An Analysis of Performance Evolution of Linux's Core Operations

Xiang (Jenny) Ren
University of Toronto
jenny.ren@mail.utoronto.ca

Kirk Rodrigues
University of Toronto
kirk.rodrigues@mail.utoronto.ca

Luyuan Chen
University of Toronto
luyuan.chen@mail.utoronto.ca

Camilo Vega
University of Toronto
camilo.vega@mail.utoronto.ca

Michael Stumm
University of Toronto
stumm@ece.utoronto.ca

Ding Yuan
University of Toronto
yuan@ece.utoronto.ca

Abstract

This paper presents an analysis of how Linux's performance has evolved over the past seven years. Unlike recent works that focus on OS performance in terms of scalability or service of a particular workload, this study goes back to basics: the latency of core kernel operations (*e.g.*, system calls, context switching, *etc.*). To our surprise, the study shows that the performance of many core operations has worsened or fluctuated significantly over the years. For example, the `select` system call is 100% slower than it was just two years ago. An in-depth analysis shows that over the past seven years, core kernel subsystems have been forced to accommodate an increasing number of *security enhancements* and *new features*. These additions steadily add overhead to core kernel operations but also frequently introduce extreme slowdowns of more than 100%. In addition, simple *misconfigurations* have also severely impacted kernel performance. Overall, we find most of the slowdowns can be attributed to 11 changes.

Some forms of slowdown are avoidable with more proactive engineering. We show that it is possible to patch two security enhancements (from the 11 changes) to eliminate most of their overheads. In fact, several features have been introduced to the kernel unoptimized or insufficiently tested and then improved or disabled long after their release.

Our findings also highlight both the feasibility and importance for Linux users to actively configure their systems to achieve an optimal balance between performance, functionality, and security: we discover that 8 out of the 11 changes can be avoided by reconfiguring the kernel, and the other 3 can be disabled through simple patches. By disabling the 11 changes with the goal of optimizing performance, we speed up Redis, Apache, and Nginx benchmark workloads by as much as 56%, 33%, and 34%, respectively.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10.
https://doi.org/10.1145/3341301.3359640

CCS Concepts • **Software and its engineering** → **Software performance**; **Operating systems**; • **Social and professional topics** → *History of software.*

Keywords Performance evolution, operating systems, Linux

ACM Reference Format:
Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. 2019. An Analysis of Performance Evolution of Linux's Core Operations. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359640

1 Introduction

In the early days of operating systems (OS) research, the performance of core OS kernel operations – in particular, system call latency – was put under the microscope [5, 11, 37, 47, 56]. However, over the past decade or two, interest in core kernel performance has waned. Researchers have seemingly shifted focus to other aspects of OS performance such as multicore scalability [10], performance under specific workloads or on new hardware, and scheduling [45], to name just a few. Indeed, the most recent comprehensive analysis of OS system call performance dates back to 1996, when McVoy and Staelin [47] studied OS system call latencies using lmbench with follow-up work from Brown and Seltzer [11] in 1997 that extended lmbench. This begs the question: are core OS kernel operations getting slower or faster?

This paper presents an analysis of how the latencies of Linux's core operations have evolved over the past seven years. We use the term "kernel operations" to encompass both system calls and kernel functions like context switching. This work first introduces LEBench, a microbenchmark suite that measures the performance of the 13 kernel operations that most significantly impact a variety of popular applications. We test LEBench on 36 Linux release versions, from 3.0 to 4.20 (the most recent), running on a single Intel Xeon server. Figure 1 shows the results. All kernel operations are slower than they were four years ago (version 4.0), except for `big-write` and `big-munmap`. The majority (75%) of the kernel operations are slower than seven years ago (version 3.0). Many of the slowdowns are substantial: the majority

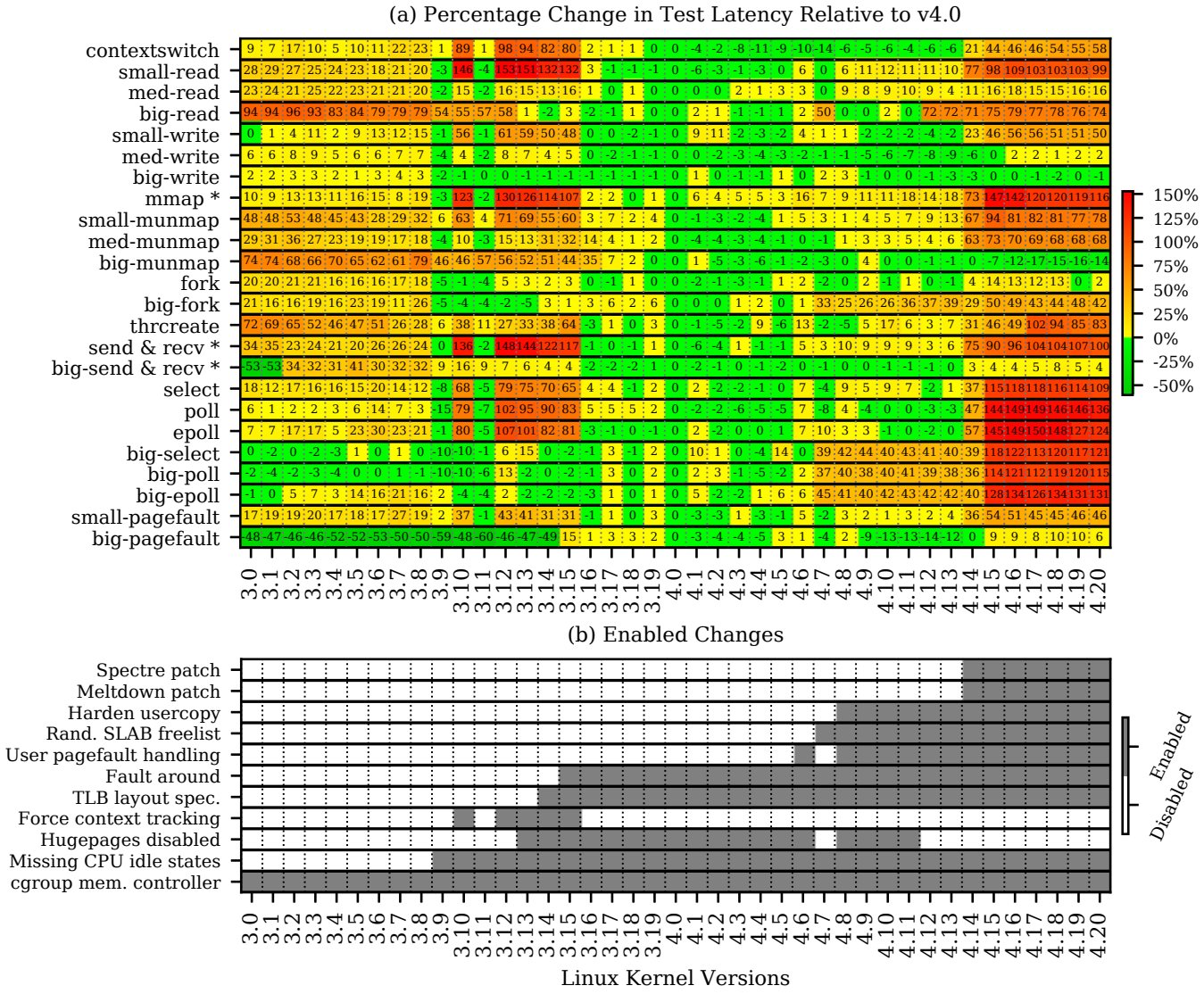

Figure 1. Main result. (a) shows the latency trend for each test across all kernels, relative to the 4.0 kernel. (We use the 4.0 kernel as a baseline to better highlight performance degradations in later kernels.) (b) shows the timeline of each performance affecting change. Each value in (a) indicates the percentage change in latency of a test relative to the same test on the 4.0 kernel. Therefore, positive and negative values indicate worse and better performance, respectively. *: for brevity, we show the averaged trend of related tests with extremely similar trends, including the average of all `mmap` tests, the `send` and `recv` test, and the `big-send` and `big-recv` test.

(67%) slow down by at least 50% and some by 100% over the last seven years (*e.g.*, `mmap`, `poll` & `select`, `send` & `recv`). Performance has also fluctuated significantly over the years.

Drilling down on these performance fluctuations, we observe that a total of 11 root causes are responsible for the major slowdowns. These root causes fall into three categories. First, we observe a growing number of *(1) security enhancements* and *(2) new features*, like support for containers and virtualization, being added to the kernel. The effect of this trend on kernel performance manifests itself in two ways: a steady creep of slowdown in core operations, and disruptive slowdowns that persist over many versions (*e.g.*, a more than 100% slowdown that persists across six versions). Such significant impacts are introduced by security enhancements and features, which often demand complex and intrusive modifications to central subsystems of the kernel, such as memory management. The last category of root causes is *(3) configuration changes*, some of which are simple misconfigurations that resulted in severe slowdowns across kernel operations, impacting many users.

While many forms of slowdowns result from fundamental trade-offs between performance and functionality or security, we find a good number could have been avoided or significantly alleviated with more proactive software engineering practices. For example, frequent lightweight testing can easily catch the simple misconfigurations that resulted in widespread slowdowns. The performance of certain kernel functions would also benefit from more eager optimizations and thorough testing: we found some features significantly degraded the performance of core kernel operations in the initial release; only long after having been introduced were they performance-optimized or disabled due to performance complaints. Furthermore, a few other changes that introduced performance slowdowns simply remained unoptimized—we patched two of the security enhancements to eliminate most of their performance overhead without reducing security guarantees. At the same time, we recognize the difficulty of testing and maintaining a generic OS kernel like Linux, which must support a diverse array of hardware and workloads [27], and evolves extremely quickly [52]. On the other hand, the benefit of being a generic OS kernel is that Linux is highly configurable—8 out of the 11 root causes can be easily disabled by reconfiguring the kernel. This creates the potential for Linux users to actively configure their kernels and significantly improve the performance of their custom workloads.

Out of the many performance-critical parts of the kernel, we chose to study core kernel operations since the significance of their performance is likely elevating; recent advances in fast non-volatile memory and network devices together with the flattened curve of microprocessor speed scaling may shift the bottleneck to core kernel operations. We also chose to focus on how the kernel's software design and implementation impact performance. Prior studies on OS performance mostly focused on comparing the implications of different architectures [5, 12, 56, 64]. Those studies occurred during a time of diverse and fast-changing CPUs, but such CPU architectural heterogeneity has largely disappeared in today's server market. Therefore, we focus on software changes to core OS operations introduced over time, making this the first work to systematically perform a longitudinal study on the performance of core OS operations.

This paper makes the following contributions. The first is a thorough analysis of the performance evolution of Linux's core kernel operations and the root causes for significant performance trends. We also show that it is possible to mitigate the performance overhead of two of the security enhancements. Our second contribution is LEBench, a microbenchmark that is collected from representative workloads together with a regression testing framework capable of evaluating the performance of an array of Linux versions. The benchmark suite and a framework for automatically testing multiple kernel versions are available at https://github.com/LinuxPerfStudy/LEBench. Finally, we evaluate the impact of the 11 identified root causes on three real-world applications and show that they can cause slowdowns as high as 56%, 33%, and 34% on the Redis key-value store, Apache HTTP server, and Nginx web server, respectively.

Application	Workload	% System Time
Apache Spark v2.2.1	spark-bench's minimal example	3%
Redis v4.0.8	redis-benchmark with 100K requests	41%
PostgreSQL v9.5	pgbench with scale factor 100	17%
Chromium browser v59.0.3071.109	Watching a video and reading a news article	29%
Build toolchain (make 4.1, gcc 5.3)	Compiling the 4.15.10 Linux kernel	7%

Table 1. Applications and respective workloads used to choose core kernel operations, and each workload's approximate execution time spent in the kernel.

The rest of the paper is organized as follows. §2 describes LEBench and the methodology we used to drive our analysis. We summarize our main findings in §3 before zooming into each change that caused significant performance fluctuations in §4. §5 discusses the performance implications of core kernel operations on three real-world applications. §6 validates LEBench's results on a different hardware setup. We discuss the challenges of Linux performance tuning in §7, and we survey related work in §9 before concluding.

2 Methodology

Our experiments focus on system calls, thread creation, page faults, and context switching. To determine which system calls are frequently exercised, we use our best efforts to select a set of representative application workloads. Table 1 lists the applications and the workloads we ran. We include workloads from three popular server-side applications: Spark, a distributed computing framework, Redis, a key-value store, and PostgreSQL, a relational database. In addition, we include an interactive user workload—web browsing through the Chromium browser—and a software development workload—building the Linux kernel. The chosen workloads exercise the kernel with varying intensities, as shown in Table 1.

We used `strace` to measure CPU time and the call-frequency of each system call used by the workloads. We then selected those system calls which took up the most time across all workloads. `wait`-related system calls were excluded as their sole purpose is to block the process. Table 2 lists each of the microbenchmarks. Where applicable, we vary the input sizes to account for a variety of usage patterns.

Our process for running each microbenchmark is as follows. Latency is measured by collecting a timestamp immediately before and after invoking a kernel operation. For system calls, the benchmark bypasses the libc wrapper whenever possible to expose the true kernel performance. We repeat

Test Name	Description
Context switch	Forces context switching by having two processes repeatedly pass one byte through two pipes.
Thread create	Measure the time immediately before and after the thread creation, both in the child and the parent. The shorter latency of the two is used to eliminate variations introduced by scheduling.
`fork`	Measure the time immediately before and after the fork, both in the child and the parent. The shorter latency of the two is used. To stress test fork, 12,000 writable pages are mapped into the parent before forking; to understand the minimum forking overhead, 0 pages are mapped.
`read`, `write`	Sequentially read or write the entire content of a file. A one page file is used to understand the bare minimum overhead. Sizes of 10 and 10,000 pages are used to test how performance changes with increasing sizes. For read tests, the page cache is warmed up by running the tests before taking measurements.
`mmap`, `munmap`	Map a number of consecutive file-backed read-only pages from a file into memory, or unmap a number of consecutive file-backed writable pages into a file. We use three file sizes: 1, 10, and 10,000 pages.
Page fault	Reads one byte from the first page of a number of newly mapped pages to trigger page faults. The test is run for 1 and 10,000 contiguous, mapped file-backed pages. The size of the mapped region affects the behavior of page fault handling under the "fault-around" patch.
`send`, `recv`	Creates a TCP connection between two processes on the same machine using a UNIX socket as the underlying communication channel. Each process repeatedly sends/receives a message to/from the other. The test is run for two message sizes: 1 byte and 96,000 bytes.
`select`, `poll`, `epoll`	Performs `select`, `poll`, or `epoll` on a number of socket file descriptors. The socket file descriptors become ready upon having enough memory for each socket. The test is run for 10 and 1,000 file descriptors.

Table 2. A description of the tests in LEBench including their usage patterns. (The size of a page is 4kB in this table.)

each measurement 10,000 times and report the value calculated using the K-best method with K set to 5 and tolerance set to 5% [59]. To do this, we order all measured values numerically, and select the lowest from the first series of five values where no two adjacent values differ by more than 5%. Selecting lower values filters the interference from background workloads, and setting K to 5 and tolerance to 5% is considered effective in ensuring consistent and accurate results across runs [59].

We run the microbenchmarks on each major version of Linux released in the past seven years. This includes versions 3.0 to 3.19 and versions 4.0 to 4.20. For every major version, we select the latest minor version (the y in v.x.y) released *before* the next major version. This is to avoid testing changes that were backported from a subsequent major version. For example, for major version 3.0, we tested minor version 3.0.7 (released just before the release of 3.1.0) since 3.0.8 may contain some changes that were introduced in 3.1.0. We only tested versions that were released. Linux distributions such as Ubuntu [68] or Arch Linux [33] typically configure the kernel differently from Linux's default configuration. We use Ubuntu's Linux distribution because, at least for web-servers, Ubuntu is the most widely used Linux distribution [70]. For example, Netflix hosts its services on Ubuntu kernels [4].

We carried out the tests on an HP DL160 G9 server with a 2.40GHz Intel Xeon E5-2630 v3 processor, 512KB L1 cache, 2MB L2 cache, and 20MB L3 cache. The server also has 128GB of 1866MHz DDR4 memory and a 960GB SSD for persistent storage. To understand how different hardware setups affect the results, we repeated the tests on a Lenovo laptop with an Intel i7 processor and analyze the differences between the two sets of results in §6.

When interpreting results from the microbenchmarks, we treat a flat latency trend as expected and analyze any increase or decrease that may signify a performance regression or improvement, respectively. We extract the causes of these performance changes iteratively: for each test, we first identify the root cause of the most significant performance change; we then disable the root cause and repeat the process to identify the root cause of the next most significant performance change. We repeat this until the difference between the slowest and fastest kernel versions is no more than 10% for the target test.

3 Overview of Results

We overview the results of our analysis in this section and make a few key observations before detailing each root cause in §4.

Figure 1 displays the latency evolution of each test across all kernels, relative to the 4.0 kernel. Only isolated tests experience performance improvements over time; the majority of tests display worsening performance trends and frequently suffer prolonged episodes of severe performance degradation. These episodes result in significant performance fluctuations across multiple core kernel operations. For example, `send` and `recv`'s performance degraded by 135% from version 3.9 to 3.10, improved by 137% in 3.11, and then degraded again by 150% in 3.12. We also observe that the benchmark's overall performance degraded by 55% going from version 4.13 to 4.15. The sudden and significant nature of these performance

Root Cause	Description	How it affects performance	Impact
Security Enhancements: max combined slowdown: 146% `poll`			
Kernel page-table isolation (KPTI) (§4.1.1)	Removes kernel memory mappings from the page table upon entering userspace to mitigate Meltdown.	A kernel entry/exit now swaps the page table pointer, which further leads to subsequent TLB misses.	`recv` 63%, `small-read` 60%
Avoid indirect branch speculation (§4.1.2)	Mitigates Spectre (v2) by avoiding speculations on indirect jumps and calls.	Adds around 30 cycles to each indirect jump or call.	`poll` 89%, `recv` 25%
SLAB freelist randomization (§4.1.3)	Randomizes the order of objects in the SLAB freelist.	Destroys spatial locality that leads to increased L3 cache misses.	`big-select` 45%
Hardened usercopy (§4.1.4)	Adds sanity checks on data-copy operations between userspace and the kernel.	Constant cost for system calls that copy data to and from userspace.	`send` 18%, `select` 18%
New Features: max combined slowdown: 167% `big-pagefault`			
Fault around (§4.2.1)	Pre-establishes mappings for pages surrounding a faulting page, if they are available in the file cache.	Adds constant overhead for page faults on read-only file-backed pages.	`big-pagefault` 167%
Control group memory controller (§4.2.2)	Accounts and limits memory usage per control group.	Adds overhead when establishing or destroying page mappings.	`big-munmap` 81%
Disabling transparent huge pages (§4.2.3)	Disables the transparent use of 2MB pages.	More page faults when sequentially accessing large memory regions.	`big-read` 83%
Userspace page fault handling (§4.2.4)	Enables userspace to provide the mapping for page faults in an address range.	Slows down `fork`, which checks each copied memory area for userspace mappings.	`big-fork` 12%
Configuration Changes: max combined slowdown: 171% `small-read`			
Forced context tracking (§4.3.1)	Misconfiguration forces unnecessary CPU time accounting and RCU handling on every kernel entry and exit.	Adds overhead on each kernel entry and exit.	`small-read` 171%, `recv` 149%
TLB layout specification (§4.3.2)	Hardcoded & outdated TLB capacity in older kernels causes `munmap` to flush the TLB when it should invalidate individual TLB entries.	Increases TLB misses due to flushes.	50% read after `munmap`
Missing CPU power-saving states (§4.3.3)	Missing power-saving states in older kernels results in decreased effective frequency.	Slows down CPU bound tests.	`select` 31%, `send` 26%

Table 3. Summary of root causes causing performance fluctuations across kernel versions. For each root cause, we report examples of significant slowdowns from highly impacted tests across all kernel versions.

degradations suggests they are caused by intrusive changes to the kernel.

We identified 11 kernel changes that explain the significant performance fluctuations as well as more steady sources of overhead. These are categorized and summarized in Table 3, and their impact on LEBench's performance is overviewed in Figure 2. The 11 changes fall into three categories: *security enhancements* (4/11), *new features* (4/11), and *configuration changes* (3/11).

Overall, Linux users are paying a hefty performance tax for security enhancements. The cost of accommodating security enhancements is high because many of them demand significant changes to the kernel. For example, the mitigation for Meltdown (§4.1.1) requires maintaining a separate page table for userspace and kernel execution, fundamentally modifying some of the core designs of memory management. Similarly, SLAB freelist randomization (§4.1.3) alters dynamic memory allocation behaviours in the kernel.

Interestingly, several security features introduce overhead by attempting to defend against untrusted code in the kernel itself. For example, the hardened usercopy feature (§4.1.4) is used to defend against bugs in kernel code that might copy too much data between userspace and the kernel. However, we note that it can be redundant with other kernel code that already carefully validates pointers. Similarly, SLAB freelist randomization (§4.1.3) attempts to protect against buffer overflow attacks that exploit buggy kernel code. However, the randomization introduces overhead for all uses of the

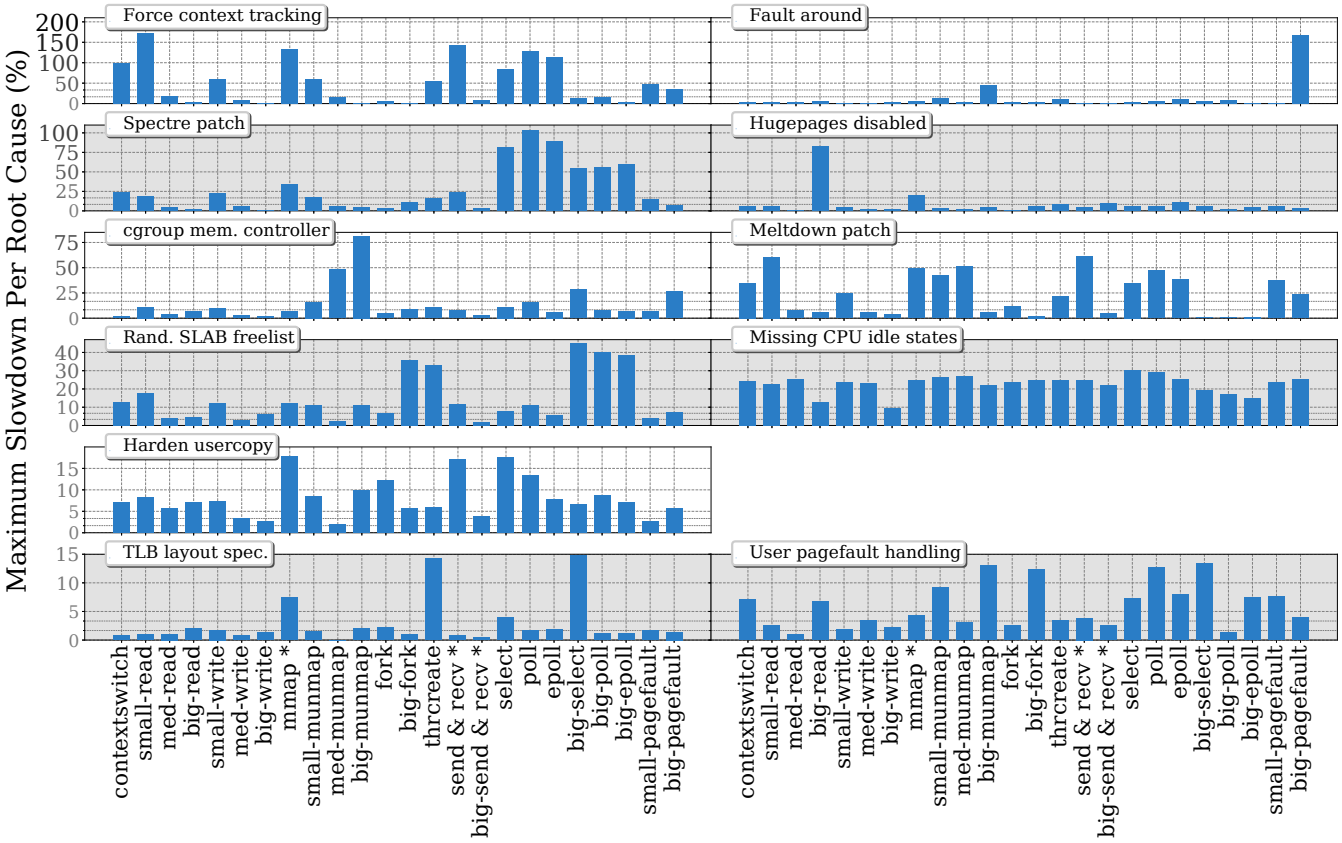

Figure 2. Impact of the 11 identified root causes on the performance of LEBench tests. For every root cause, we display the maximum slowdown across all kernels for each test. Note that the Y-axis scales are different for each row of subgraphs: Root causes with highest possible impacts on LEBench are ordered first.

SLAB freelist, including correct kernel code. This suggests a trust issue that is fundamentally rooted in the monolithic kernel design [8].

Similar to the security enhancements, supporting many new features demands complex and intricate changes to the core kernel logic. For example, the control group memory controller feature (§4.2.2), which supports containerization, requires tracking every page allocation and deallocation; in an early unoptimized version, it slowed down the `big-pagefault` and `big-munmap` tests by as much as 26% and 81% respectively.

While the complexity of certain features may increase the difficulty of performance optimization. Simple misconfigurations have also significantly impacted kernel performance. For example, mistakenly turning on forced context tracking (§4.3.1) caused all the benchmark tests to slowdown by an average of 50%.

Two aforementioned changes (forced context tracking and control group memory controller) were significantly optimized or disabled entirely *reactively*, *i.e.*, only after performance degradations were observed in released kernels, instead of *proactively*. Forced context tracking (§4.3.1) was only disabled after plaguing five versions for more than 11 months, and has become a well-known cause of performance troubles for Linux users [46, 48, 57]; control group memory controller (§4.2.2) remained unoptimized for 6.5 years, and continues to cause significant performance degradation in real workloads [49, 69]. Both cases are clearly captured by LEBench, suggesting that more frequent and thorough testing, as well as more proactive performance optimizations, would have avoided these impacts on users.

As another example where Linux performance would benefit from more proactive optimization, we were able to easily optimize two other security enhancements, namely avoiding indirect jump speculation (§4.1.2) and hardened user copy (§4.1.4), largely eliminating their slowdowns without sacrificing security guarantees.

Finally, with little effort, Linux users can avoid most of the performance degradation from the identified root causes by actively reconfiguring their systems. In fact, 8 out of 11 root causes can be disabled through configuration, and the other 3 can be disabled through simple patches. Users that do not require the new functionalities or security guarantees can disable them to avoid paying unnecessary performance penalties. In addition, our findings also point to the fact that Linux is shipped with static configurations that cannot adapt

to workloads with diverse characteristics. This suggests that Linux users should pay close attention to performance when their workload's characteristics change or when updating the kernel; in such scenarios, kernel misconfigurations (with respect to the workload) or Linux performance regressions could be avoided by proactive kernel reconfiguration. This practice has the potential to offer significant performance gains.

4 Performance Impacting Root Causes

This section describes the 11 root causes from Table 3. For each root cause, we first explain the background of the change before analyzing its performance impact.

4.1 Security Enhancements

Four security enhancements to the kernel resulted in significant performance slowdowns in LEBench. The first two are a response to recently discovered CPU vulnerabilities, and the last two are meant to protect against buggy kernel code.

4.1.1 Remove Kernel Mappings in Userspace

Introduced after kernel version 4.14, kernel page table isolation (KPTI) [41] is a security patch to mitigate the Meltdown vulnerability [44] that affects several current generation processor architectures, including Intel x86 [18, 22]. The average slowdown caused by KPTI across all microbenchmark tests is 22%; `recv` and `read` tests are affected the most, slowing down by 63% and 59% respectively.

Meltdown allows a userspace process to read kernel memory. When the attacker performs a read of an unauthorized address, the processor schedules both the read and the privilege check in its instruction pipeline. However, before the privilege check is complete, the value read may have already been returned from memory and loaded into the cache. Once the privilege check fails, the processor does not eliminate all side-effects of the read and the value remains in the cache. The attacker can exploit this by using a "timing-channel" to leak the value.

KPTI mitigates Meltdown by using a different page table in the kernel than in userspace. Before the patch, kernel and user mode shared the same address space using one shared page table with kernel memory protected by requiring a higher privilege level for access. However, this protection is ineffective with Meltdown. With KPTI, the kernel-space page table still contains both kernel and user mappings; whereas the userspace page table removes the vast majority of kernel mappings, leaving only the bare-minimum ones necessary to service a trap (*e.g.*, handlers for system calls and interrupts) [31].

The overhead of keeping two separate page tables is minimal. KPTI only needs to keep a separate copy of the *top-level* page table for both kernel and user page tables; all lower-level page tables in the user page table can be accessed from the kernel page table. The top-level page table contains only 512 entries and is modified very infrequently, requiring very little "synchronization" between the two copies.

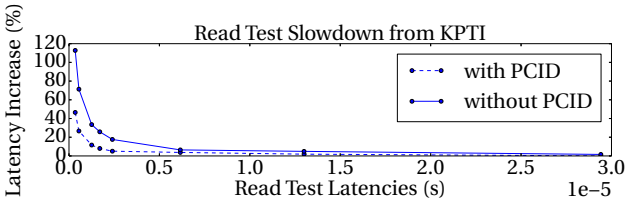

Figure 3. Slowdown of the `read` test due to KPTI, with increasing baseline latency, with and without the PCID feature enabled.

KPTI's most serious source of overhead stems from swapping the page table pointer on every kernel entry and exit, each time forcing a TLB flush. This leads to a constant cost of the two writes to the page table pointer register (`CR3` on Intel processors) and a variable cost from increased TLB misses. With KPTI, the lower-bound of the constant cost is on the order of 400–500 cycles, whereas without KPTI, the kernel entry and exit overhead is less than 100 cycles.[1] The variable cost of TLB misses depends on different workloads' memory access patterns. For example, `small-read` and `big-read` spend an additional 700 and 6000 cycles in the TLB miss handler, respectively.

The kernel developers released an optimization with the KPTI patch that avoids the TLB flush on processors with the process-context identifier (PCID) feature [23].[2] The feature allows tagging each TLB entry with a unique PCID pertaining to an address space, such that only entries with the currently active PCID are used by the CPU. The kernel developers use this feature to assign a separate PCID for the kernel and user TLB entries, hence the kernel no longer needs to flush the TLB on each entry and exit.

The performance improvement is significant. Figure 3 compares KPTI's overhead on the `read` test with and without the PCID optimization. For the shortest `read` test with a baseline latency of 344ns, the PCID optimization reduces the slowdown from 113% to 47%. The number of increased cycles in the TLB miss handler is reduced from 700 to just 30. (Figure 3 also shows that tests with short latencies are more sensitive to the overhead caused by KPTI.)

Despite the TLB flush being avoided, we find the lower-bound of the constant cost of KPTI is still 400–500 cycles. This is because the kernel still needs to write to the `CR3` register on every entry and exit, since on Intel processors, the active PCID is stored in bits that are a part of `CR3`. Writing to `CR3` is expensive, costing around 200 cycles. This is why,

[1] We measure the constant cost by comparing the result of running an empty system call with and without the KPTI patch. We measure the cycles spent in the MMU's TLB miss handler. The constant cost is estimated by subtracting the increase in cycles spent in the TLB miss handler from the overall increase in latency.

[2] The results in Figure 1 and Table 3 are obtained with PCID enabled.

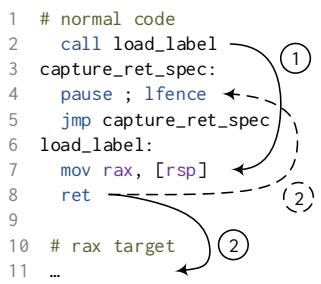

Figure 4. An example showing how Retpoline replaces `jmp [rax]`. The solid lines indicate actual execution paths, whereas the dotted line indicates a speculatively executed path.

as shown in Figure 3, the shortest `read` test still experiences a 59% slowdown with PCID-optimized KPTI. The PCID optimization alone has a minimal cost: it results in around two additional instruction TLB misses per round-trip to the kernel, compared to pre-KPTI kernels. This is because the optimization requires additional code, for example, to swap the active PCID on kernel entry and exit.

Interestingly, the PCID optimization benefits all tests with the exception of the `med-munmap` test, whose slowdown *increases* from 18% to 53% with PCID enabled. This is because `med-munmap` shoots down individual TLB entries, and the instruction to invalidate a tagged TLB entry is more expensive.

4.1.2 Avoiding Indirect Branch Speculation

Introduced in version 4.14, the Retpoline patch [67] mitigates the second variant (V2) of the Spectre attacks [35] by bypassing the processor's speculative execution of indirect branches. The patch slows down half of the tests by more than 10% and causes severe degradation to the `select`, `poll`, and `epoll` tests, resulting in an average slowdown of 66%. In particular, `poll` and `epoll` slow down by 89% and 72%, respectively.

An indirect branch is a jump or call instruction whose target is not determined statically—it is only resolved at runtime. An example is `jmp [rax]`, which jumps to an address that is stored in the `rax` register. Modern processors use the indirect branch predictor to speculatively execute the instructions at the predicted target. However, Intel and AMD processors do not completely eliminate all side effects of an incorrect speculation, *e.g.*, by leaving data in the cache as described in §4.1.1 [1, 22]. Attackers can exploit such "side-channels" by carefully polluting the indirect branch target history, hence tricking the processor into speculatively executing the desired branch target.

Retpoline mitigates Spectre v2 by replacing each indirect branch with a sequence of instructions—called a "thunk"—during compilation. Figure 4 shows the thunk that replaces `jmp [rax]`. The thunk starts with a `call`, which pushes the return address (line 4) onto the stack, before jumping to line 7. Line 7, however, replaces the return address with

fs/select.c

```
int do_select(...) {
 for (;;) {
    ...
    mask = (*f_op->poll)(f.file, wait);
 }
 ...
}
```

net/socket.c

```
const struct file_operations
          socket_file_ops = {
 .poll =   sock_poll,
 ...
};
```

Figure 5. Left: the indirect branch code snippet used by `select`, `poll`, and `epoll`. Right: assignment of the `poll` function pointer for sockets.

```
  for (;;) {
    ...
-   mask = (*f_op->poll)(f.file, wait);
+   if ((*f_op->poll) == sock_poll)
+     mask = sock_poll(f.file, wait);
+   else if ((*f_op->poll) == pipe_poll)
+     mask = pipe_poll(f.file, wait);
+   else if ((*f_op->poll) == timerfd_poll)
+     mask = timerfd_poll(f.file, wait);
+   else
+     mask = (*f_op->poll)(f.file, wait);
    ...
  }
```

Figure 6. Our patch to optimize Retpoline's overhead in `select`, `poll`, and `epoll`.

the original jump destination, stored in `rax`, by moving it onto the stack. This causes the `ret` at line 8 to jump to the original jump destination, `[rax]`, instead of line 4. Thus, the thunk achieves the same behavior as `jmp [rax]` without using indirect branches.

A careful reader would have noticed that even without lines 4 and 5, the speculative path would still fall into an infinite loop at lines 7 and 8. What makes lines 4–5 necessary is that repeatedly executing line 8, even speculatively, greatly perturbs a separate return address speculator, resulting in high overhead. In addition, the `pause` instruction at line 4 provides a hint to the CPU that the two lines are a spin-loop, allowing the CPU to optimize for power consumption [2, 24].

The slowdown caused by Retpoline is proportional to the number of indirect jumps and calls in the test. The penalty for each such instruction is similar to that of a branch misprediction. We further investigate the effects of Retpoline on the `select` test. Without Retpoline, the `select` test executes an average of 31 indirect branches, all of which are indirect `calls`; the misprediction rate of these is less than 1 in 30,000. Further analysis shows that 95% of these indirect calls are from just three program locations that use function pointers to invoke the handler of a specific resource type. Figure 5 shows one of the program locations, which is also on the critical path of `poll` and `epoll`. The `poll` function pointer is invoked repeatedly inside `select`'s main loop, and the actual target is decided by the file type (a socket, in our case).

With Retpoline, all of the 31 indirect branches executed by `select` are replaced with the thunk, and the `ret` in the thunk always causes a return address misprediction that has 30-35 cycles of penalty, resulting in a total slowdown of 68% for the test.

We alleviated the performance degradation by turning each indirect call into a switch statement, *i.e.*, a direct conditional branch, which Spectre-V2 cannot exploit. Figure 6 shows our patch on the program location shown in Figure 5. It directly invokes the specific target after matching for the type of the resource. It reduced `select`'s slowdown from 68% to 5.7%, and `big-select`'s slowdown from 55% to 2.5% respectively. This patch also reduces Retpoline's overhead on `poll` and `epoll`.

4.1.3 SLAB Freelist Randomization

Introduced since version 4.7, SLAB freelist randomization increases the difficulty of exploiting buffer overflow bugs in the kernel [66]. A SLAB is a chunk of contiguous memory for storing equally-sized objects [9, 39]. It is used by the kernel to allocate kernel objects. A group of SLABs for a particular type or size-class is called a cache. For example, `fork` uses the kernel's SLAB allocator to allocate `mm_structs` from SLABs in the `mm_struct` cache. The allocator keeps track of free spaces for objects in a SLAB using a "freelist," which is a linked list connecting adjacent object spaces in memory. As a result, objects allocated one after another will be adjacent in memory. This predictability can be exploited by an attacker to perform a buffer overflow attack. Oberheide [28] describes an example of an attack that has occurred in practice.

The SLAB freelist randomization feature randomizes the order of free spaces for objects in a SLAB's freelist such that consecutive objects in the list are not reliably adjacent in memory. During initialization, the feature generates an array of random numbers for each cache. Then for every new SLAB, the freelist is constructed in the order of the corresponding random number array.

This patch resulted in notable overhead on tests that sequentially access a large amount of memory. It caused `big-fork` to slow down by 37%, and the set of tests—`big-select`, `big-poll`, and `big-epoll`—to slow down by an average of 41%. The slowdown comes from two sources. The first is the time spent randomizing the freelist during its initialization. In particular, `big-fork` spent roughly 6% of its execution time just randomizing the freelist since it needs to allocate several SLABs for the new process. The second and more significant source of slowdown is poor locality caused by turning sequential object access patterns into random access patterns. For example, `big-fork`'s L3 cache misses increased by around 13%.

4.1.4 Hardened Usercopy

Introduced since version 4.8, the hardened usercopy patch validates kernel pointers used when copying data between userspace and the kernel [26]. Without this patch, bugs in the kernel could be exploited to either cause buffer overflow attacks when too much data is copied from userspace, or to leak data when too much is copied to userspace. This patch protects against such bugs by performing a series of sanity checks on kernel pointers during *every* copy operation. However, this adds unnecessary overhead to kernel code that already validates pointers.

For example, consider `select`, which takes a set of file descriptors for every type of event the user wants to watch for. When invoked, the kernel copies the set from userspace, modifies it to indicate which events occurred, and then copies the set back to userspace. During this operation, the kernel already checks that kernel memory was allocated correctly and only copies as many bytes as were allocated. However, the hardened usercopy patch adds several *redundant* sanity checks to this process. These include checking that *i*) the kernel pointer is not null, *ii*) the kernel region involved does not overlap the text segment, and *iii*) the object's size does not exceed the size limit of its SLAB if it is allocated using the SLAB allocator. To evaluate the cost of these redundant checks, we carefully patched the kernel to remove them.

The cost of hardened usercopy depends on the type of data being copied and the amount. For `select`, the cost of checking adds 30ns of overhead. This slows down the test by a maximum of 18%. `poll` operates similarly to `select` and also has to copy file descriptors and events to and from userspace. Interestingly, `epoll` does not experience the same degree of slowdown since it copies less data; the list of events to watch for is kept in the kernel, and only the events which have occurred are copied to userspace. In contrast, the `read` tests copy one page to userspace at a time, but the page does not belong to a SLAB. As a result, only basic checks such as checking for a valid address are performed, costing only around 5ns for each page copied. This source of overhead is not significant even for `big-read`, which copies 10,000 pages.

4.2 New Features

Next we describe the root causes that are new kernel features. One of them, namely fault around (§4.2.1), is in fact, an optimization. It improves performance for workloads with certain characteristics at the cost of others. Disabling transparent huge pages (§4.2.3) can also improve performance for certain workloads. However, these features also impose nontrivial overhead on LEBench's microbenchmarks. The other two features are new kernel functionalities mostly intended for virtualization or containerization needs.

4.2.1 Fault Around

Introduced in version 3.15, the fault around feature ("fault-around") is an optimization that aims to reduce the number of minor page faults [34]. A minor page fault occurs when no page table entry (PTE) exists for the required page, but

the page is resident in the page cache. On a page fault, fault-around not only attempts to establish the mapping for the faulting page, but also for the surrounding pages. Assuming the workload has good locality and several of the pages adjacent to the required page are resident in the page cache, fault-around will reduce the number of subsequent minor page faults. However, if these assumptions do not hold, fault-around can introduce overhead. For example, Roselli *et al.* studied several file system workloads and found that larger files tend to be accessed randomly, which renders prefetching unhelpful [63].

The `big-pagefault` test experiences a 54% slowdown as a result of fault-around. `big-pagefault` triggers a page fault by accessing a single page within a larger memory-mapped region. When handling this page fault, the fault-around feature searches the page cache for surrounding pages and establishes their mappings, leading to additional overhead.

4.2.2 Control Groups Memory Controller

Introduced in version 2.6, the control group (cgroup) memory controller records and limits the memory usage of different control groups [42]. Control groups allow a user to isolate the resource usage of different groups of processes. They are a building block of containerization technologies like Docker [16] and Linux Containers (LXC) [43]. This feature is tightly coupled with the kernel's core memory controller so it can credit every page deallocation or debit every page allocation to a certain cgroup. It introduces overhead on tests that heavily exercise the kernel memory controller, even though they do not use the cgroup feature.

The `munmap` tests experienced the most significant slowdown due to the added overhead during page deallocation. In particular, `big-munmap` and `med-munmap` experienced an 81% and 48% slowdown, respectively, in kernels earlier than version 3.17.

Interestingly, the kernel developers only began to optimize cgroup's overhead since version 3.17, 6.5 years after cgroups was first introduced [29]. During `munmap`, the memory controller needs to "uncharge" the memory usage from the cgroup. Before version 3.17, the uncharging was done once for every page that was deallocated. It also required synchronization to keep the uncharging and the actual page deallocation atomic. Since version 3.17, uncharging is batched, *i.e.*, it is done only once for all the removed mappings. It also occurs at a later stage when the mappings are invalidated from the TLB, so it no longer requires synchronization. Consequently, after kernel version 3.17, the slowdowns of `big-munmap` and `med-munmap` are reduced to 9% and 5%, respectively.

In contrast, the memory controller only adds 2.7% of overhead for the page fault tests. When handling a page fault, the memory controller first ensures that the cgroup's memory usage will stay within its limit following the page allocation, then "charges" the cgroup for the page. Here we do not see as significant a slowdown as in the case of `munmap`, because during each page fault, only one page is "charged" — the memory controller's overhead is still dwarfed by the cost of handling the page fault itself. In contrast, `munmap` often unmaps multiple pages together, aggregating the cost of the inefficient "uncharging." Note that `mmap` is generally unaffected by this change because each mmapped page is allocated on demand when it is later accessed. In addition, the `read` and `write` tests are not affected since they use pre-allocated pages from the page cache.

4.2.3 Transparent Huge Pages

Enabled from version 3.13 to 4.6, and again from 4.8 to 4.11, the transparent huge pages (THP) feature automatically adjusts the default page size [38]. It allocates 2MB pages (huge pages), and it also has a background thread that periodically promotes memory regions initially allocated with base pages (4KB) into huge pages. Under memory pressure, THP may decide to fall back to 4KB pages or free up more memory through compaction. THP can decrease the page table size and reduce the number of page faults; it also increases "TLB reach," so the number of TLB misses is reduced.

However, THP can also negatively impact performance. It could lead to internal fragmentation within huge pages. (Unlike FreeBSD [53], Linux could promote a 2MB region that has unallocated base pages into using a huge page [36]). Furthermore, the background thread can also introduce overhead [36]. Given this trade-off, kernel developers have been going back-and-forth on whether to enable THP by default. From version 4.8 to the present, THP is disabled by default.

In general, THP has positive effects on tests that access a large amount of memory. In particular, `huge-read` slows down by as much as 83% on versions with THP disabled. It is worth noting that THP also diminishes the slowdowns caused by other root causes. For example, THP reduces the impact of Kernel Page Table Isolation (§4.1.1), since KPTI adds overhead on every kernel trap whereas THP reduces the number of page faults.

4.2.4 Userspace Page Fault Handling

Enabled in versions 4.6, 4.8, and later versions, userspace page fault handling allows a userspace process to handle page faults for a specified memory region [30]. This is useful for a userspace virtual machine monitor (VMM) to better manage memory. A VMM could inform the kernel to deliver page faults within the guest's memory range to the VMM. One use of this is for virtual machine migration so that the pages can be migrated on-demand. When the guest VM page faults, the fault will be delivered to the VMM, where the VMM can then communicate with a remote VMM to fetch the page.

Overall, userspace page fault handling introduced negligible overhead except for the `big-fork` test which was slowed down by 4% on average. This is because `fork` must check

each memory region in the parent process for associated userspace page fault handling information and copy this to the child if necessary. When the parent has a large number of pages that are mapped, this check becomes expensive.

4.3 Configuration Changes

Three of the root causes are non-optimal configurations. Forced context tracking (§4.3.1) is a misconfiguration by the kernel and Ubuntu developers and causes the biggest slowdown in this category. The other two are the consequences of older kernel versions lacking specifications for the newer hardware used in our experiments, thus leading to non-optimal decisions being made. While this reflects a limitation of our methodology (*i.e.*, running old kernels on new hardware), these misconfigurations could impact real Linux users. First, kernel patches on hardware specifications may not be released in a timely manner: the release of the (simple) patch that specifies the size of second-level TLB did not take place until six months after the release of the Haswell processors, during which time users of the new hardware could suffer a 50% slowdown on certain workloads (§4.3.2). This misconfiguration could impact any modern processor with a second level of TLB. Furthermore, hardware specifications for the popular family of Haswell processors are not back-ported to older kernel versions that still claim to be actively supported (§4.3.3).

4.3.1 Forced Context Tracking

Released into the kernel by mistake in versions 3.10 and 3.12–15, forced context tracking (FCT) is a debugging feature that was used in the development of another feature, reduced scheduling-clock ticks [40]. Nonetheless, FCT was enabled in several Ubuntu release kernels due to misconfigurations. This caused a minimum of approximately 200–300ns overhead in every trip to and from the kernel, thus significantly affecting all of our tests (see Figure 1). On average, FCT slows down each of the 28 tests by 50%, out of which 7 slow down by more than 100% and another 8 by 25–100%.

The reduced scheduling-clock ticks (RSCT) feature allows the kernel to disable the delivery of timer interrupts to idle CPU cores or cores running only one task. This reduces power consumption for idle cores and interruptions for cores running a single compute-intensive task. However, work normally done during these timer interrupts must now be done during other user-kernel mode transitions like system calls. Such work is referred to as context tracking.

Context tracking involves two tasks—CPU usage tracking and participation in the read-copy update (RCU) algorithm. Tracking how much time is spent in userspace and the kernel is usually performed by counting the number of timer interrupts. Without timer interrupts, this must be done on other kernel entries and exits instead. Context tracking also participates in RCU, a kernel subsystem that provides lockless synchronization. Conceptually, under RCU, each object is immutable; when writing to the object, it is copied and updated, resulting in a new version of the object. Because the write does not perturb existing reads, it can be carried out at any time. However, deleting the old version of the object can only be done when it is no longer being read. Therefore, each write also sets a callback to be invoked later to delete the old version of the object when it is safe to do so. The readers cooperate by actively informing RCU when they start and finish reading an object. Normally, RCU checks for ready callbacks and invokes them at each timer interrupt; but under RSCT, this has to be performed at other kernel entries and exits.

FCT performs context tracking on every user-kernel mode transition for *every core*, even on the ones without RSCT enabled. FCT was initially introduced by the Linux developers to test context tracking before RSCT was ready, and is automatically enabled with RSCT. The Ubuntu developers mistakenly enabled RSCT in a release version, hence inadvertently enabling FCT. When this was reported as a performance problem [14], the Ubuntu developers disabled RSCT. However, this still failed to disable FCT, as the Linux developers accidentally left FCT enabled even after RSCT was working. This was only fixed in later Ubuntu distributions as a result of another bug report [17], 11 months after the initial misconfiguration.

4.3.2 TLB Layout Change

Introduced in kernel version 3.14, this patch improves performance by enabling Linux to recognize the size of the second-level TLB (STLB) on newer Intel processors. Knowing the TLB's size is important for deciding how to invalidate TLB entries during `munmap`. There are two options: one is to shoot down (*i.e.*, invalidate) individual entries, and the other is to flush the entire TLB. Shoot-down should be used when the number of mappings to remove is small relative to the TLB's capacity, whereas TLB flushing is better when the number of entries to invalidate is comparable to the TLB's capacity.

Before this patch was introduced, Linux used the size of the first-level data and instruction TLB (64 entries on our test machines) as the TLB's size, and is not aware of the larger second-level TLB with 1024 entries. This resulted in incorrect TLB invalidation decisions: for a TLB capacity of 64, Linux calculates the flushing threshold to be 64/64 = 1. This means that, without the patch, invalidating more than just one entry will cause a full TLB flush. As a result, the `med-munmap` test, which removes 10 entries, suffers as much as a 50% slowdown on a subsequent read of a memory-mapped file of 1024 pages due to the increased TLB misses. With the patch, the TLB flush threshold is increased to 16 (1024/64) on our processor, so `med-munmap` no longer induced a full flush. However, this patch was only released six months after the earliest version of the Haswell family of processors was released. Note that `small-munmap` and `big-munmap` were

not affected because the kernel still made the right decision by invalidating a single entry in `small-munmap` and flushing the entire TLB in `big-munmap`.

4.3.3 CPU Idle Power-State Support

Introduced in kernel version 3.9, this patch specifies the fine-grained idle power saving modes of the Intel processor with the Haswell microarchitecture used by our server. Modern processors save power by idling their components. With more of its components turning idle, the processor is said to enter a deeper idle state and will consume less power. However, deeper idle states also take longer to recover from, and this latency results in a lower overall effective operating frequency [25].

Before this patch, the kernel only recognized coarse-grained power saving states. Therefore, when trying to save power, it always turned the processor to the deepest idle state. With this patch, the kernel's idle driver takes control of the processor's power management and utilizes lighter idle states. This increases the effective frequency by 31%. On average, this patch speeds up LEBench by 21%, with the CPU intensive `select` test achieving the most significant speedup of 31%.

While this patch was released in advance of the release of the Xeon processors, it was not backported to the LTS kernel lines which were still supported at the time, including 3.0, 3.2, and 3.4. This means that in order to achieve the best performance for newer hardware, a user might be forced to adopt the newer kernel lines at the cost of potentially unstable features.

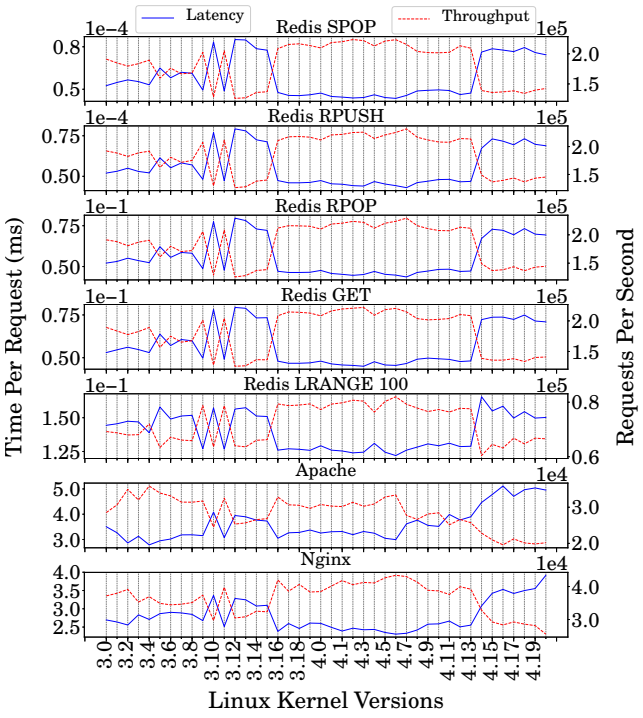

Figure 7. Latency and throughput trends of the Apache Benchmark and selected Redis Benchmark tests (3 write tests and 2 read tests with highest system times).

5 Macrobenchmark

To understand how the 11 identified root causes affect real-world workloads, we evaluate the Redis key-value store [62], Apache HTTP Server [7], and Nginx web server [55],[3] across the Linux kernel versions on which we tested LEBench. Redis' workload was used to build LEBench, while workloads from the other two applications serve as validation. We use Redis' and Apache's built-in benchmarks—Redis Benchmark [61] and ApacheBench [6]—respectively; we also use ApacheBench to evaluate Nginx. Each benchmark is configured to issue 100,000 requests through 50 (for Redis) or 100 (for Apache and Nginx) concurrent connections.

All three applications spend significant time in the kernel and exhibit performance trends (shown in Figure 7) similar to those observed from LEBench. For each test, the throughput trend tends to be the inverse of the latency trend. For brevity, we only display Redis Benchmark's three most kernel-intensive write tests, responsible for inserting (RPUSH) or deleting (SPOP, RPOP) records from the key-value store, and the two most kernel-intensive read tests, responsible for returning the value of a key (GET) and returning a range of values for a key (LRANGE) [60].

We disable the 11 root causes on the kernels and evaluate their impact on the applications. Overall, disabling the 11 root causes brings significant speedup for all three applications, improving the performance of Redis, Apache, and Nginx by a maximum of 56%, 33%, and 34%, and an average of 19%, 6.5%, and 10%, respectively, across all kernels. Four changes—forced context tracking (§4.3.1), kernel page table isolation (§4.1.1), missing CPU idle power states (§4.3.3), and avoiding indirect jump speculation (§4.1.2)—account for 88% of the slowdown across all applications. This is not surprising given that these four changes also resulted in the most significant and widespread impact on LEBench tests, as evident in Figure 2. The rest of the performance-impacting changes create more tolerable and steady sources of overhead: across all kernels, they cause an average combined slowdown of 4.2% for Redis, and 3.2% for Apache and Nginx; this observation is again consistent with the results obtained from LEBench, where these changes cause an average slowdown of 2.6% across the tests. It is worth noting that these changes could cause more significant individual fluctuations—if we only count the worst kernels, on average, each change can cause as much as a 5.8%, 11.5%, and 12.2% slowdown for Redis, Apache, and Nginx, respectively.

[3] In 2019, Redis is the most popular key value store [15]. Apache and Nginx rank first and third in web server market share, respectively, and together account for more than half of all market share [54].

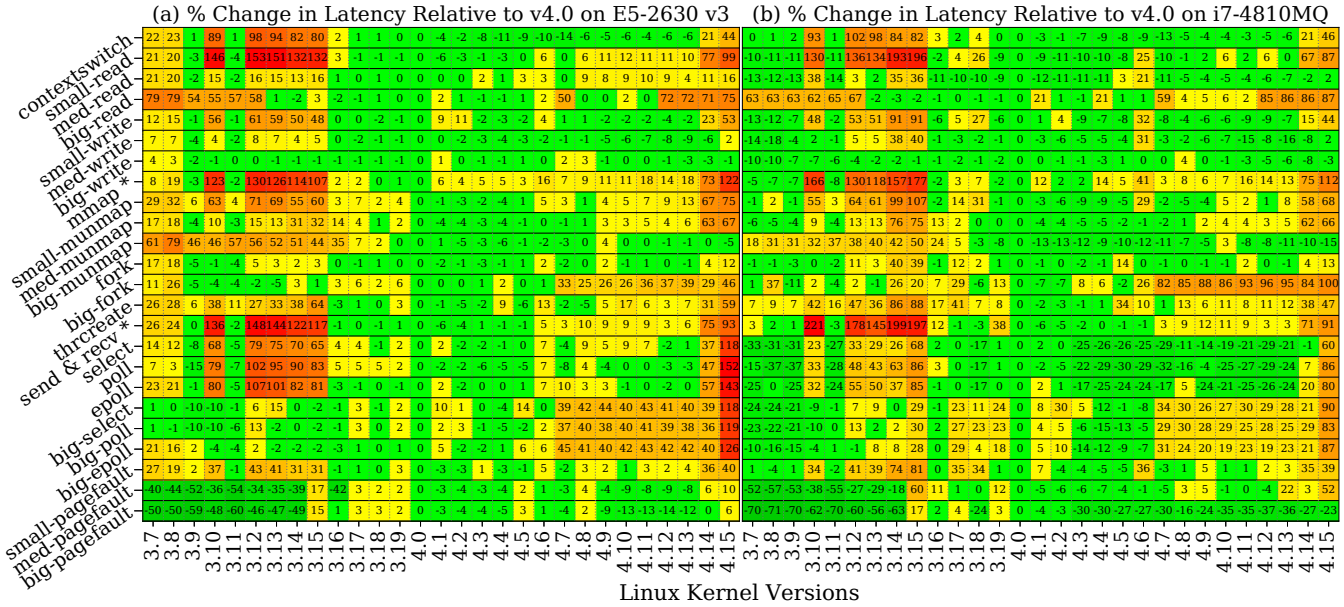

Figure 8. Comparing the results of LEBench on two machines. For brevity, we only show results after v3.7 and before v4.15.

Overall, we find that not only do the macrobenchmarks and LEBench display significant overlap in overall performance trends, they are also impacted by the 11 changes in very similar ways. While it is a limitation that we carried out detailed analysis on the results from a microbenchmark, which does not always exercise kernel operations the same way as a macrobenchmark, we note that even different real-world workloads do not necessarily exercise kernel operations identically. We chose to construct LEBench out of a set of representative real-world workloads, and our evaluation results from the macrobenchmarks confirm the relevance of LEBench.

6 Sensitivity Analysis

To understand how different hardware affects the results from LEBench, we repeat the tests on a laptop with a 2.8GHz Intel i7-4810MQ processor, 32GB of 1600MHz DDR4 memory and a 512GB SSD. Figure 8 displays a side-by-side comparison of the results.

Out of the 11 changes described in §4, 10 have similar performance impacts on LEBench, on the i7 laptop. Updating CPU idle states does not impact the i7 processor's frequency. The other 10 changes manifest in differing degrees of performance impact on each machine due to different hardware speeds. For example, the i7 laptop has a faster processor and slower memory. Therefore, the slowdown due to increased L3 misses from randomizing the SLAB freelist gets exaggerated for `big-fork` (seen after v4.6), likely because the test is memory bound. In addition, we observe more performance variability in the results collected from the laptop, caused by CPU throttling due to over-heating.

7 Discussion

Our findings suggest that kernel performance tuning can play an important role. Unfortunately, thorough performance tuning of the Linux kernel can be extremely expensive. For example, Red Hat and Suse normally require 6-18 months to optimize the performance of an upstream Linux kernel before it can be released as an enterprise distribution [65]. Adding to the difficulty, Linux is a generic OS kernel and thus must support a diverse array of hardware configurations and workloads; many forms of performance optimization do not make sense unless a workload's characteristics are taken into account. For example, Google's data center kernel is carefully performance tuned for their workloads. This task is carried out by a team of over 100 engineers, and for each new kernel, the effort can also take 6-18 months [65].

Unfortunately, this heavyweight performance tuning process cannot catch up with the pace at which Linux is evolving. Our study observes an increasing number of features and security enhancements being added to Linux. In fact, Linux releases a new kernel every 2-3 months, and every release incorporates between 13,000 and 18,000 commits [32]. It is estimated that the mainline Linux kernel accepts 8.5 changes every hour on average [19]. Under such a tight schedule, each release effectively only serves as an integration and stabilization point; therefore, systematic performance tuning is not carried out by the kernel or distribution developers for most kernel releases [27, 65].

Clearly, performance comes at a high cost, and unfortunately, this cost is difficult to get around. Most Linux users cannot afford the amount of resource large enterprises like Google put into custom Linux performance tuning. For the

average user, it may be more economical to pay for a Red Hat Enterprise Linux (RHEL) licence, or they may have to compensate for the lack of performance tuning by investing in hardware (*i.e.*, purchasing more powerful servers or scaling their server pool) to make up for slower kernels. All of these facts point to the importance of kernel performance, whose optimization remains a difficult challenge.

8 Limitations

We restrict the scope of our study due to practical limitations. First, while LEBench tests are obtained from profiling a set of popular workloads, we omitted many other types of popular Linux workloads, for example, HPC or virtualization workloads [3]. Second, we only used two machine setups in our study, and both use Intel processors. A more comprehensive study should sample other types of processor architectures, for example, those used in embedded devices on which Linux is widely deployed. Finally, our study focuses on Linux, and our results may not be general to other OSes.

9 Related Work

Prior works on analyzing core OS operation performance focused either on comparing the same OS on different architectures or different OSes on the same architecture. In comparison, this paper is the first to compare historical versions of the same OS, systematically analyzing the root causes of performance fluctuations over time.

Ousterhout [56] analyzed OS performance on a range of computers and concluded that the OS was not getting faster at the same rate as the processor due to the increasing discrepancy between the speed of the processor and other devices. Anderson *et al.* [5] further zoomed into processor architectures and provided a detailed analysis on the performance implications of different architecture designs to the OS. Rosenblum *et al.* [64] evaluated the impact of architectural trends on operating system performance. They found that despite faster processors and bigger caches, OS performance continued to be bottlenecked by disk I/O and by memory on multiprocessors. Chen and Patterson [12] developed a self-scaling I/O benchmark and used it to analyze a number of different systems. McVoy and Staelin [47] developed lmbench, a microbenchmark that measures a variety of OS operations. Brown and Seltzer [11] further extended lmbench. A large number of tests in lmbench were still focused on measuring different hardware speeds. In comparison, we selected tests from workloads commonly used today; therefore, LEBench might be more relevant for modern applications.

Others have evaluated Linux kernel performance using macrobenchmarks. Phoronix [50, 51] studied Linux's performance across multiple versions but focused on macrobenchmarks, many of which are not kernel intensive. Moreover, they do not analyze the causes of these performance changes. In 2007, developers at Intel introduced a Linux regression testing framework using a suite of micro- and macrobenchmarks [21], which caught a number of performance regressions in release candidates [13]. In contrast, our study focuses on performance changes in stable versions that persist over many versions, which are more likely to impact real users.

Additional studies have analyzed other aspects of OS performance. Boyd-Wickizer *et al.* [10] analyzed Linux's scalability and found that the traditional kernel design can be adapted to scale without architectural changes. Lozi *et al.* [45] discovered Linux kernel bugs that resulted in leaving cores idle even when runnable tasks exist. Pillai *et al.* [58] discovered Linux file systems often trade crash consistency guarantees for good performance.

Finally, Heiser and Elphinstone [20] examined the evolution of the L4 microkernel for the past 20 years and found that many design and implementation choices have been phased out because they either are too complex or inflexible, or complicate verification.

10 Concluding Remarks

This paper presents an in-depth analysis on the evolution of core OS operation performance in Linux. Overall, most of the core Linux operations today are much slower than a few years ago, and substantial performance fluctuations are common. We attribute most of the slowdowns to 11 changes grouped into three categories: security enhancements, new features and misconfigurations. Studying each change in detail, we find that many of the performance impacting changes are possible to mitigate with more proactive performance testing and optimizations; and most of the performance impact is possible to avoid through custom configuration of the kernel. This highlights the importance of investing more in kernel performance tuning and its potential benefits.

Acknowledgements

We would like to thank our shepherd, Edouard Bugnion, and the anonymous reviewers for their extensive feedback and comments on our work. We thank Theodore Ts'o for explaining the practices of Linux performance tuning used by the kernel developers, distributors like Red Hat and Suse, and users like Google.

We thank Tong Liu for collecting traces of the real-world workloads used to develop LEBench. We thank Serguei Makarov for sharing his experiences with upstream software project development. We also thank David Lion for his feedback on this paper.

This research is supported by an NSERC Discovery grant, a NetApp Faculty Fellowship, a VMware gift, and a Huawei grant. Xiang (Jenny) Ren and Kirk Rodrigues are supported by SOSP 2019 student scholarships from the ACM Special Interest Group in Operating Systems to attend the SOSP conference.

References

[1] Advanced Micro Devices. 2018. "Speculative Store Bypass" Vulnerability Mitigations for AMD Platforms. https://www.amd.com/en/corporate/security-updates.

[2] Advanced Micro Devices. 2019. *AMD64 Architecture Programmer's Manual*. Vol. 3. Chapter 3, 262.

[3] Al Gillen and Gary Chen. 2011. The Value of Linux in Today's Fast-Changing Computing Environments.

[4] Amazon Web Services. 2017. AWS re:Invent 2017: How Netflix Tunes Amazon EC2 Instances for Performance (CMP325). https://www.youtube.com/watch?v=89fYOo1V2pA.

[5] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. 1991. The Interaction of Architecture and Operating System Design. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, 108–120.

[6] Apache. 2018. ab - Apache HTTP Server Benchmarking Tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[7] Apache. 2018. Apache HTTP Server Project. https://httpd.apache.org/.

[8] Simon Biggs, Damon Lee, and Gernot Heiser. 2018. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys '18)*. ACM, Article 16, 7 pages.

[9] Jeff Bonwick. 1994. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proceedings of the 1994 USENIX Summer Technical Conference (USTC '94)*. USENIX Association, 87–98.

[10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association, 1–16.

[11] Aaron B. Brown and Margo I. Seltzer. 1997. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '97)*. ACM, 214–224.

[12] Peter M. Chen and David A. Patterson. 1993. A New Approach to I/O Performance Evaluation: Self-scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*. ACM, 1–12.

[13] Tim Chen, Leonid I. Ananiev, and Alexander V. Tikhonov. 2007. Keeping Kernel Performance from Regressions. In *Proceedings of the Linux Symposium*, Vol. 1. 93–102.

[14] Colin Ian King. 2013. Context Switching on 3.11 Kernel Costing CPU and Power. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1233681.

[15] DB-Engines. 2019. DB-engines Ranking. https://db-engines.com/en/ranking.

[16] Docker. 2018. Docker. https://www.docker.com/.

[17] George Greer. 2014. getitimer Returns it_value=0 Erroneously. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1349028.

[18] Graz University of Technology. 2018. Meltdown and Spectre. https://meltdownattack.com/.

[19] Greg Kroah-Hartman. 2017. Linux Kernel Release Model. http://kroah.com/log/blog/2018/02/05/linux-kernel-release-model/.

[20] Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transaction on Computer Systems* 34, 1, Article 1 (April 2016), 29 pages.

[21] Intel Corporation. 2017. Linux Kernel Performance. https://01.org/lkp.

[22] Intel Corporation. 2018. Speculative Execution and Indirect Branch Prediction Side Channel Analysis Method. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00088.html.

[23] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3A. Chapter 4.10.1.

[24] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1. Chapter 11.4.4.4.

[25] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3. Chapter 14.5.

[26] Jake Edge. 2016. Hardened Usercopy. https://lwn.net/Articles/695991/.

[27] Jake Edge. 2017. Testing Kernels. https://lwn.net/Articles/734016/.

[28] Jon Oberheide. 2010. Linux Kernel CAN SLUB Overflow. https://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/.

[29] Jonathan Corbet. 2007. Notes from a Container. https://lwn.net/Articles/256389/.

[30] Jonathan Corbet. 2015. User-space Page Fault Handling. https://lwn.net/Articles/636226/.

[31] Jonathan Corbet. 2017. The Current State of Kernel Page-table Isolation. https://lwn.net/Articles/741878/.

[32] Jonathan Corbet and Greg Kroah-Hartman. 2017. 2017 State of Linux Kernel Development. https://www.linuxfoundation.org/2017-linux-kernel-report-landing-page/.

[33] Judd Vinet and Aaron Griffin. 2018. Arch Linux. https://www.archlinux.org/.

[34] Kirill A. Shutemov. 2014. mm: Map Few Pages Around Fault Address if They are in Page Cache. https://lwn.net/Articles/588802/.

[35] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. (Jan. 2018). arXiv:1801.01203

[36] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 705–721.

[37] Kevin Lai and Mary Baker. 1996. A Performance Comparison of UNIX Operating Systems on the Pentium. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC '96)*. USENIX Association, 265–277.

[38] Linux. 2017. = Transparent Hugepage Support =. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[39] Linux. 2017. Short Users Guide for SLUB. https://www.kernel.org/doc/Documentation/vm/slub.txt.

[40] Linux. 2018. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

[41] Linux. 2018. Page Table Isolation. https://www.kernel.org/doc/Documentation/x86/pti.txt.

[42] Linux. 2019. Memory Resource Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

[43] Linux Containers. 2018. Linux Containers. https://linuxcontainers.org/.

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. (Jan. 2018). arXiv:1801.01207

[45] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*. ACM, Article 1, 16 pages.

[46] Markus Podar. 2014. Current Ubuntu 14.04 Uses Kernel with Degraded Disk Performance in SMP Environment. https://github.com/jedi4ever/veewee/issues/1015.

[47] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC '96)*. USENIX Association, 279–294.

[48] Michael Dale Long. 2016. Unnaccounted for High CPU Usage While Idle. https://bugzilla.kernel.org/show_bug.cgi?id=150311.

[49] Michael Kerrisk. 2012. KS2012: memcg/mm: Improving Memory cgroups Performance for Non-users. https://lwn.net/Articles/516533/.

[50] Michael Larabel. 2010. Five Years of Linux Kernel Benchmarks: 2.6.12 Through 2.6.37. https://www.phoronix.com/scan.php?page=article&item=linux_2612_2637.

[51] Michael Larabel. 2016. Linux 3.5 Through Linux 4.4 Kernel Benchmarks: A 19-Way Kernel Showdown Shows Some Regressions. https://www.phoronix.com/scan.php?page=article&item=linux-44-19way.

[52] Michael Larabel. 2017. The Linux Kernel Gained 2.5 Million Lines of Code, 71k Commits in 2017. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kernel-Commits-2017.

[53] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. USENIX Association, 89–104.

[54] Netcraft. 2019. March 2019 Web Server Survey | Netcraft. https://news.netcraft.com/archives/2019/03/28/march-2019-web-server-survey.html.

[55] Nginx. 2019. NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/.

[56] John K. Ousterhout. 1990. Why Aren't Operating Systems Getting Faster As Fast as Hardware?. In *Proceedings of the 1990 USENIX Summer Technical Conference (USTC '90)*. USENIX Association, 247–256.

[57] Philippe Gerum. 2018. Troubleshooting Guide. https://gitlab.denx.de/Xenomai/xenomai/wikis/Troubleshooting.

[58] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, 433–448.

[59] Randal E. Bryant and David R. O'Hallaron. 2002. *Computer Systems: A Programmer's Perspective* (1 ed.). Prentice Hall, 467–470.

[60] Redis. 2018. Command Reference — Redis. https://redis.io/commands.

[61] Redis. 2018. How Fast is Redis? https://redis.io/topics/benchmarks.

[62] Redis. 2018. Redis. https://redis.io/.

[63] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. 2000. A Comparison of File System Workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference (ATC '00)*. USENIX Association, 41–54.

[64] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. 1995. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, 285–298.

[65] Theodore Y. Ts'o. 2019. Personal Communication.

[66] Thomas Garnier. 2016. mm: SLAB Freelist Randomization. https://lwn.net/Articles/682814/.

[67] Thomas Gleixner. 2018. x86/retpoline: Add Initial Retpoline Support. https://patchwork.kernel.org/patch/10152669/.

[68] Ubuntu. 2018. Ubuntu. https://www.ubuntu.com/.

[69] Vlad Frolov. 2016. [REGRESSION] Intensive Memory CGroup Removal Leads to High Load Average 10+. https://bugzilla.kernel.org/show_bug.cgi?id=190841.

[70] W3Techs. 2018. Usage Statistics and Market Share of Linux for Websites. https://w3techs.com/technologies/details/os-linux/all/all.