View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HAL-UNICE

The Linux Scheduler: a Decade of Wasted Cores

Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, Alexandra Fedorova

▶ **To cite this version:**

Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, et al.. The Linux Scheduler: a Decade of Wasted Cores. EuroSys 2016, Apr 2016, London, United Kingdom. Proceedings of the Eleventh European Conference on Computer Systems. <http://eurosys16.doc.ic.ac.uk/>. <10.1145/2901318.2901326>. <hal-01295194>

HAL Id: hal-01295194

https://hal.archives-ouvertes.fr/hal-01295194

Submitted on 31 Mar 2016

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

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

The Linux Scheduler: a Decade of Wasted Cores

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

Baptiste Lepers
EPFL
baptiste.lepers@epfl.ch

Justin Funston
University of British Columbia
jfunston@ece.ubc.ca

Fabien Gaud
Coho Data
me@fabiengaud.net

Vivien Quéma
Grenoble INP / ENSIMAG
vivien.quema@imag.fr

Alexandra Fedorova
University of British Columbia
sasha@ece.ubc.ca

Abstract

As a central part of resource management, the OS thread scheduler must maintain the following, simple, invariant: make sure that ready threads are scheduled on available cores. As simple as it may seem, we found that this invariant is often broken in Linux. Cores may stay idle for seconds while ready threads are waiting in runqueues. In our experiments, these performance bugs caused many-fold performance degradation for synchronization-heavy scientific applications, 13% higher latency for kernel `make`, and a 14-23% decrease in TPC-H throughput for a widely used commercial database. The main contribution of this work is the discovery and analysis of these bugs and providing the fixes. Conventional testing techniques and debugging tools are ineffective at confirming or understanding this kind of bugs, because their symptoms are often evasive. To drive our investigation, we built new tools that check for violation of the invariant online and visualize scheduling activity. They are simple, easily portable across kernel versions, and run with a negligible overhead. We believe that making these tools part of the kernel developers' tool belt can help keep this type of bug at bay.

1. Introduction

> "And you have to realize that there are not very many things that have aged as well as the scheduler. Which is just another proof that scheduling is easy."
>
> Linus Torvalds, 2001 [43]

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

EuroSys '16, April 18 - 21, 2016, London, United Kingdom
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901326

Classical scheduling problems revolve around setting the length of the scheduling quantum to provide interactive responsiveness while minimizing the context switch overhead, simultaneously catering to batch and interactive workloads in a single system, and efficiently managing scheduler run queues. By and large, by the year 2000, operating systems designers considered scheduling to be a solved problem; the Linus Torvalds quote is an accurate reflection of the general opinion at that time.

Year 2004 brought an end to Dennard scaling, ushered in the multicore era and made energy efficiency a top concern in the design of computer systems. These events once again made schedulers interesting, but at the same time increasingly more complicated and often broken.

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 latencies (NUMA), high costs of cache coherency 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.

The main contribution of this work is the discovery and study of four performance bugs in the Linux scheduler. These bugs cause the scheduler to leave cores idle while runnable threads are waiting for their turn to run.[1] Resulting performance degradations are in the range 13-24% for typical Linux workloads, and reach 138× in some corner cases. Energy waste is proportional. Since these bugs undermine a crucial kernel sub-system, cause substantial, sometimes massive, degradation of performance, and evade conventional testing and debugging techniques, understanding their nature and provenance is important.

[1] This occurs even though the scheduler is not explicitly configured to save power by purposefully leaving cores unused so they can be brought into a low-power state.

These bugs have different root causes, but a common symptom. ***The scheduler unintentionally and for a long time leaves cores idle while there are runnable threads waiting in runqueues.*** Short-term occurrences of this condition are acceptable: the system may temporarily enter such a state when, for instance, a thread exits or blocks or when a thread is created or becomes unblocked. Long-term occurrences are not an expected behavior. The Linux scheduler is work-conserving, meaning that it should never leave cores idle if there is work to do. Long-term presence of this symptom is, therefore, unintentional: it is due to bugs and it hurts performance.

We provide fixes to these bugs, and observe substantial performance improvements. Synchronization-intensive applications experienced many-fold speedups; one barrier-heavy scientific application ended up running 138 times faster.[2] Kernel `make` and a TPC-H workload on a widely used commercial DBMS improved performance by 13% and 14% respectively. The TPC-H query most affected by the bug sped up by 23%.

Detecting these 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. With the TPC-H workload, for example, 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.

We initially suspected scheduler bugs when we observed unexplained performance in the TPC-H database workload, which we were evaluating for a different project. Conventional tools were unhelpful to us in either confirming the bugs or understanding their root causes. To that end, we designed two new tools. The first tool, which we call a *sanity checker*, periodically checks for the violation of the aforementioned invariant, catches the bugs on a live system and collects a trace with relevant information for offline analysis. The second tool visualizes traces of scheduling activity to expedite debugging. These tools were easy to port between kernel versions (from Linux 3.17 through 4.3), ran with negligible overhead and consistently detected invariant violations. Keeping them in the standard tool belt can help reduce future occurrence of this class of bugs.

The rest of the paper is organized as follows. Section 2 describes the architecture of the Linux scheduler. Section 3 introduces the bugs we discovered, analyzes their root causes and reports their effect on performance. Section 4 presents the tools. In Section 5 we reflect on the lessons learned as a result of this study and identify open research problems. Section 6 discusses related work and Section 7 summarizes our findings.

[2] As we explain later in the paper, scheduling shortcomings exacerbated lock contention.

2. The Linux Scheduler

We first describe how Linux's Completely Fair Scheduling (CFS) algorithm works on a single-core single-user system (Section 2.1). From this perspective, the algorithm is quite simple. Then, in (Section 2.2) we explain how limitations of modern multicore systems force developers to work-around potential performance bottlenecks, which results in a substantially more complex and bug-prone implementation.

2.1 On a single-CPU system, CFS is very simple

Linux's CFS is an implementation of the weighted fair queueing (WFQ) scheduling algorithm, wherein the available CPU cycles are divided among threads in proportion to their weights. To support this abstraction, CFS (like most other CPU schedulers) time-slices the CPU among the running threads. The key decisions made in the scheduler are: *how to determine a thread's timeslice?* and *how to pick the next thread to run?*

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* (runtime of the thread divided by its weight). Once a thread's *vruntime* exceeds its assigned timeslice, the thread is pre-empted from the CPU if there are other runnable threads available. A thread might also get pre-empted if another thread with a smaller *vruntime* is awoken.

Threads are organized in a *runqueue*, implemented as a red-black tree, in which the threads are sorted in the increasing order of their *vruntime*. When a CPU looks for a new thread to run it picks the leftmost node in the red-black tree, which contains the thread with the smallest *vruntime*.

2.2 On multi-core systems, CFS becomes quite complex

In multicore environments the implementation of the scheduler becomes substantially more complex. Scalability concerns dictate using per-core runqueues. The motivation for per-core runqueues is that upon a context switch the core would access only its local runqueue, when it looks for a thread to run. Context switches are on a critical path, so they must be fast. Accessing only a core-local queue prevents the scheduler from making potentially expensive synchronized accesses, which would be required if it accessed a globally shared runqueue.

However, in order for the scheduling algorithm to still work correctly and efficiently in the presence of per-core

runqueues, the runqueues must be kept balanced. Consider a dual-core system with two runqueues that are not balanced. Suppose that one queue has one low-priority thread and another has ten high-priority threads. If each core looked for work only in its local runqueue, then high-priority threads would get a lot less CPU time than the low-priority thread, which is not what we want. We could have each core check not only its runqueue but also the queues of other cores, but this would defeat the purpose of per-core runqueues. Therefore, what Linux and most other schedulers do is periodically run a load-balancing algorithm that will keep the queues roughly balanced.

> "I suspect that making the scheduler use per-CPU queues together with some inter-CPU load balancing logic is probably _trivial_. Patches already exist, and I don't feel that people can screw up the few hundred lines too badly."
>
> Linus Torvalds, 2001 [43]

Conceptually, load balancing is simple. In 2001, CPUs were mostly single-core and commodity server systems typically had only a handful of processors. It was, therefore, difficult to foresee that on modern multicore systems load balancing would become challenging. Load balancing is an expensive procedure on today's systems, both computation-wise, because it requires iterating over dozens of runqueues, and communication-wise, because it involves modifying remotely cached data structures, causing extremely expensive cache misses and synchronization. As a result, the scheduler goes to great lengths to avoid executing the load-balancing procedure often. At the same time, not executing it often enough may leave runqueues unbalanced. When that happens, cores might become idle when there is work to do, which hurts performance. So in addition to periodic load-balancing, the scheduler also invokes "emergency" load balancing when a core becomes idle, and implements some load-balancing logic upon placement of newly created or newly awoken threads. These mechanisms should, in theory, ensure that the cores are kept busy if there is work to do.

We next describe how load balancing works, first explaining the algorithm and then the optimizations that the scheduler employs to maintain low overhead and to save power. Later we show that some of these optimizations make the code more complex and cause bugs.

2.2.1 The load balancing algorithm

Crucial for understanding the load balancing algorithm is the metric that the CFS scheduler uses to track load. We begin by explaining the metric and then describe the actual algorithm.

The load tracking metric. A strawman load-balancing algorithm would simply ensure that each runqueue has roughly the same number of threads. However, this is not necessarily what we want. Consider a scenario with two runqueues, where one queue has some number of high-priority threads and another queue has the same number of low-priority threads. Then high-priority threads would get the same amount of CPU time as low-priority threads. That is not what we want. One idea, then, is to balance the queues based on threads' weights, not their number.

Unfortunately, balancing the load based solely on thread weights is not sufficient either. Consider a scenario with ten threads in two runqueues: one thread is of high priority and nine threads are of low priority. Let us assume that the weight of the high-priority thread is nine times higher than those of the low-priority threads. With the load balanced according to threads' weights, one runqueue would contain the high-priority thread, while the other would contain the nine low-priority threads. The high-priority thread would get nine times more CPU than the low-priority threads, which appears to be what we want. However, suppose that the high-priority thread often sleeps for short periods of time, so the first core often goes idle. This core would have to frequently steal work from the other core's runqueue to keep itself busy. However, we do not want work stealing to become the common case, because this defeats the purpose of per-core runqueues. What we really want is to balance the runqueues in a smarter way, accounting for the fact that the high priority thread does not need a whole core.

To achieve this goal, CFS balances runqueues not just based on weights, but based on a metric called *load*, which is the combination of the thread's weight and its average CPU utilization. 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. Consider a scenario where we have one process with lots of threads, and another process with few threads. Then the threads of the first process combined will receive a lot more CPU time than the threads of the second process. As a result, the first process would use most of the CPU cycles and starve the other process. This would be unfair. So as of version 2.6.38 Linux added a group scheduling feature to bring fairness between groups of threads (`cgroup` feature). When a thread belongs to a `cgroup`, its load is further divided by the total number of threads in its `cgroup`. This feature was later extended to automatically assign processes that belong to different `ttys` to different `cgroups` (`autogroup` feature).

The load-balancing algorithm. A basic load balancing algorithm would compare the load of all cores and then transfer tasks from the most loaded core to the least loaded core. Unfortunately this would result in threads being migrated across the machine without considering cache locality or NUMA. Instead, the load balancer uses a hierarchical strategy.

The cores are logically organized in a hierarchy, at the bottom of which is a single core. How the cores are grouped in the next levels of the hierarchy depends on how they

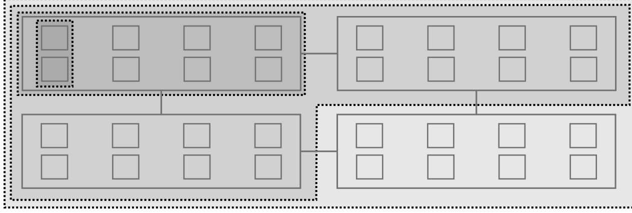


Figure 1: A machine with 32 cores, four nodes (eight cores per node) and SMT-level sharing among pairs of cores. The four grey areas represent the scheduling domains relative to the first core of the machine. Note that at the second level of the hierarchy we have a group of three nodes. That is because these three nodes are reachable from the first core in one hop. At the 4th level, we have all nodes of the machines because all nodes are reachable in 2 hops. Figure 4 shows the connectivity between the nodes on our system.

share the machine's physical resources. In the example provided here we describe the hierarchy on our experimental machine (see Table 5), where pairs of cores share functional units (e.g., FPU), and groups of eight cores share a last-level cache (LLC). A group of eight cores sharing an LLC form a NUMA node. Different NUMA nodes have varying connectivity as explained below and as shown in Figure 4. Consequently, on our target system, at the second level of the hierarchy we have pairs of cores, and at the next level we have groups of eight cores, each sharing an LLC (e.g., a NUMA node). NUMA nodes are further grouped according to their level of connectivity [23]. Nodes that are one hop apart from each other will be at the next level, and so on. An example of such a hierarchy is shown in Figure 1. Each level of the hierarchy is called a ***scheduling domain***.

The load balancing algorithm is summarized in Algorithm 1. Load balancing is run for each scheduling domain, starting from the bottom to the top. At each level, one core of each domain is responsible for balancing the load. This core is either the first idle core of the scheduling domain if the domain has idle cores whose free CPU cycles can be used for load balancing, or the first core of the scheduling domain otherwise (Lines 2–9). Following this, the average load is computed for each scheduling group of the scheduling domain (Line 10), and the *busiest* group is picked, based on heuristics that favor overloaded and imbalanced groups (Line 10). If the busiest group's load is lower than the local group's load, the load is considered balanced at this level (Line 16). Otherwise, the load is balanced between the local CPU and the busiest CPU of the group, with a tweak to ensure that load balancing works even in the presence of tasksets (Lines 18–23).

Assume, for the time being, that this algorithm is run by all cores in every load-balancing period; in the next section we will explain that, as an optimization, not all cores actually do. A core executing the algorithm begins at the second-to-lowest level of the hierarchy and balances the load one level below. For example, on the system in Figure 1 the core will begin at the pair-of-cores level and will balance the load between the two individual cores contained therein. Then, it will proceed to the level of the NUMA node, balancing the load one level below (among pairs of cores, in this case), but **not between individual cores of the NUMA node**. In a scheduling domain, the sets of cores among which the load balancing is performed are called ***scheduling groups***. At the NUMA node domain, there will be four scheduling groups, each corresponding to a pair of cores. The core will find the busiest scheduling group other than its own and will steal tasks from the busiest core in that group.

Algorithm 1 Simplified load balancing algorithm.

```
    {Function running on each cpu cur_cpu:}
 1: for all sd in sched_domains of cur_cpu do
 2:   if sd has idle cores then
 3:     first_cpu = 1st idle CPU of sd
 4:   else
 5:     first_cpu = 1st CPU of sd
 6:   end if
 7:   if cur_cpu ≠ first_cpu then
 8:     continue
 9:   end if
10:   for all sched_group sg in sd do
11:     sg.load = average loads of CPUs in sg
12:   end for
13:   busiest = overloaded sg with the highest load
        (or, if inexistent) imbalanced sg with highest load
        (or, if inexistent) sg with highest load
14:   local = sg containing cur_cpu
15:   if busiest.load ≤ local.load then
16:     continue
17:   end if
18:   busiest_cpu = pick busiest cpu of sg
19:   try to balance load between busiest_cpu and cur_cpu
20:   if load cannot be balanced due to tasksets then
21:     exclude busiest_cpu, goto line 18
22:   end if
23: end for
```

2.2.2 Optimizations

The scheduler prevents duplicating work by running the load-balancing algorithm only on the *designated core* for the given scheduling domain. When each active core receives a periodic clock tick and begins running the load-balancing algorithm, it checks whether it is the lowest-numbered core in the domain (if all cores are busy), or if it is the lowest-numbered idle core (if any core is idle). This is shown in Line 2 of the Algorithm. If this condition holds, the core deems itself as designated and continues to run the algorithm.

Power-related optimizations may further reduce the frequency of load balancing on an idle core. Originally, idle cores were always awoken on every clock tick; at this point they would run the load-balancing algorithm. However, since version 2.6.21 Linux has the option (now enabled by default) to avoid periodically waking up sleeping cores: they enter a *tickless idle* state, in which they can reduce their energy use. The only way for a core in this state to get work when another core is overloaded is to be awoken by another core. To this end, on each scheduling tick, if a core deems itself "overloaded", it checks whether there have been tickless idle cores in the system for some time, and if so, it wakes up the first tickless idle core and assigns it the role of *NOHZ balancer*. The NOHZ balancer core is responsible, on each tick, to run the periodic load balancing routine for itself and on behalf of all tickless idle cores.

On top of periodic load balancing, the scheduler also balances load when waking up threads. When a thread wakes up, after sleeping or waiting for a resource (e.g., locks, I/O), the scheduler tries to place it on the idlest core. Special rules apply when the thread is awoken by another thread (waker thread). In that case the scheduler will favour cores sharing a cache with the waker thread to improve cache reuse.

3. Bugs

> "Nobody actually creates perfect code the first time around, except me. But there's only one of me."
>
> Linus Torvalds, 2007 [44]

With so many rules about when the load balancing does or does not occur, it becomes difficult to reason about how long an idle core would remain idle if there is work to do and how long a task might stay in a runqueue waiting for its turn to run when there are idle cores in the system. Since there are very few developers who "create the perfect code the first time around", this complexity leads to bugs. Understanding the bugs is necessary to appreciate why they evade conventional testing and debugging tools. Therefore, we describe the bugs first and delay the presentation of the tools that we used to confirm and understand them until Section 4. Table 4 summarizes the bugs described in this section.

3.1 The Group Imbalance bug

The bug. We encountered this bug on a multi-user machine which we used to perform kernel compilation and data analysis using the `R` machine learning package. We suspected that this system, a 64-core, eight-node NUMA server, did not use all available cores for highly-threaded computations, instead crowding all threads on a few nodes. We illustrate this bug with the output from our visual tool, shown on Figures 2a and 2b.

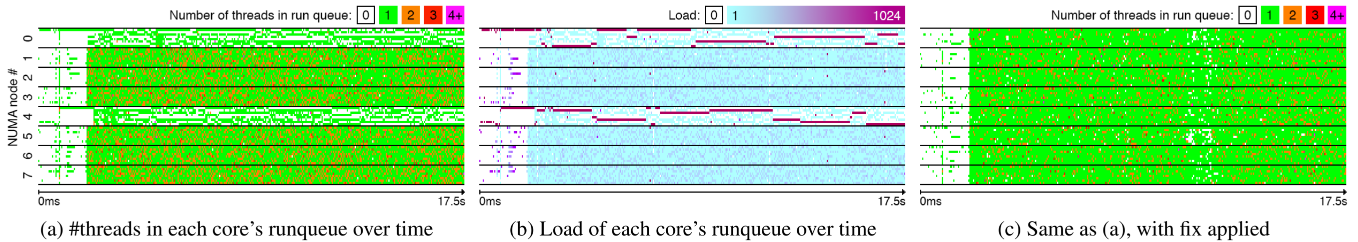
In the time period shown in the figure, the machine was executing a compilation of the kernel (`make` with 64 threads), and running two `R` processes (each with one thread). The `make` and the two `R` processes were launched from 3 different `ssh` connections (i.e., 3 different `ttys`). Figure 2a is a heatmap colour-coding the number of threads in each core's runqueue over time. The warmer the colour, the more threads a core hosts; white corresponds to an idle core. The chart shows that there are two nodes 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 runqueue.

After investigation, we found that the scheduler is not balancing load because of (i) the complexity of the load-tracking metric, and (ii) the hierarchical design. Let us first focus on the load. Remember that a threads' load is a combination of its weight and how much CPU it needs. With `autogroups`, the thread's load is also divided by the number of threads in the parent `autogroup`. In our case, 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 runqueue: a darker colour 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 runqueues based on load; obviously the underloaded cores on Nodes 0 and 4 should not steal from the overloaded core on their own node, because that core runs only a single thread. However, they must be able to steal from the more loaded cores on other nodes. Why is this not the case?

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 (line 11 of Algorithm 1). If the average load of the victim scheduling group is greater than that of its own, it will attempt to steal from that group; otherwise it will not. This is the exact reason why in our situation the underloaded cores fail to steal from the overloaded cores on other nodes. They observe that the average load of the victim node's scheduling group is not any greater than their own. The core trying to steal work runs on the same node as the high-load `R` thread; that thread skews up the average load for that node and conceals the fact that some cores are actually idle. At the same time, cores on the victim node, with roughly the same average load, have lots of waiting threads.

A valid question to ask is whether work stealing *should* occur in this case, since theoretically we want threads with a higher load to get more CPU time than threads with a lower load. The answer to that question is "yes": the Linux CFS scheduler is work-conserving in nature, so threads may get more than their fair share of CPU cycles if there are idle



(a) #threads in each core's runqueue over time (b) Load of each core's runqueue over time (c) Same as (a), with fix applied

Figure 2: The Group Imbalance bug. Y-axis shows CPU cores. Nodes are numbered 0-7. Each node contains eight cores.

cores in the system; in other words, idle cores should be always given to waiting threads. As we have seen in the scenario illustrated here, this is not necessarily the case.

The fix. To fix the 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 (lines 11 and 15 of Algorithm 1). The minimum load is the load of the least loaded core in that group. 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 *all* cores in the other group, and thus a core in the first group must steal from the second group. This algorithm ensures that no core of the second group will remain overloaded while a core of the first group has a smaller load, thus balancing the load across cores. Note that this fix works, because load is also balanced *inside* the groups (because of load balancing calls at lower levels of the hierarchy). Just as the original algorithm, we use the special cases of group imbalance (line 13 of Algorithm 1) to deal with corner cases due to tasksets. These modifications add no algorithmic complexity to the scheduler as computing the minimum load of a scheduling group and its average have the same cost. In our experience, this fix does not result in an increased number of migrations between scheduling groups (ping-pong effect).

Impact on performance. Figure 2c is a trace of the execution of that workload with the fix applied (showing a heatmap of runqueue sizes, in the same fashion as Figure 2a). After we fixed the bug, the completion time of the `make` job, in the `make/R` workload described earlier in this section, decreased by 13%. The completion time of the two `R` processes did not change. Performance impact could be much higher in other circumstances. For example, in a workload running `lu` from the NAS benchmark[3] with 60 threads, and four single threaded `R` processes, `lu` ran 13× 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.

[3] The NAS benchmark comprises a set of small programs designed to help evaluate the performance of supercomputers [32].

3.2 The Scheduling Group Construction bug

The 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 4, 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. Since threads are created on the same node as their parent thread, the net effect is that the pinned application runs only on one node, no matter how many threads it has.

The bug is due to 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. We explain with an example.

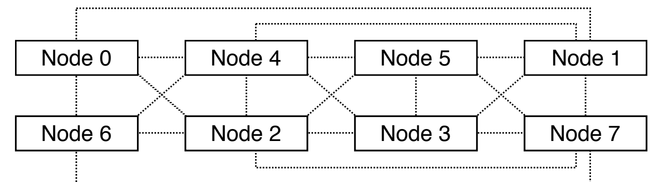


Figure 4: *Topology of our 8-node AMD Bulldozer machine*

In our machine, shown in Figure 4, the first scheduling group contains the cores of Node 0, plus the cores of all the nodes that are one hop apart from Node 0, namely Nodes 1, 2, 4 and 6. The second scheduling group contains the cores of the first node not included into the first group (Node 3), plus cores of all nodes that are one hop apart from Node 3: Nodes 1, 2, 4, 5, 7. The first two scheduling groups are thus:

$$\{0, 1, 2, 4, 6\}, \{1, 2, 3, 4, 5, 7\}$$

Note that Nodes 1 and 2 are included in both scheduling groups. Further note that these two nodes are actually two hops apart from one another. If the scheduling groups were constructed from the perspective of Node 1, Node 1 and 2

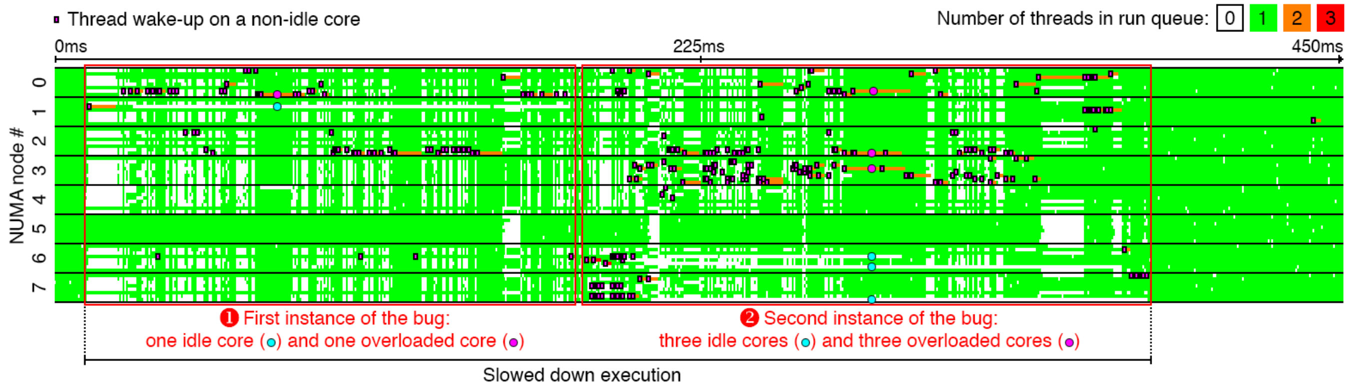


Figure 3: Several instances of the Overload-on-Wakeup bug.

would not be together in all groups. Let us see what this implies for load balancing.

Suppose that an application is pinned on Nodes 1 and 2 and that all of its threads are being *created* on Node 1 (Linux spawns threads on the same core as their parent thread; when an application spawns multiple threads during its initialization phase, they are likely to be created on the same core – so this is what typically happens). Eventually we would like the load to be balanced between Nodes 1 and 2. However, when a core on 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 1 hop apart, nodes 2 hops apart, etc.) were introduced to increase the likelihood for threads to remain close to their original NUMA node.

The fix. We modified the construction of scheduling groups so that each core uses scheduling groups constructed from its perspective. After the fix, when cores of Node 1 and 2 try to steal tasks at the level of the machine, Nodes 1 and 2 are no longer included in all scheduling groups. The cores are thus able to detect an imbalance and to steal work.

Impact on performance. Table 1 presents the performance difference in NAS 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 27× is experienced by `lu`. The slowdown is a lot more than the expected 2× because of locking effects. NAS applications use spinlocks and spin-barriers; when all threads execute on the same node due to the taskset bug, the thread that executes the critical section may be descheduled in favour of a thread that will waste its timeslice by spinning. `lu` is an extreme example of this: it uses a pipeline algorithm to parallelize work; threads wait for the data processed by other threads. When multiple threads are forced to execute on the same core, this results in large performance losses [22].

Application	Time w/ bug (sec)	Time w/o bug (sec)	Speedup factor (×)
bt	99	56	1.75
cg	42	15	2.73
ep	73	36	2
ft	96	50	1.92
is	271	202	1.33
lu	1040	38	27
mg	49	24	2.03
sp	31	14	2.23
ua	206	56	3.63

Table 1: Execution time of NAS applications with the Scheduling Group Construction bug and without the bug. All applications are launched using `numactl --cpunodebind=1,2 <app>`.

3.3 The Overload-on-Wakeup bug

The bug. The gist of this bug is that a thread that was asleep may wake up on an overloaded core while other cores in the system are idle. The bug was introduced by an optimization in the wakeup code (`select_task_rq_fair` function). When a thread goes to sleep on Node X and the thread that wakes it up later is running on that same node, the scheduler only considers the cores of Node X for scheduling the awakened thread. If all cores of Node X are busy, the thread will wake up on an already busy core and miss opportunities to use idle cores on other nodes. This can lead to a significant under-utilization of the machine, especially on workloads where threads frequently wait.

The rationale behind this optimization is to maximize cache reuse. Essentially, the scheduler attempts to place the woken up thread physically close to the waker thread, e.g., so

both run on cores sharing a last-level cache, in consideration of producer-consumer workloads where the woken up thread will consume the data produced by the waker thread. This seems like a reasonable idea, but for some workloads waiting in the runqueue for the sake of better cache reuse does not pay off.

This bug was triggered by a widely used commercial database configured with 64 worker threads (1 thread per core) and executing the TPC-H workload. This workload, in combination with transient short-lived threads from other applications, triggers both the Group Imbalance bug[4] and the Overload-on-Wakeup bug. Since we already described the Group Imbalance bug in Section 3.1, we disabled `autogroups` in this experiment in order to better illustrate the Overload-on-Wakeup bug.

Figure 3 illustrates several instances of the wakeup bug. During the first time period (noted ❶), 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 (noted ❷), 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 database thread. This occurs when the kernel launches tasks that last less than a millisecond to perform background operations, such as logging or irq handling. 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 the threads, which often sleep and wake up, keep waking up on Node B, even if there are no idle cores on that node. This occurs, because the wakeup code only considers cores from the local node for the sake of better cache reuse.

We now understand how a thread might wake up on a loaded core despite the presence of idle cores in the system. Note in Figure 3 that the system eventually recovers from the load imbalance: the load balancing algorithm finally migrates threads from overloaded cores to idle cores. The question is, why does it take several milliseconds (or even seconds) to recover?

Note that there are two kinds of idle cores in the system: *short-term* and *long-term*. Short-term idle cores go idle for short periods, because the database thread running on that core intermittently sleeps due to a synchronization or I/O event. Ideally we want the load balancing events to migrate a thread from an overloaded core to a long-term idle core. Migrating to a short-term idle core is of little help: a thread that used to run on that core will shortly awaken, and as we have seen, the scheduler may place it on another loaded core in the same node due to the cache locality optimization.[5] The imbalance will thus persist.

Unfortunately, when the scheduler considers where to migrate a thread from the overloaded core, it makes no distinction between short-term and long-term idle cores. All it does is to check whether the core is idle at the moment when the load-balancer is invoked. Remember from Section 2.2.1 that the load balancing algorithm is invoked at different levels of the hierarchy by the "designated core". If there are multiple idle cores eligible to be "designated", only one gets chosen. If we are lucky, the long-term idle core gets chosen, and the balance gets restored. This is exactly what happens in Figure 3, when the system eventually recovers from the imbalance. However, as we observed, pure luck is not enough to maintain optimal performance.

The fix. 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.

Waking up the thread on a long-term idle core may have implications for power consumption. Cores that have been idle for a long time usually go into a low-power mode. Waking up a thread on that core will force the core to exit that mode and run at full power. For that reason, we only enforce the new wakeup strategy if the system's power management policy does not allow cores to enter low-power states at all. Furthermore, our fix only matters for workloads where threads frequently go to sleep and awaken and where the system is intermittently oversubscribed (there are more threads than cores). In these situations, waking up threads on long-term idle cores makes sense. In other situations, since thread wakeups are rare, our fix does not significantly alter the behavior of the scheduler.

Looking for a long-term idle core in the system adds no overhead to the wakeup function: the kernel already maintains a list of all idle cores in the system, so picking the first one (this is the one that has been idle the longest) takes constant time.

Impact on performance. This bug has a major impact on the database TPC-H workload, because the threads often wait for each other, which means that any two threads that

[4] The commercial database we are using relies on pools of worker threads: a handful of container processes each provide several dozens of worker threads. Each container process is launched in a different `autogroup`, which means that worker threads also belong to different `autogroups`. Since different container processes have a different number of worker threads, different worker threads have different loads. The bug occurs for the same reasons as explained in Section 3.1.

[5] Actually, thread migrations to short-term idle cores explain why the extra threads are not always located on the same cores in Figure 3.

Bug fixes	TPC-H request #18	Full TPC-H benchmark
None	55.9s	542.9s
Group Imbalance	48.6s (−13.1%)	513.8s (−5.4%)
Overload-on-Wakeup	43.5s (−22.2%)	471.1s (−13.2%)
Both	43.3s (−22.6%)	465.6s (−14.2%)

Table 2: Impact of the bug fixes for the Overload-on-Wakeup and Group Imbalance bugs on a popular commercial database (values averaged over five runs).

are stuck on the same core end up slowing down *all* the remaining threads. This effect is visible in Figure 3: during ❶ and ❷: many threads have gaps in their execution, i.e., they all sleep at the same time, waiting for "straggler" threads that are sharing a core. When all instances of the bug are resolved (rightmost part of the graph), the gaps disappear.

Table 2 shows the performance impact of our bug fixes for the Overload-on-Wakeup and Group Imbalance bugs on the commercial database. We use two workloads: (1) the 18th query of TPC-H, which is one of the queries that is most sensitive to the bug, and (2) the full TPC-H benchmark. Our bug fix for the Overload-on-Wakeup improves performance by 22.2% on the 18th query of TPC-H, and by 13.2% on the full TPC-H workload. Using the Group Imbalance bug fix in addition to the Overload-on-Wakeup bug fix improves performance by 22.6% on the 18th query of TPC-H, and by 14.2% on the full TPC-H benchmark. The effect of the Group Imbalance bug fix is small in these experiments, because the Group Imbalance bug only occurs at the level of a pair of cores (i.e., one core is idle while another one is overloaded), and only during some parts of the experiments.

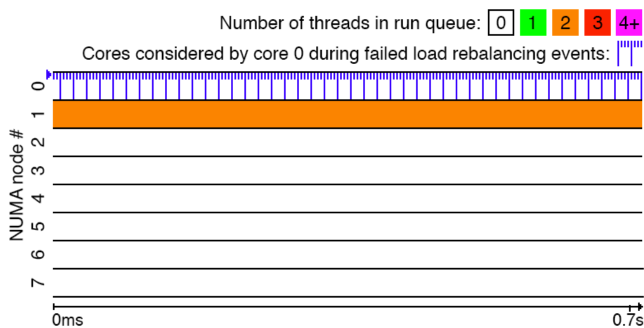


Figure 5: The Missing Scheduling Domains bug, from the point of view of Core 0. The vertical blue lines represent the cores considered by Core 0 for each (failed) load balancing call. There is one load balancing call every 4ms. We can see that Core 0 only considers its sibling core and cores on the same node for load balancing, even though cores of Node 1 are overloaded.

3.4 The Missing Scheduling Domains bug

The bug. When a core is disabled and then re-enabled using the `/proc` interface, load balancing between any NUMA nodes is no longer performed. The bug is due to an incorrect update of a global variable representing the number of scheduling domains in the machine. When a core is disabled, this variable is set to the number of domains inside a NUMA node. As a consequence, the main scheduling loop (line 1 of Algorithm 1) exits earlier than expected.

As a result, threads can only run on the node on which they ran before the core had been disabled (even if the node they run on is not the same as that on which the core was disabled and then re-enabled). For processes created after disabling the core, all threads will run on the same node as their parent process. Since all processes are usually created from the same "root" process (e.g., `sshd` daemon and the `ssh` processes it spawns), this bug usually results in all newly created threads executing on only one node of the machine, regardless of the number of threads.

Figure 5 shows a visualization of this bug. An application with 16 threads is launched on the machine. After the threads have been created, all cores on Node 1 run two threads (orange horizontal lines). The blue vertical lines originating from Core 0 represent the cores considered by Core 0 when it tries to steal work. Because the loop exits earlier than it should, Core 0 only considers cores local to its node, and not cores of Node 1.

The fix. We traced the root cause of the bug to the code that regenerates the machine's scheduling domains. Linux regenerates scheduling domains every time a core is disabled. Regenerating the scheduling domains is a two-step process: the kernel regenerates domains inside NUMA nodes, and then across NUMA nodes. Unfortunately, the call to the function generating domains across NUMA nodes was dropped by Linux developers during code refactoring. We added it back, and doing so fixed the bug.

Application	Time w/ bug (sec)	Time w/o bug (sec)	Speedup factor (×)
`bt`	122	23	5.24
`cg`	134	5.4	24.90
`ep`	72	18	4.0
`ft`	110	14	7.69
`is`	283	53	5.36
`lu`	2196	16	137.59
`mg`	81	9	9.03
`sp`	109	12	9.06
`ua`	906	14	64.27

Table 3: Execution time (in seconds) of NAS applications with the Missing Scheduling Domains bug and without it.

Impact on performance. Table 3 summarizes the impact of the Missing Scheduling Domains bug on several NAS

applications, after disabling and reenabling one core in the system. Applications are launched with 64 threads, the default configuration on our 64-core machine. The Missing Scheduling Domains bug causes all threads of the applications to run on a single node instead of eight. In some cases, the performance impact is greater than the 8× slowdown that one would expect, given that the threads are getting 8× less CPU time than they would without the bug (they run on one node instead of eight). `lu`, for example, runs 138× faster! Super-linear slowdowns occur in cases where threads frequently synchronize using locks or barriers: if threads spin on a lock held by a descheduled thread, they will waste even more CPU time, causing cascading effects on the entire application's performance. Some applications do not scale ideally to 64 cores and are thus a bit less impacted by the bug. The minimum slowdown is 4×.

CPUs	8 × 8-core Opteron 6272 CPUs (64 threads in total)
Clock frequency	2.1 GHz
Caches (each CPU)	768 KB L1 cache, 16 MB L2 cache, 12 MB L3 cache
Memory	512 GB of 1.6 Ghz DDR-3
Interconnect	HyperTransport 3.0 (see Figure 4)

Table 5: Hardware of our AMD Bulldozer machine.

3.5 Discussion

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 it would be a long-term solution, 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 major redesigns. The original scheduler had high algorithmic complexity, which resulted in poor performance 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 successful but soon required modifications for new architectures like NUMA and SMT. 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 n) but it was deemed worthwhile to provide the desired features.

As the hardware and workloads became more complex, CFS too succumbed to bugs. The addition of `autogroups` coupled with the hierarchical load balancing introduced the Group Imbalance bug. Asymmetry in new, increasingly more complex NUMA systems triggered the Scheduling Group Construction bug. "NUMA-ness" of modern systems was responsible for the Missing Scheduling Domains 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. However, 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 implementation of the load metric. This change is reported to be "done in a way that significantly reduces complexity of the code" [5]. Simplifying the load metric could get rid of the Group Imbalance bug, which is directly related to it. However, we confirmed, using our tools, that the bug is still present.[6]

Kernel developers rely on mutual code review and testing to prevent the introduction of bugs. This could potentially be effective for bugs like the Missing Scheduling Domains and Scheduling Group Construction 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, they 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.

In summary, conventional testing techniques and debugging tools were not helpful to us in either confirming the bugs, after we initially suspected them, or understanding their root causes. Our experience motivated us to build new tools, using which we could productively confirm the bugs and understand why they occur. The following section describes the tools.

4. Tools

4.1 Online Sanity Checker

Sanity checker is a term we use to describe a mechanism for periodically checking an invariant that must be maintained

[6] All our fixes will be submitted to the kernel developers shortly.

Name	Description	Kernel version	Impacted applications	Maximum measured performance impact
Group Imbalance	When launching multiple applications with different thread counts, some CPUs are idle while other CPUs are overloaded.	2.6.38+	All	13×
Scheduling Group Construction	No load balancing between nodes that are 2-hops apart	3.9+	All	27×
Overload-on-Wakeup	Threads wake up on overloaded cores while some other cores are idle.	2.6.32+	Applications that sleep or wait	22%
Missing Scheduling Domains	The load is not balanced between NUMA nodes	3.19+	All	138×

Table 4: Bugs found in the scheduler using our tools.

by the software. The invariant verified by our sanity checker is shown in Algorithm 2. It verifies that no core is idle while another core's runqueue has waiting threads. We strived to keep the code simple, perhaps at the expense of a higher algorithmic complexity, to minimize the chance of bugs in the sanity checker itself.

Algorithm 2 "No core remains idle while another core is overloaded"

```
1:  for all CPU1 in CPUs do
2:    if CPU1.nr_running ≥ 1 {CPU1 is not idle} then
3:      continue
4:    end if
5:    for all CPU2 in CPUs do
6:      if CPU2.nr_running ≥ 2 and can_steal(CPU1,
        CPU2) then
7:        Start monitoring thread operations
8:      end if
9:    end for
10: end for
```

Our sanity checker is different from an assertion or a watchdog in that, in our case, it must be specifically tailored to check for conditions that are acceptable for a short period of time, but unacceptable if they persist. While an assert would fire as soon as the desired invariant is violated, a sanity checker must minimize the probability of flagging short-term transient violations, and catch long-term violations with a high probability.

To meet this requirement we implement the sanity checker as follows. The invariant check is invoked periodically at an interval S. If the invariant violation is detected, we actively monitor additional scheduler events (described below) for a short time period M to see if this is a "legal" short-term violation that is promptly fixed by the scheduler. If it is not fixed, we record profiling information (described below) and flag a bug.

We set S and M to minimize the overhead and the probability of false positives and to maximize the probability of detecting the actual bugs. Our setting for S is one second; this helps ensure that the invariant violation is detected for all but very short programs and keeps the overhead low. We measured the overhead to be under 0.5% on our system for workloads of as many as 10,000 threads when $S = 1$.

The load balancer runs every 4ms, but because of the hierarchical design multiple load balancing attempts might be needed to recover from invariant violation in a bug-free system. We conservatively set M to 100ms to virtually eliminate the probability of false positives. The monitoring that occurs during that period tracks thread migration, creation and destruction, because it is these events that can help the system to recover. Tracking these events requires adding a function call in the `move_thread`, `fork` and `exit` functions that check where threads are migrated. The overhead of this function call is negligible.

The probability of detecting the actual bugs, i.e., long-term invariant violations, depends on the frequency and duration of the invariant violation. In most cases we described, once the bug triggers an invariant violation, the system never recovers. Such invariant violations are trivially detected by the sanity checker. In one case (the Overload-on-Wakeup bug), invariant violations persisted for shorter periods, on the order of hundreds of milliseconds, then disappeared and reappeared again. In this case, the probability of catching the violation depends on the total fraction of time that the system spends in the undesirable state. If the fraction is small, the chances of detecting the bug are also small, but so is the impact on performance. Longer-lasting and/or more frequently occurring violations are detected with a higher probability. Furthermore, if the bug-triggering workload keeps running, the chances that the sanity checker detects the bug during at least one of the checks keep increasing.

If the bug is detected, the sanity checker begins gathering profiling information to include in the bug report. We use `systemtap` to profile calls to all load balancing functions (e.g., `load_balance`, `select_task_rq_fair`) along with all the statements executed by these functions and the values of the variables they use. We used these profiles to understand how the load-balancing functions were executed and why they failed to balance the load. Note that we do not

record the part of the execution where the bug is triggered, but this is not an issue because we only want to understand why all load balancing calls fail during a significant period of time. Monitoring with `systemtap` has a high overhead (around 7% in our measurements), so we only begin profiling *after* detecting the bug and stop profiling after 20ms.

Thanks to its small size, the sanity checker is easily portable between kernel versions, and does not conflict with patches that modify the scheduler. We originally implemented the sanity checker in Linux 3.17. Porting to 4.3 only required changing one line of code (the `do_posix_clock_monotonic_gettime` function, that we used to get timing data, changed its name).

4.2 Scheduler Visualization tool

The visualization tool we are about to describe was tremendously helpful in gauging the nature of bug symptoms and further understanding their root causes. The tool illustrates salient scheduling activity over time (the charts from this tool were shown to illustrate the bugs in the previous section). These charts rely on additional instrumentation of the kernel, whose overhead is negligible. Our visual tool makes it possible to profile and to plot (1) the size of run queues, (2) the total load of run queues, and (3) the cores that were considered during periodic load balancing and thread wakeups. In order to provide maximum accuracy, it does not use sampling, instead, it records every change in the size of run queues or load, as well as a set of considered cores at each load rebalancing or thread wakeup event. To keep the overhead low, we store all profiling information in a large global array in memory of a static size. Each element of this array is an *event* that corresponds to either (1), (2), or (3):

- For (1), we instrument kernel functions `add_nr_running` and `sub_nr_running`, which are the only functions that directly alter the variables that stores the size of each run queue. In these functions, we store an event in our global array that contains a timestamp, the core number, and the new runqueue size.
- Similarly, for (2), we instrument kernel functions `account_entity_enqueue` and `account_entity_enqueue`, which are the only functions that directly alter the variables that store the load of each run queue. In these functions, we store an event in our global array that contains a timestamp, the core number, and the new load.
- Finally, for (3), we instrument kernel functions `select_idle_sibling`, `update_sg_lb_stats`, `find_busiest_queue` and `find_idlest_group`. In these functions, we store an event in our global array that contains a timestamp, as well as a bit field with 0's for cores that were not considered during the operation, and 1's for cores that were.

Implementing these changes in the Linux kernel took less than 150 lines of code. In order to write an event, a thread uses an atomic increment instruction to find the position in the array where to write the event; it then writes its event to memory which may incur a cache miss. On our architecture, 20 bytes are sufficient to store each event. In Section 3, the application that produced events at the highest frequency was the commercial database: it produces around 60,200 events of type (1) per second, 58,000 events of type (2) per second, and 68,000 events of type (3) per second, for a total of around 186,200 events per second. Consequently, when active our profiler uses 3.6 MB of RAM per second on a machine with 64 cores. In practice the profiler is only active when a bug is detected, and its impact on performance negligible in the non-buggy case.

In addition to the changes in the Linux kernel, we also wrote a kernel module that makes it possible to start and end a profiling session on demand, and to output the global array to a file. We also wrote scripts that plot the results. Figures 2, 4, and 5 are examples of these plots.

In our experience, confirming and understanding the tricky performance bugs described in this paper, and detecting them across kernel versions was substantially more productive after we developed these tools. Since the tools catch important bugs, while at the same time being simple, virtually overhead-free and easy to port across kernel versions, we believe that it is a good idea to keep them as part of a standard kernel developers' tool belt.

5. Lessons Learned

We now reflect on the lessons learned during this study and identify open problems.

The bugs we described resulted from developers wanting to put more and more optimizations into 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.

As of circa 2000, dozens of papers 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 multithreaded CPU pipelines [8, 9, 24, 29, 34, 42, 46]. There were algorithms that reduced communication distance among threads sharing data [41] and determined the optimal number of cores to allocate to multithreaded workloads [17]. There were algorithms that addressed the scheduling of threads on asymmetric multicore CPUs [22, 35] and algorithms that integrated scheduling with the management of power and temperature [19]. Finally, there were algorithms that managed the memory on

NUMA systems, which was closely tied to how threads are being scheduled [7, 14, 18, 38], and algorithms that scheduled threads to minimize communication latency on systems with an asymmetric interconnect [23]. 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 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. What we need is to rethink the architecture of the scheduler, since it can no longer remain a small, compact and largely isolated part of the kernel. We now understand that rapid evolution of hardware that we are witnessing today will motivate more and more scheduler optimizations. The scheduler must be able to easily integrate them, and to have a way of reasoning about how to combine them. We envision a scheduler that is a collection of modules: the core module and optimization modules. The core module embodies the very basic function of the scheduler: assigning runnable threads to idle cores and sharing the cycles among them in some fair fashion. The optimization modules suggest specific enhancements to the basic algorithm. For example, a load-balancing module might suggest an alternative load balancing schedule that avoids excessive overhead. A cache affinity module might suggest waking up a thread on a core where it recently ran. A resource contention module might suggest a placement of threads that reduces the chances of contention-induced performance degradation. The core module should be able to take suggestions from optimization modules and to act on them whenever feasible, while always maintaining the basic invariants, such as not letting cores sit idle while there are runnable threads. Deciding how to combine multiple optimizations when they conflict, e.g., if a cache affinity module and a resource contention module suggest different thread placements or if a load balancer risks to break memory-node affinity as it moves threads among runqueues, is a difficult open problem in scheduling.

Another lesson we learned as a result of this work is the crucial importance of visualization tools. Understanding the root causes of the bugs we described would not have been possible without visualization of the execution events relevant to the problem. Visualizing execution is definitely not a new idea. While isolated clusters of system developers (e.g., certain engineering teams at Google [26, 40]) have embraced it, the developer community at large is missing effective visualization tools. There are many great trace-gathering tools, such as `systemtap`, Pivot tracing [27], or Intel Processor Tracing. However, without effective visualization, developers tend to only summarize and aggregate the information available in the traces, which obscures outliers that are responsible for bugs and performance problems (think of the Overload-on-Wakeup bug!). Though visualization is conceptually simple, implementing effective tools is not easy. Execution traces can be very large, so visualizing everything is not feasible and not useful. Understanding what to visualize and how to let the user effectively explore the trace when the user does not know what they are looking for is an important open problem. There most certainly exists research on this topic in the visualization community, but the systems community needs to more actively embrace those techniques and adapt them to the specificity of the problems we solve in systems.

6. Related Work

Performance bugs. Performance bugs are a recurring problem for operating systems. The Linux Kernel Performance project [13] was started in 2005 to combat performance regressions. Despite this effort, performance bugs are repeatedly reported by OS researchers. Boyd et al. [10] reported a variety of scalability bugs, and Harji et al. [20] reported other kinds performance bugs in 2.6 kernels.

Mollison et al. [30] proposed a tool that performs regression testing specifically for schedulers. It targets real-time schedulers implemented as plug-ins for the LInux Testbed for MUltiprocessor Scheduling in Real-Time Systems (LITMUS^{RT}) project. The tool is limited (e.g., it does not take blocking into account) and does not target the Linux kernel's fair scheduling policy (CFS).

To combat performance bugs, Perl et al. [33] proposed to add assertions to kernels to check that function calls finish in a timely manner. Shen et al. [39] built a throughput model for checking the performance of I/Os. These techniques detect performance bugs using assertions, which is not applicable for detecting a problematic load imbalance in the scheduler. Temporary load imbalances are expected and not problematic. Therefore, our online sanity checker uses a different design to discriminate between problematic and non-problematic invariant violations.

Kernel correctness. A large body of work has also been done on verifying system correctness. RacerX [15] and Eraser [37] detect deadlocks and race conditions. Erickson et al. [16] detect data races in kernel modules. It would be ideal to extend these systems to target performance bugs, which do not necessarily result in systems crash; however, this is a difficult problem because short-term invariant violations are acceptable in our environment.

D3S [25], Likely Invariants [36], and ExpressOS [28] use invariants and predicates to check correctness of simple operations (e.g., ensure that a variable is within a certain range). Our online sanity checker makes it possible to detect more subtle incorrect states of the system (i.e., systems with cores that are idle for a long time while other cores are overloaded), and it could be integrated into operating systems to validate higher-level design choices.

Model checking has also been used to detect bugs in kernels. CMC [31] inject states in the kernel to find implementation bugs. Yang et al. [45] found errors in file systems (e.g. deadlocks) using model checking. Model checking is a useful approach to find bugs before they happen. The scheduler is particularly challenging to model check due to the large number of possible workloads and intricacies with hardware behavior (e.g., topology, behavior of sleeping cores). Model checkers could be used to find crash bugs in the scheduler, but to the best of our knowledge, none of these works could be used to detect more subtle performance bugs.

Formal verification is a method for proving software correctness by analyzing the source code. Traditionally, formal verification was limited to small codebases and languages other than C, but recently, through heroic efforts of several OS researchers, formal verification was applied to OS kernels [11, 12, 21]. Even so, these state-of-the-art methods do not apply to the bugs described here, because they are limited to single-threaded environments and do not have a way of reasoning about time. Formal tools work by describing the system as series of state transitions, pre-conditions and post-conditions, and then reason whether any state transitions may lead to violation of post-conditions given the possible pre-conditions. The problem is that in our environment, short and intermittent violations of post-conditions (i.e., idle cores in the presence of waiting threads) are totally acceptable. It is the long-term violations that are problematic. Unfortunately, existing tools do not have the mechanisms allowing to reason how timing affects transient violation of invariants. Extending these tools to work in multithreaded environments and to reason about time could make them more suitable, but having Linux developers write formal specifications will be another hurdle for adoption of these tools.

Tracing. Due to the complexity of some of the bugs we encountered, the Overload-on-Wakeup bug in particular, profiling tools were not sufficient to fully understand their causes: we needed traces in order to precisely follow the behavior of the scheduler.

Many tracing tools have been proposed to help detect performance bugs. Event Tracing for Windows [2] and DTrace [1] are frameworks that make it possible to trace application and kernel events, including some scheduler events, for the Microsoft Windows, Solaris, MacOS X and FreeBSD operating systems. The Linux kernel comes with its own set of tracers, such as Ftrace [3] and SystemTap [6]. Some additional tools such as KernelShark [4] produce graphical traces.

Our online sanity checker does not provide a novel way to monitor events, and directly relies on SystemTap for this purpose. What our sanity checker does that is not included in these tools, however, is detecting an invariant violation in order to only monitor events while a performance bug is occurring: running existing tracing tools all the time would be of little use to detect performance bugs, since they would produce large amounts of data that would overwhelmingly consist of non-buggy execution traces, with no way to jump to the buggy sections of these traces. Moreover, our visualization tool is capable of monitoring very fine-grained scheduler events that were crucial to understanding some of the performance bugs presented in this paper, such as individual scheduling events (which makes it possible to plot which scheduling event was responsible each time a thread moves across cores), or individual iterations in the work-stealing loop (which makes it possible to monitor which cores are considered during load balancing).

7. Conclusion

Scheduling, as in *dividing CPU cycles among threads* was thought to be a solved problem. We show that this is not the case. Catering to complexities of modern hardware, a simple scheduling policy resulted in a very complex bug-prone implementation. We discovered that the Linux scheduler violates a basic work-conserving invariant: scheduling waiting threads onto idle cores. As a result, runnable threads may be stuck in runqueues for seconds while there are idle cores in the system; application performance may degrade manyfold. The nature of these bugs makes it difficult to detect them with conventional tools. We fix these bugs, understand their root causes and present tools, which make catching and fixing these bugs substantially easier. Our fixes and tools will be available at `http://git.io/vaGOW`.

References

[1] DTrace. `http://dtrace.org/`.

[2] Event Tracing for Windows. `https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx`.

[3] Ftrace. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`.

[4] KernelShark. `http://people.redhat.com/srostedt/kernelshark/HTML/`.

[5] Linux 4.3 scheduler change "potentially affects every SMP workload in existence". `http://www.phoronix.com/scan.php?page=news_item&px=Linux-4.3-Scheduler-SMP`.

[6] SystemTap. `https://sourceware.org/systemtap/`.

[7] J. Antony, P. P. Janes, and A. P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, 2006.

[8] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, 2010.

[9] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC'11, 2011.

[10] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[11] 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 *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[12] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015.

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

[14] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[15] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[16] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[17] J. R. Funston, K. El Maghraoui, J. Jann, P. Pattnaik, and A. Fedorova. An SMT-selection metric to improve multithreaded applications' performance. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, 2012.

[18] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma. Large pages may be harmful on NUMA systems. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC'14, 2014.

[19] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, 2004.

[20] A. S. Harji, P. A. Buhr, and T. Brecht. Our troubles with Linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, 2011.

[21] 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 *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[22] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[23] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on NUMA systems: asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, 2015.

[24] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, 2007.

[25] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, 2008.

[26] D. Luu. The Nyquist theorem and limitations of sampling profilers today, with glimpses of tracing tools from the future. `http://danluu.com/perf-tracing`.

[27] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, 2015.

[28] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[29] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[30] M. S. Mollison, B. Brandenburg, and J. H. Anderson. Towards unit testing real-time schedulers in $\text{LITMUS}^{\text{RT}}$. In *Proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, OSPERT '09, 2009.

[31] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI), Dec. 2002.

[32] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.

[33] S. E. Perl and W. E. Weihl. Performance assertion checking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, 1993.

[34] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V. Kalogeraki. FACT: a framework for adaptive contention-aware thread migrations. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, 2011.

[35] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[36] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4), Nov. 1997.

[38] L. T. Schermerhorn. A matter of hygiene: automatic page migration for Linux. 2007. URL `https://linux.org.au/conf/2007/talk/197.html`.

[39] K. Shen, M. Zhong, and C. Li. I/O system performance debugging using model-driven anomaly characterization. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 23–23, 2005.

[40] D. Sites. Data center computers: modern challenges in CPU design. `https://www.youtube.com/watch?v=QBu2Ae8-8LM`.

[41] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, 2007.

[42] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009.

[43] L. Torvalds. *The Linux Kernel Mailing List*. `http://tech-insider.org/linux/research/2001/1215.html`, Feb. 2001.

[44] L. Torvalds. *Tech Talk: Linus Torvalds on git*, Google. `http://www.youtube.com/watch?v=4XpnKHJAok8`, Mar. 2007.

[45] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4), Nov. 2006.

[46] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.