Your Cores Are Slacking Off—
Or Why OS Scheduling Is a Hard Problem

JEAN-PIERRE LOZI, BAPTISTE LEPERS, JUSTIN FUNSTON, FABIEN GAUD, VIVIEN QUÉMA, AND ALEXANDRA FEDOROVA

Jean-Pierre Lozi is an Associate Professor at the University of Nice Sophia-Antipolis, in the French Riviera. When he’s not reading OS papers on the beach, you can usually find him around Château Valrose, teaching multicore programming, big data, and other hot topics.
jplozi@unice.fr

Baptiste Lepers is a postdoc at EPFL. His research topics include performance profiling, optimizations for NUMA systems, multicore programming, and proofs of concurrent programs.
baptiste.lepers@gmail.com

Justin Funston is a PhD student at the University of British Columbia and is advised by Alexandra Fedorova. His research interests include contention management on multicore and NUMA systems, parallel and high performance computing, and operating systems in general.
jfunston@ece.ubc.ca

Fabien Gaud is a Senior Software Engineer at Coho Data, focusing on performance and scalability. He received his PhD in 2010 from Grenoble University, and from 2011 to 2014 he was a postdoctoral fellow at Simon Fraser University.
me@fabiengaud.net

As a central component of resource management, the OS thread scheduler must make sure that ready threads are scheduled on available cores. As surprising as it may seem, we found that this simple rule is often broken in Linux. Cores may stay idle for seconds while ready threads are waiting in run queues, delaying applications and wasting energy. This phenomenon is not due to an intentional design but to performance bugs. These bugs can slow down scientific applications many-fold and degrade performance of workloads like kernel compilation and OLAP on a widely used commercial database by tens of percent, particularly on machines with a large number of cores. The root cause of the bugs is the increasing scheduler complexity, linked to rapid evolution in modern hardware. In this article, we describe the bugs and their effects and reflect on ways to combat them.

Our recent experience with the Linux scheduler revealed that the pressure to work around the challenging properties of modern hardware, such as non-uniform memory access (NUMA) latencies, high costs of cache coherence and synchronization, and diverging CPU and memory latencies, resulted in a scheduler with an incredibly complex implementation. As a result, the very basic function of the scheduler, which is to make sure that runnable threads use idle cores, fell through the cracks. We have discovered four performance bugs that cause the scheduler to leave cores idle while runnable threads are waiting for their turn to run. Resulting performance degradations are in the range 13–24% for typical Linux workloads, and reach many-fold slowdowns in some corner cases. In this article, we describe three of the four bugs. For the complete description, please refer to our extended paper [7].

Detecting the aforementioned bugs is difficult. They do not cause the system to crash or hang, but eat away at performance, often in ways that are difficult to notice with standard performance monitoring tools. For instance, when executing the OLAP workload TPC-H on a widely used commercial database, the symptom occurred many times throughout the execution, but each time it lasted only a few hundreds of milliseconds—too short to detect with tools like htop, sar, or perf. Yet, collectively, these occurrences did enough damage to slow down the most affected query by 23%. Even in cases where the symptom was present for a much longer duration, the root cause was difficult to discover because it was a result of many asynchronous events in the scheduler.

In the rest of this article we provide relevant background on the Linux scheduler, describe the bugs and their root causes, demonstrate their performance effects, and finally reflect on ways to combat them, focusing especially on the tools that were crucial for the bug discovery.

The Linux Scheduler

Linux’s Completely Fair Scheduling (CFS) is an implementation of the weighted fair queueing (WFQ) scheduling algorithm, wherein the available CPU cycles of each core are divided among threads in proportion to their weights. To support this abstraction, CFS time-slices the CPU cycles among the running threads.
The scheduler defines a fixed time interval during which each thread in the system must run at least once. The interval is divided among threads proportionally to their weights. The resulting interval (after division) is what we call the timeslice. A thread’s weight is essentially its priority, or niceness in UNIX parlance. Threads with lower niceness have higher weights and vice versa.

When a thread runs, it accumulates vruntime (the runtime of the thread divided by its weight). Once a thread’s vruntime exceeds its assigned timeslice, the thread is preempted from the CPU if there are other runnable threads available. A thread might also get preempted if another thread with a smaller vruntime is awoken.

Threads are organized in run queues. As a matter of efficiency, there is one run queue per core. When a core looks for a new thread to run, it picks the thread in its run queue that has the smallest vruntime.

For the overall system to be efficient, run queues must be kept balanced. To this end, CFS periodically runs a load-balancing algorithm that will keep the queues roughly balanced.

CFS balances run queues based on a metric called load, which is the combination of the thread’s weight and its average CPU utilization. Intuitively, if a thread does not use much of a CPU, its load will be decreased accordingly. Additionally, the load-tracking metric accounts for varying levels of multithreading in different processes.

When a thread belongs to a group of threads (called a cgroup), its load is further divided by the total number of threads in its cgroup. Cgroups are used to group threads or processes that logically belong together, such as threads in the same application or processes launched from the same terminal (tty). This is done for fairness purposes, such that the CPU is shared among applications rather than individual instruction streams.

CFS implements a hierarchical load-balancing strategy. The cores are logically organized in a hierarchy, at the bottom of which is a single core. How the cores are grouped at the hierarchy’s next levels depends on how they share the machine’s physical resources. On the example machine illustrated in Figure 1, pairs of cores share functional units, such as the

![Figure 1](https://www.usenix.org/conference/winter2016/papers/730928.png)

Figure 1: A machine with 32 cores, four NUMA nodes (eight cores per node sharing a last-level cache), and pairs of cores sharing a floating point unit. The dashed lines outline the scheduling domains as perceived by the left-topmost core. Level 3 of the hierarchy shows a group of three nodes: that is because these nodes are reachable from the left-topmost core in a single hop. (See Figure 3 for a detailed overview of node connectivity in our system.) At the fourth level, we have all the nodes of the machine, because they can be reached from the left-topmost core in at most two hops.
floating point unit, and groups of eight cores share a last-
level cache; these groups of eight also form a NUMA node. As a
result, at the second level of the hierarchy we have pairs of cores,
and at the third level we have NUMA nodes. NUMA nodes are
further grouped according to their level of connectivity. This
is where things become a bit tricky, because the hierarchy is
constructed from the point of view of a particular “designated”
core; in the load-balancing algorithm it is the core that performs
load balancing. In Figure 1 the hierarchy levels are shown as if
the left-topmost core were designated. Hence, the third level of
the hierarchy includes all nodes that can be reached from that
designated core in one hop. The fourth level includes the nodes
that can be reached from it in at most two hops, i.e., all nodes in
the system.

Each level of the hierarchy is called a scheduling domain. If a
scheduling domain includes sub-domains, such as the NUMA-
ode domain including core-pair domains, those sub-domains
are referred to in Linux terminology as scheduling groups.

At the high level, the load-balancing algorithm works as follows.
Load balancing is run for each scheduling domain, starting
from the lowest level of the hierarchy that contains more than
a single core (the pair-of-cores level in our example) to the top.
At each level, the algorithm is run by the designated core. The
core is designated if it is either the first idle core of the domain
or, if none of the cores are idle, the core with the lowest ID in
the domain. The designated core computes the average load for
each scheduling group of the domain and picks the busiest group,
based on the load and on heuristics that favor overloaded and
imbalanced groups. If the load of the busiest group is higher than
the load of the designated core’s home group, the designated core
steals threads from the busiest group so as to balance the load.

The scheduler implements a set of optimizations to improve the
efficiency of the load-balancing mechanism, but as we will see
later they increase complexity and nurture bugs. For example,
in earlier versions of Linux, idle cores, which are typically
transitioned into a lower power state, were always awoken on
every clock tick; at this point they would run the load-balancing
algorithm. Since version 2.6.21, Linux included the option, now
enabled by default, to avoid periodically waking up sleeping
cores. It is the responsibility of overloaded cores to wake up the
sleeping cores when needed. Another set of optimizations has
to do with the placement of threads that become unblocked.
Normally when a thread wakes up, after sleeping or waiting for
a resource like a lock or I/O, the scheduler tries to place it on
the idlesst core. However, when a thread is awoken by another
“waker” thread, the scheduler will favor cores sharing a cache
with the waker thread to improve cache reuse.

The first bug we encountered is illustrated in Figures 2a and 2b.
The figures show the state of the scheduler when we execute a
workload on an eight-node NUMA system summarized in Table
1. The x-axis shows the time and the y-axis shows CPU cores.
Nodes are numbered 0–7. Each node contains eight cores.

The Group Imbalance Bug
The first bug we encountered is illustrated in Figures 2a and 2b.
The figures show the state of the scheduler when we execute a
workload on an eight-node NUMA system summarized in Table
1. The x-axis shows the time and the y-axis shows the cores,
grouped by their node number. In the time period shown in the
figure, the machine was executing a compilation of the kernel
(make with 64 threads) and running two make processes (each with
one thread). The make and the two make processes were launched
from three different ssh connections (i.e., three different ttys).
Figure 2a is a heatmap showing the number of threads in each
Your Cores Are Slacking Off—Or Why OS Scheduling Is a Hard Problem

core’s run queue over time. The chart shows that there are two nodes (zero and four) whose cores run either only one thread or no threads at all, while the rest of the nodes are overloaded, with many of the cores having two threads in their run queue.

After investigation, we found that the scheduler is not balancing the load properly. Remember that a thread’s load is a combination of its weight and its CPU utilization. Threads launched from the same tty belong to the same cgroup, and their load is thus divided by the number of threads in their cgroup. As a result, a thread in the 64-thread make process has a load roughly 64 times smaller than a thread in a single-threaded R process.

Discrepancies between threads’ loads are illustrated in Figure 2b, which shows the combined load of threads in each core’s run queue: a darker color corresponds to a higher load. Nodes 0 and 4, the ones running the R processes, each have one core with a very high load. These are the cores that run the R threads.

The Linux load balancer steals work from other run queues based on load; obviously the underloaded cores in Nodes 0 and 4 should not steal from the overloaded core in their own node, because that core runs only a single thread. However, they must be able to steal from the more loaded cores in other nodes. This is not happening for the following reason. Remember that to limit algorithmic complexity, the load-balancing algorithm uses a hierarchical design. When a core attempts to steal work from another node or, in other words, from another scheduling group, it does not examine the load of every core in that group, it only looks at the group’s average load. If the average load of the victim group is greater than that of its own, it will attempt to steal threads from that group; otherwise it will not. In our case, the idle core looking for work is in the same group as the high-load R thread. So the average load for that group is actually the same as the load of the group with many overloaded cores. As a result, no stealing occurs, despite the victim group having overloaded cores with waiting threads.

To fix this bug, we changed the part of the algorithm that compares the load of scheduling groups. Instead of comparing the average loads, we compare the minimum loads. The minimum load is the load of the least loaded core in that group. Intuitively, if the minimum load of one scheduling group is lower than the minimum load of another scheduling group, it means that the first scheduling group has a core that is less loaded than any core in the other group, and thus a core in the first group must steal from the second.

Figure 2c is a visualization of the same workload after we fixed the bug (showing a heatmap of run queue sizes, in the same fashion as Figure 2a). We observe that the imbalance disappears. With the fix, the completion time of the make job, in the make/R workload decreased by 13%. Performance impact could be much higher in other circumstances. For example, in a workload running lu from the NPB (NASA Advanced Supercomputing Parallel Benchmarks) suite with 60 threads, and four single-threaded R processes, lu ran 13x faster after fixing the Group Imbalance bug. lu experienced a super-linear speedup, because the bug exacerbated lock contention when multiple lu threads ran on the same core.

The Scheduling Group Construction Bug

Linux defines a command, called taskset, that enables pinning applications to run on a subset of the available cores. The bug we describe in this section occurs when an application is pinned on nodes that are two hops apart. For example, in Figure 3, which demonstrates the topology of our NUMA machine, Nodes 1 and 2 are two hops apart. The bug will prevent the load-balancing algorithm from migrating threads between these two nodes.

The bug results from the way scheduling groups are constructed, which is not adapted to modern NUMA machines such as the one we use in our experiments. In brief, the groups are constructed from the perspective of a specific core (Core 0), whereas they should be constructed from the perspective of the core responsible for load balancing on each node, the designated core. We explain with an example.

<table>
<thead>
<tr>
<th>CPUs</th>
<th>4 × 16-core Opteron 6272 CPUs (64 threads in total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock rate</td>
<td>2.1 GHz</td>
</tr>
<tr>
<td>Caches</td>
<td>64 KB shared L1 i-cache</td>
</tr>
<tr>
<td>(Each core)</td>
<td>16 KB L1 d-cache</td>
</tr>
<tr>
<td>Memory</td>
<td>512 GB of 1.6 GHz DDR-3</td>
</tr>
<tr>
<td>Interconnect</td>
<td>HyperTransport 3.0 (see Figure 3)</td>
</tr>
</tbody>
</table>

Table 1: Hardware of our AMD Bulldozer machine

Figure 3: Topology of our 8-node AMD Bulldozer machine
Let us walk through the key steps of the load-balancing algorithm when the balancing is performed at the top of the hierarchy, i.e., at the scheduling domain including all the machine's nodes. The algorithm will construct the scheduling groups (the sub-domains) included within that scheduling domain. The first scheduling group on the machine in Figure 3 will include Node 0 plus all the nodes that are one hop apart from Node 0, namely Nodes 1, 2, 4, and 6. The second group will include the lowest-numbered node that was not included in the first group: Node 3, in this case, and all nodes that are one hop apart from Node 3: Nodes 1, 2, 4, 5, 7. The two scheduling groups are thus: \{0, 1, 2, 4, 6\} and \{1, 2, 3, 4, 5, 7\}.

Suppose that an application is pinned on Nodes 1 and 2 and that all of its threads are being created on Node 1. Eventually we would like the load to be balanced between Nodes 1 and 2. However, when a core in Node 2 looks for work to steal, it will compare the load between the two scheduling groups shown earlier. Since each scheduling group contains both Nodes 1 and 2, the average loads will be the same, so Node 2 will not steal any work!

The bug originates from an attempt to improve the performance of Linux on large NUMA systems. Before the introduction of the bug, Linux would balance the load inside NUMA nodes and then across all NUMA nodes. New levels of hierarchy (nodes one hop apart, nodes two hops apart, etc.) were introduced to increase the likelihood for threads to remain close to their original NUMA node.

To fix the bug, we modified the construction of scheduling groups so that each core uses scheduling groups constructed from its own perspective. After the fix, the cores were able to detect the imbalance and to steal the work. Table 2 presents the performance difference in NPB applications with and without the Scheduling Group Construction bug. Applications are launched on two nodes with as many threads as there are cores. The maximum slowdown of 27x is experienced by lu. The slowdown is a lot more than the expected 2x because of locking effects.
the data produced by the waker thread. This seems like a reason-
able idea, but for some workloads, waiting in the run queue for
the sake of better cache reuse does not pay off.

This bug was triggered by and affected the runtime of a widely
used commercial database configured with 64 worker threads
(one thread per core) and running an OLAP (TPC-H) workload.

Figure 4 illustrates several instances of the Overload-on-
Wakeup bug. During the first time period (1), one core is idle
while a thread that ideally should be scheduled on that core
keeps waking up on other cores, which are busy. During the
second time period (2), there is a triple instance of the bug:
three cores are idle for a long time, while three extra threads
that should be scheduled on those cores keep waking up on other
busy cores. The Overload-on-Wakeup bug is typically caused
when a transient thread is scheduled on a core that runs a data-
base thread. When this happens, the load balancer observes a
heavier load on the node that runs the transient thread (Node A)
and migrates one of the threads to another node (Node B). This
is not an issue if the transient thread is the one being migrated,
but if it is the database thread, then the Overload-on-Wakeup
bug will kick in. Node B now runs an extra database thread, and
threads of Node B, which often sleep and wake up, keep waking
up on that node, even if there are no idle cores on that node. This
occurs because the wake up code only considers cores from the
local node for the sake of better cache reuse and results in a core
running multiple threads when some cores that are always idle
are present on other nodes.

To fix this bug, we alter the code that is executed when a thread
wakes up. We wake up the thread on the local core—i.e., the core
where the thread was scheduled last—if it is idle; otherwise, if
there are idle cores in the system, we wake up the thread on the
core that has been idle for the longest amount of time. If there are
no idle cores, we fall back to the original algorithm to find the
core where the thread will wake up.

Our bug fix improves performance by 22.6% on the 18th query of
TPC-H, and by 13.2% on the full TPC-H workload.

Discussions and Lessons Learned
The first question to ask is whether these bugs could be fixed
with a new, cleaner scheduler design that is less error-prone
and easier to debug, but still maintains the features we have today.
Historically, though, this does not seem like a long-term solu-
tion, in addition to the fact that the new design would need to be
implemented and tested from scratch. The Linux scheduler has
gone through a couple of major redesigns. The original scheduler
had high algorithmic complexity, which resulted in poor perfor-
ance when highly multithreaded workloads became common.
In 2001, it was replaced by a new scheduler with \(O(1)\) complexity
and better scalability on SMP systems. It was initially success-
ful but soon required modifications for new architectures like
NUMA and SMT (simultaneous multithreading). At the same
time, users wanted better support for desktop use cases such as
interactive and audio applications, which required more changes
to the scheduler. Despite numerous modifications and proposed
heuristics, the \(O(1)\) scheduler was not able to meet expectations
and was replaced by CFS in 2007. Interestingly, CFS sacrifices
\(O(1)\) complexity for \(O(\log (\# \text{ threads}))\), but it was deemed worth-
while to provide the desired features.

As the hardware and workloads became more complex, CFS,
too, succumbed to bugs. The addition of autogroups (i.e., the
automatic grouping of threads from the same tty into a cgroup)
coupled with hierarchical load balancing introduced the Group
Imbalance bug. Asymmetry in new, increasingly complex NUMA
systems triggered the Scheduling Group Construction bug.
Cache-coherency overheads on modern multi-node machines
motivated the cache locality optimization that caused the
Overload-on-Wakeup bug.

The takeaway is that new scheduler designs come and go. How-
ever, a new design, even if clean and purportedly bug-free
initially, is not a long-term solution. Linux is a large open-source
system developed by dozens of contributors. In this environment,
we will inevitably see new features and “hacks” retrofitted into
the source base to address evolving hardware and applications.

The recently released Linux 4.3 kernel features a new imple-
mentation of the load metric. This change is reported to be “done
in a way that significantly reduces complexity of the code” [1].
Simplifying the load metric could get rid of the Group Imbalance
bug, which is directly related to it. However, we confirmed, using
our tools (see [7] for a full description of our tools and the end of
this article for a link to the code), that the bug is still present.

Kernel developers rely on mutual code review and testing to pre-
vent the introduction of bugs. This could potentially be effective
for bugs, like the Scheduling Group Construction bug, that are
easier to spot in the code (of course, it still was not effective in
these cases), but it is unlikely to be reliable for the more arcane
types of bugs.

Catching these bugs with testing or conventional performance
monitoring tools is tricky. They do not cause the system to
crash or to run out of memory, but they do silently eat away at
performance. As we have seen with the Group Imbalance and
the Overload-on-Wakeup bugs, they introduce short-term idle
periods that “move around” between different cores. These
microscopic idle periods cannot be noticed with performance
monitoring tools like htop, sar, or perf. Standard performance
regression testing is also unlikely to catch these bugs, as they
occur in very specific situations (e.g., multiple applications
with different thread counts launched from distinct ttys). In
practice, performance testing on Linux is done with only one
application running at a time on a dedicated machine—this is the
standard way of limiting factors that could explain performance differences.

One of the most important lessons we learned in the process of finding and diagnosing these bugs is that it was crucially important to have the tools that trace and visualize microscopic events during the execution, such as the state of the kernel run queues and the transitions of threads between the cores. The visualizations that we relied on for detection and diagnosis of the bugs are shown in Figures 2 and 4. We built our own tools that perform precise tracing of kernel events and plot them as the space/time charts shown earlier.

Although we found it convenient to build our own tools, there is also a variety of powerful third-party dynamic tracing tools, such as DTrace, LTTNG, Event Tracing for Windows, Ftrace, and SystemTap. Unfortunately, effective visual front-ends and trace analysis tools that are necessary to make the traces useful are lacking. Most of the user-friendly performance tools available today rely on sampling and display averages and aggregates, which is not powerful enough for detecting performance anomalies like those caused by the scheduler bugs. We strongly feel that the software engineering community must embrace dynamic tracing and visualization for efficient diagnosis of egregious performance anomalies.

**Reflection on the Future of OS Scheduling**

The bugs we described resulted from increasingly more optimizations in the scheduler, whose purpose was mostly to cater to complexity of modern hardware. As a result, the scheduler, that once used to be a simple isolated part of the kernel, grew into a complex monster whose tentacles reached into many other parts of the system, such as power and memory management.

The optimizations studied in this paper are part of the mainline Linux, but even more scheduling optimizations were proposed in the research community.

Since 2000, dozens of papers have described new scheduling algorithms catering to resource contention, coherency bottlenecks, and other idiosyncrasies of modern multicore systems. There were algorithms that scheduled threads so as to minimize contention for shared caches, memory controllers, and multi-threaded CPU pipelines [2, 6, 8]. There were algorithms that reduced communication distance among threads sharing data [10]. There were algorithms that addressed scheduling on asymmetric multicore CPUs [4, 9] and algorithms that integrated scheduling with the management of power and temperature [3]. Finally, there were algorithms that scheduled threads to minimize communication latency on systems with an asymmetric interconnect [5]. All of these algorithms showed positive benefits, either in terms of performance or power, for some real applications. However, few of them were adopted in mainstream operating systems, mainly because it is not clear how to integrate all these ideas in the scheduler safely.

If every good scheduling idea is slapped as an add-on to a single monolithic scheduler, we risk more complexity and more bugs, as we saw from the case studies in this paper. Rapid evolution of hardware that we are witnessing today will motivate more and more scheduler optimizations. Instead of producing yet another monolithic scheduler design, what we may need is to switch to a more modular architecture.

One possible avenue is to decouple time management from space management. Historically, on single-core systems, the scheduler was tasked with managing time, that is, sharing the CPU cycles among the threads. On multicore systems, the scheduler evolved to also manage space, that is to decide where to place the threads. Several researchers postulated that space management need not be done at as fine a time granularity as time management, and this idea becomes more and more feasible in the age where machines are evolving to have more cores than most applications need. Perhaps space management could be done at coarse time intervals by placing groups of threads on subsets of cores such that each thread always has a free core whenever it needs one. Then time management would be largely out of the picture, and the space manager would deal with issues like NUMA locality and resource contention. It would adjust the mapping of threads to cores somewhat infrequently to reflect the changes in application behavior over time. Still, understanding how to combine different, perhaps conflicting, space optimizations and reason about how they interact remains an open research problem.

Our bug fixes and tools are available at http://git.io/vaGOW.
Your Cores Are Slacking Off—Or Why OS Scheduling Is a Hard Problem

References


