of CFS with the O(1) scheduler that was the default Linux scheduler prior to 2.6.23 [23], and with a RSDL scheduler (Rotating Staircase Deadline Scheduler) [22]. Groves et al. [9] compare the overhead of CFS against BFS (Brain Fuck Scheduler), a simplistic scheduler aimed at improving responsiveness on machines with few cores. Other work has studied the overhead of schedulers. Kanev et al. [12] report that the CFS alone accounts for more than 5% of all datacenter cycles.

The performance of operating systems is frequently assessed by measuring the evolution of performance between kernel versions. The Linux Kernel Performance project [8] started in 2005 to measure performance regressions in the Linux Kernel. Mollison et al. [14] propose Litmus tests to find performance regressions in schedulers. Performance issues in operating systems are also frequently reported in the Systems community. Lozi et al. [13] report bugs in the Linux scheduler that could lead to cores being permanently left idle while work was waiting to be scheduled on other cores. Harji et al. [11] report similar performance bugs in earlier kernel versions. During the work on this paper we also reported bugs in the scheduler to the FreeBSD community [1]. In this study we chose to compare "glitch free" versions of ULE and CFS by fixing obvious bugs that were not intended as features of the schedulers.

8 Conclusion

Scheduling threads on a multicore machine is hard. In this paper, we perform a fair comparison of the design choices of two widely used schedulers: the ULE scheduler from FreeBSD and CFS from Linux. We show that they behave differently even on simple workloads, and that no scheduler performs better than the other on all workloads.

References

[1] [PATCH] Fix bug in which the long term ULE load balancer is executed only once. `https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=223914`.

[2] Phoronix Test Suite. `https://www.phoronix.com`.

[3] Sysbench. `https://github.com/akopytov/sysbench`.

[4] ABAFFY, J., AND KRAJČOVIČ, T. Latencies in Linux and FreeBSD kernels with different schedulers-O(1), CFS, 4BSD, ULE. In *Proceedings of the 2nd International Multi-Conference on Engineering and Technological Innovation* (2009), pp. 110–115.

[5] ABAFFY, J., AND KRAJČOVIČ, T. Pi-ping-benchmark tool for testing latencies and throughput in operating systems. *Innovations in Computing Sciences and Software Engineering* (2010), 557–560.

[6] BAILEY, D., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H., VENKATAKRISHNAN, V., AND WEERATUNGA, S. The NAS parallel benchmarks summary and pre-

liminary results. In *Supercomputing '91*. (Nov. 1991), pp. 158–165.

[7] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT* (2008).

[8] CHEN, T., ANANIEV, L. I., AND TIKHONOV, A. V. Keeping kernel performance from regressions. In *Linux Symposium* (2007), vol. 1, pp. 93–102.

[9] GROVES, T., KNOCKEL, J., AND SCHULTE, E. BFS vs. CFS scheduler comparison. *The University of New Mexico* (2009). http://cs. unm. edu/~eschulte/classes/cs587/data/bfsv-cfs_groves-knockel-schulte. pdf (accessed Jan. 5, 2017).

[10] Hackbench. `https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/tree/src/hackbench?h=v0.93`, 2008.

[11] HARJI, A. S., BUHR, P. A., AND BRECHT, T. Our troubles with Linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, pp. 2:1–2:5.

[12] KANEV, S., DARAGO, J. P., HAZELWOOD, K., RANGANATHAN, P., MOSELEY, T., WEI, G.-Y., AND BROOKS, D. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on* (2015), IEEE, pp. 158–169.

[13] LOZI, J.-P., LEPERS, B., FUNSTON, J., GAUD, F., QUÉMA, V., AND FEDOROVA, A. The Linux scheduler: a decade of wasted cores. In *EuroSys'16* (2016), ACM, pp. 1:1–1:16.

[14] MOLLISON, M. S., BRANDENBURG, B., AND ANDERSON, J. H. Towards unit testing real-time schedulers in LITMUS^{RT}. In *Proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications* (2009), OSPERT.

[15] ROBERSON, J. ULE: a modern scheduler for FreeBSD.

[16] ROCKSDB - PERFORMANCE BENCHMARKS. `https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks`.

[17] SCHNEIDER, F., AND WALLERICH, J. Performance evaluation of packet capturing systems for high-speed networks. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology* (2005), ACM, pp. 284–285.

[18] THE DESIGN OF CFS. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[19] TORREY, L. A., COLEMAN, J., AND MILLER, B. P. A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler. *Software: Practice and Experience 37*, 4 (2007), 347–364.

[20] TORVALDS, L. Is sendfile all that sexy? `http://yarchive.net/comp/linux/sendfile.html`.

[21] ULE PORT IN LINUX. `https://github.com/JBouron/linux/tree/loadbalancing`.

[22] WANG, S., CHEN, Y., JIANG, W., LI, P., DAI, T., AND CUI, Y. Fairness and interactivity of three CPU schedulers in Linux. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on* (2009), IEEE, pp. 172–177.

[23] WONG, C., TAN, I., KUMARI, R., LAM, J., AND FUN, W. Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In *Information Technology, 2008. ITSim 2008. International Symposium on* (2008), vol. 4, IEEE, pp. 1–8.