;login:

FALL 2020

VOL. 45, NO. 3

usenix

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Guide to the Performance of Optane DRAM
Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson

Understanding Linux Superpages
Weixi Zhu, Alan L. Cox, and Scott Rixner

Interview with Ion Stoica about Serverless Computing
Rik Farrow

Measuring the Health of Open Source Projects
Georg J.P. Link

Using OpenTrace to Track DB Performance Problems
Anatoly Mikhaylov

Columns

SRE and Justice
Laura Nolan

Socially Distant Projects
Cory Lueninghoener

Communicating with BPF
Dave Josephsen

Hand-Over-Hand Locking for Highly Concurrent Collections
Terence Kelly

Everything Is a Punch Card
Simson L. Garfinkel

;login:

FALL 2020 VOL. 45, NO. 3

EDITORIAL

2 **Musings** *Rik Farrow*

SYSTEMS

6 **An Empirical Guide to the Behavior and Use of Scalable Persistent Memory** *Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson*

12 **How to Not Copy Files** *Yang Zhan, Alex Conway, Nirjhar Mukherjee, Ian Groombridge, Martín Farach-Colton, Rob Johnson, Yizheng Jiao, Michael A. Bender, William Jannen, Donald E. Porter, and Jun Yuan*

18 **Interview with Dick Sites** *Rik Farrow*

20 **Understanding Transparent Superpage Management** *Weixi Zhu, Alan L. Cox, and Scott Rixner*

PROGRAMMING

28 **Interview with Ion Stoica** *Rik Farrow*

OPEN SOURCE

31 **Open Source Project Health** *Georg J.P. Link*

SRE

36 **Beneath the SURFace: An MRI-like View into the Life of a 21st-Century Datacenter** *Alexandru Uta, Kristian Laursen, Alexandru Iosup, Paul Melis, Damian Podareanu, and Valeriu Codreanu*

42 **Practical Mitigation Guidance for Hardware Vulnerabilities** *Antonio Gómez-Iglesias, Evan Peregrine, Agata Gruza, Neelima Krishnan, and Pawan Kumar Gupta*

47 **Using OpenTrace to Troubleshoot DB Performance** *Anatoly Mikhaylov*

COLUMNS

52 **SRE and Justice** *Laura Nolan*

54 **Systems Notebook: Socially Distant Projects** *Cory Lueninghoener*

57 **iVoyeur: BPF—Part Three** *Dave Josephsen*

61 **Programming Workbench: Hand-Over-Hand Locking for Highly Concurrent Collections** *Terence Kelly*

67 **SIGINFO: Everything Is a Punch Card** *Simson L. Garfinkel*

72 **For Good Measure: Security Measurement in the Present Tense** *Dan Geer*

75 **/dev/random: WFH** *Robert G. Ferrell*

BOOKS

77 **Book Reviews** *Mark Lamourine and Rik Farrow*

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

EDITOR
Rik Farrow

MANAGING EDITOR
Michele Nelson

COPY EDITORS
Steve Gilmartin
Amber Ankerholz

PRODUCTION
Arnold Gatilao
Ann Heron
Jasmine Murcia
Olivia Vernetti

TYPESETTER
Linda Davis

USENIX ASSOCIATION
2560 Ninth Street, Suite 215
Berkeley, California 94710, USA
Phone: +1 510.528.8649
login@usenix.org

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published quarterly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA.

A one-year print subscription to *;login:* magazine is available only to USENIX Association members at the Sustainer level and higher. Periodicals postage is paid at Berkeley, CA, USA, and additional mailing offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA.

Cover Image by Gerd Altmann, distributed by Pixabay.

EDITORIAL

Musings

RIK FARROW

Rik is the editor of ;login:.
rik@usenix.org

Imagine you are charged with defending the security of one or more systems, yet must also allow other people to run the code of their choice on your systems. I could be talking about your web browsers, those sources of malware infections, but my focus is actually public clouds.

Public clouds offer customers the ability to run any software that is not openly hostile through behaviors like port scanning or launching denial-of-service attacks. That leaves a lot of leeway for various mischief on the hosts they run on, especially if those hosts are running containers or cloud functions—lambdas in AWS-speak.

The initial way of strengthening security for containers was to run each customer's containers on top of a kernel running in a virtual machine (VM). VMs rely on hardware-based security, and while Sun, HP, IBM, and SGI had hardware support for VMs on or before the 1990s, Intel and AMD support appeared in 2005. Hardware support placed guest operating systems in a privilege ring above the virtual machine monitor (VMM), meaning that the VMM had control over the hardware and its treasures: CPUs, memory, storage, and networking.

But running containers inside of VM guests means that cloud vendors lose a lot of what they wanted to gain from container technology. They can't start up containers wherever they want to, as they are constrained by a customer's VM instances. And VMs are slow to start, taking seconds, and require at least an order of magnitude more memory and other resources than lightweight containers. Thus began a quest for more efficient ways of hosting containers.

One early example was NEMU, a stripped down version of QEMU, the open source system emulator. NEMU runs as a process, like QEMU, but instead of having all of the capabilities of QEMU, NEMU dispenses with support for things you can't use in the cloud, like USB, as well as most other devices and hardware emulation, making NEMU smaller and simpler than QEMU. Both QEMU and NEMU are type two hypervisors.

AWS and Google have created their own type two hypervisors, each with the goal of dispensing with VMs for isolating containers and cloud functions/lambdas. Like NEMU, each solution catches accesses to the underlying system, and each limits access using `seccomp()` to reduce the number of system calls that can be made to the host from the hypervisors. You can download the source code to both hypervisors from GitHub if you want to, as both are open source. But the way each has been designed is quite different.

You can read the Firecracker paper [1], presented at NSDI '20, for more on motivation and the deeper details. I found myself fascinated that the paper's authors talk about running more than 8,000 instances on a high-end server, so as to maximize the use of total physical memory, multiple CPUs, and NIC queues.

Firecracker is written in Rust, and like NEMU, provides a limited, virtual system on top of Linux KVM. Firecracker uses `seccomp` to limit the number of system calls to 24 and 30 `ioctls`. Firecracker, like Docker, also relies on some of the same features for isolation, such as namespaces, cgroups, and dropping of privileges. Firecracker provides support for the container or lambdas being run by including a stripped-down Linux kernel. Instead of taking seconds to boot, all this support structure can be running in less than 200 milliseconds. And Firecracker uses more than an order of magnitude less memory and storage than a VM-based approach.

Google programmers used Go, another language that provides strong typing and dynamically allocated and reclaimed memory, like Rust. The Google system, named gVisor, consists of two processes. The first, a hypervisor called Sentry, emulates a Linux system call interface sufficient to run most cloud functions or containers. Like Firecracker, Sentry needs to make system calls to the host, and uses a less severely restricted set at 68. Sentry has its own `netstack` to implement TCP/IP, unlike Firecracker, which uses its guest Linux kernel for the network stack. While Firecracker does away with access to the host file system, instead creating an image file like a VM, gVisor uses a helper application, called Gofer, to handle file system access.

The Firecracker paper doesn't include performance comparisons to gVisor, but a paper by Anjali et al. [2] examines both performance and a measure of security of LXC (native Linux containers), gVisor, and Firecracker. Anjali et al. use microbenchmarks to compare these three container solutions, along with Linux without containers. They report that the Firecracker has high network latency, while gVisor is slower at memory management and network streaming. gVisor is also much slower when it comes to opening and writing files. For security, the paper authors look at code coverage in the Linux kernel including KVM, with the assumption that an isolation solution that relies on more lines of kernel code, running at the highest privilege level, is less likely to be secure due to the potential for kernel bugs. Firecracker does rely less on the underlying Linux kernel, but not by much, using 9.59% of the kernel's 806,318 lines of code versus 11.31% for gVisor.

There are other approaches for isolating containers and cloud functions. Library OSes, also called unikernels, rely on building an application that includes the needed operating system support, and can run on bare metal or on top of a hypervisor like KVM. I ran across Nabla while reading [2] and discovered that Nabla is based upon MirageOS, a unikernel system written in OCaml. Using Nabla requires that the library OS be linked with the application, something I considered a roadblock back when I learned of unikernels [3, 4]. But Nabla was supposed to have a simple, three-step build process, and I tried the example for running "Hello, World!" The build failed at the second step, unable to find `seccomp.h`, even though there were copies of `seccomp.h` handy on my system, including one downloaded for the build.

AWS and Google know that many organizations prefer to build their apps using JavaScript and Python, and though that's possible using unikernel approaches, Firecracker and gVisor are designed to *just work*, as if you were running within a VM running Linux.

The Lineup

We start out this issue with an article based on the FAST '20 paper on Optane performance. Intel Optane, previously known as 3D Xpoint, can be used as main memory or in SSDs, and in the article, Yang et al. use microbenchmarks to tease out the performance characteristics of a system endowed with Optane DIMMs alongside ordinary DRAM, with the hardware support for making data flushed from CPU caches persistent even if power is interrupted.

Zahn et al. wrote "How to Not Copy Files" for FAST '20, and besides being curious about the paper title, I wondered just what was special about their approach—and what was wrong with how other file systems handle file copying. File copying is more important than ever in current systems, with copy-on-write (CoW) being used to speed up file cloning often used with containers. Zahn et al. demonstrate how BetrFS is better and faster at file cloning than any of the current Linux file system favorites while describing their modified B-epsilon trees.

I took advantage of my temporary access to Dick Sites, who wrote about his KUtrace tool in the Summer 2020 issue [5], to ask him some more questions. Honestly, there were a lot more I would have liked to have covered, as Sites has had an insider's view of developments in compilers and CPU architecture since 1966, but at least we dealt with several areas and provided pointers to where you can learn more.

Zhu et al. had an interesting paper about superpages in the Linux kernel. I had heard that superpage support should be disabled, and wondered just what the problem was with something that should increase the performance of memory-hungry applications. It turns out that the answer is complicated, but Zhu and his co-authors do a very good job of explaining the issues while presenting their own solution to improving superpage problems on Linux.

I had wanted to get a couple of the authors of Firecracker and gVisor to write for this issue, but that didn't work out. I did run across a fascinating technical report about cloud programming, and interviewed one of the authors, Ion Stoica, about issues raised in that report. While the cloud does abstract the details of operations, programming in the cloud mostly means microservices today, something very different than what most programmers have been taught how to do.

Georg Link offered to write about open source health: for example, how can you tell if an open source project is healthy enough to be around in five years? The answer to that isn't easy to figure out, but Link provides good suggestions about what he and the Linux Foundation's CHAOSS Project look for when determining health.

Uta et al. add to the understanding of how large clusters work by contributing time-series data of a datacenter in the Netherlands. They call this data MRI-like because it does allow analysis in multiple dimensions. Their contribution, and that of their organization, differs from other contributed traces of large clusters because of the types of applications being run in their DC.

Gómez-Iglesias et al. explain how Intel CPU bugs with names like Meltdown and Spectre are actually likely to affect systems. The authors explain what it takes to carry off a successful attack, the various ways that systems can be patched, and the tradeoffs associated with the different mitigations, mainly loss of performance.

Anatoly Mikhaylov shares his experience in using tagging and OpenTrace to connect requests coming in to a service with database performance issues. Associating a particular request with an unusually slow SQL query isn't easy, because of the intermediate layers found in today's software architecture. Mikhaylov, who works at Zendesk, explains how he and coworkers have worked out how to do this cleanly.

Laura Nolan, reacting to Black Lives Matter, writes about how SREs and other technologists can contribute to changing how people of color are treated. Nolan suggests actions that include changing technical language, being aware, and making changes that are within your sphere of action.

Cory Lueninghoener presents the first installment in his column named "Systems Notebook." Lueninghoener describes how he and a group of coworkers avoided failure in the design of a new system management stack. Instead of plugging away in isolation and later presenting their new system, his group decided to involve others at his site to avoid problems down the road with missing features and lack of acceptance because they had excluded interested parties.

Dave Josephsen continues with his examination of eBPF. Josephsen begins with mythical lovers, forced to communicate through a crack in a wall. He compares this to communications between BPF within the kernel and its Python stub in userspace, and describes three ways that this communication can occur.

Terence Kelly, in his new column, "Programming Workbench," focuses on a locking technique that often gets mentioned but has been poorly documented. Kelly explains hand-over-hand locking, an easy-to-understand method for protecting data structures, such as linked lists, on systems with multiple threads. Kelly plans to continue on this theme, providing code examples in C for interesting algorithms that deserve more exploration.

Simson Garfinkel launches his own column, "SIGINFO," with some history involving his current place of work. Garfinkel begins with the story of how we wound up with 80-column terminals, covers UNIX's "everything is a file" concept, and winds up tying Multics segments to NVRAM. Garfinkel has a long history of writing, and he loves to get his research right as well.

Dan Geer, working solo this time, considers questions we should be asking about security in the time of the coronavirus. Geer, whose column focuses on metrics and measuring security, takes a deep look at how the pandemic has changed not just the way we work, but also the threats our computer systems and networks now face.

Robert Ferrell contrasts working from home with working remotely. He's done both and suggests that one is definitely more comfortable and sensible than the other.

Mark Lamourine has reviewed three books this time, one about algorithms, another concerning skepticism, and the third about re-engineering legacy software. I review a book of stories about interesting things, often failures, that happened to IT architects and the resulting build outs.

I once asked a professor why there weren't any papers about new operating systems at the SOSP we were attending. His answer was succinct: operating systems are hard. I think it is also hard to create ways to protect those operating systems from the software running above them, doing so in ways that are performant but also should remain secure. When I learned about Firecracker late in 2019, I started studying the current methods, from unikernels to system calls reimplemented in Go. Just as VMs and containers have their place in the clouds of today, so do cloud functions and lambdas, and for these to work efficiently they need to be secured with lightweight technology.

I don't think we have heard the last of developments in this area.

References

[1] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*: https://www.usenix.org/conference/nsdi20/presentation/agache.

[2] Anjali, T. Caraza-Harter, and M. M. Swift, "Blending Containers and Virtual Machines: A Study of Firecracker and gVisor," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*: https://dl.acm.org/doi/pdf/10.1145/3381052.3381315.

[3] A. Kantee and J. Cormack, "Rump Kernels: No OS? No Problem!" *;login:*, vol. 39, no. 5 (October 2014): https://www.usenix.org/publications/login/october-2014-vol-39-no-5/rump-kernels-no-os-no-problem.

[4] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the Library OS from the Top Down," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*: https://dl.acm.org/doi/abs/10.1145/1950365.1950399.

[5] R. L. Sites, "Anomalies in Linux Processor Use," *;login:*, vol. 45, no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/sites.

Notice of Annual Meeting

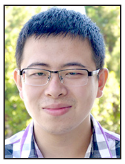
The USENIX Association's Annual Meeting with the membership and the Board of Directors will take place online on Friday, September 25, at 9:00 am PDT.

www.usenix.org/annual-meeting-2020-registration

SYSTEMS

An Empirical Guide to the Behavior and Use of Scalable Persistent Memory

JIAN YANG, JUNO KIM, MORTEZA HOSEINZADEH, JOSEPH IZRAELEVITZ, AND STEVEN SWANSON


Jian Yang received his PhD in computer science at the University of California, San Diego, in 2019. He is currently working on host networking at Google. jian@ia32.me

Juno Kim is a PhD student at the University of California, San Diego, in the Non-Volatile System Laboratory. His research interests lie in the field of storage systems optimized for persistent memory. His advisor is Professor Steven Swanson. juno@eng.ucsd.edu

Morteza Hoseinzadeh is a PhD candidate at the University of California, San Diego, in the Non-Volatile System Laboratory. His research interests include software solutions for safe persistent memory programming, formal verification of persistent data structures, and system programming with a focus on non-volatile memories. He is advised by Professor Steven Swanson. mhoseinzadeh@cs.ucsd.edu

Joseph (Joe) Izraelevitz is an assistant professor at the University of Colorado, Boulder. His interests lie at the intersection of shared memory systems and non-volatile memory. joseph.izraelevitz@colorado.edu

Steven Swanson is a professor in the Department of Computer Science and Engineering at the University of California, San Diego, and the director of the Non-Volatile Systems Laboratory. His research interests include software and architecture of non-volatile, solid-state memories. swanson@eng.ucsd.edu

Researchers have been anticipating the arrival of commercially available, scalable non-volatile main memory technologies that provide "byte-addressable" storage that survives power outages. With the arrival of Intel's Optane DC Persistent Memory Module, we can start to understand the real capabilities and characteristics of these memories and start designing systems to fully leverage them. We experimented with an Intel system complete with Optane and have learned how to get the most performance out of this new technology. Our testing has helped us understand the hidden complexities of Intel's new devices.

Optane Memory Architecture

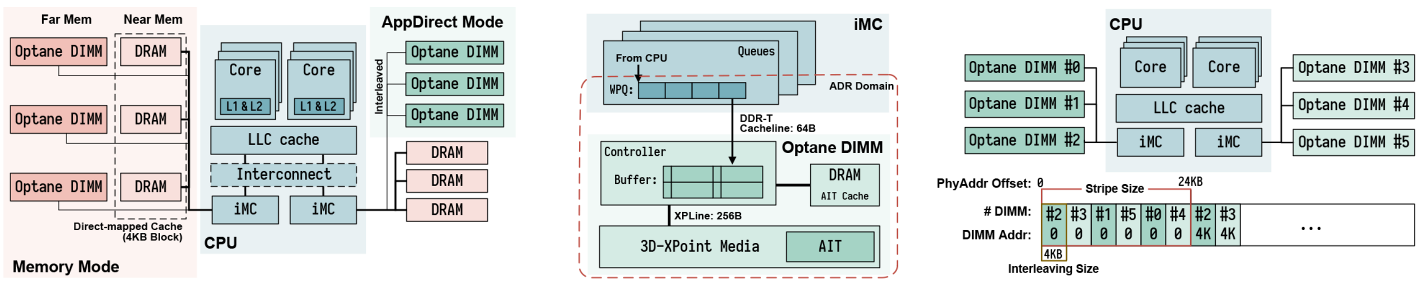
Intel's Optane DC Persistent Memory Module (which we refer to as the Optane DIMM) is the first scalable, commercially available non-volatile DIMM (NVDIMM). Compared to existing storage devices, including Optane SSDs that connect to an external interface such as PCIe, the Optane DIMM has lower latency, higher read bandwidth, and presents a memory address-based interface. Compared to DRAM, it has higher density and persistence.

Like traditional DRAM DIMMs, the Optane DIMM sits on the memory bus, and connects to the processor's integrated memory controller (iMC) (Figure 1a). Intel's Cascade Lake processors are the first CPUs to support the Optane DIMM. Each processor die has two iMCs, and each iMC supports three channels. Therefore, in total, a processor die can support six Optane DIMMs across its two iMCs.

To ensure persistence, the iMC sits within the *asynchronous DRAM refresh* (ADR) domain—Intel's ADR feature ensures that CPU stores that reach the ADR domain will survive a power failure (i.e., will be flushed to the NVDIMM within the hold-up time) [4]. The iMC maintains read and write pending queues (RPQs and WPQs) for each of the Optane DIMMs (Figure 1b), and the ADR domain includes WPQs. Once data reaches the WPQs, the ADR ensures that it will survive power loss. The ADR domain does not include the processor caches, so stores are only persistent once they reach the WPQs. Stores are pulled from the WPQ and sent to the Optane DIMM in cache-line (64-byte) granularity.

Memory accesses to the NVDIMM (Figure 1b) arrive first at the on-DIMM controller (the *Optane controller*), which coordinates access to the Optane media. Similar to SSDs, the Optane DIMM performs an internal address translation and maintains an *address indirection table* (AIT) for this translation [1].

After address translation, the actual access to storage media occurs. As the Optane physical media access granularity is 256 bytes (an *Optane block*), the Optane controller translates smaller requests into larger 256-byte accesses, causing write amplification where small stores become read-modify-write operations. The Optane controller has a small buffer (the *Optane buffer*) to merge adjacent writes.



(a) Optane Platform Modes (Memory and AppDirect) (b) Optane DIMM Overview (c) Interleaved Optane DIMMs

Figure 1: **Overview of (a) Optane platform, (b) Optane DIMM, and (c) how Optane memories interleave.** Optane DIMMs can either be a volatile far memory with a DRAM cache (Memory mode) or persistent memory (App Direct mode).

Operation Modes

Optane DIMMs can operate in two modes (Figure 1a): Memory and App Direct.

Memory mode uses Optane to expand main memory capacity without persistence. It combines an Optane DIMM with a conventional DRAM DIMM that serves as a cache for the NVDIMM. The CPU and operating system simply see the Optane DIMM as a larger (volatile) portion of main memory.

App Direct mode provides persistence and does not use a DRAM cache. The Optane DIMM appears as a separate, persistent memory device.

In both modes, Optane memory can be (optionally) interleaved across channels and DIMMs (Figure 1c). On our platform, the only supported interleaving size is 4 KB. With six DIMMs, an access larger than 24 KB will access all DIMMs.

Instruction Support

In App Direct mode, applications and file systems can access the Optane DIMMs with load and store instructions. Applications modify the Optane DIMM's content using store instructions, and those stores will eventually become persistent. The cache hierarchy, however, can reorder stores, making recovery challenging [3]. The current Intel ISA provides `clflush` and `clflushopt` instructions to flush cache lines back to memory, and `clwb` can write back (but not evict) cache lines. Alternatively, non-temporal stores (`ntstore`) bypass the caches and write directly to memory. All these instructions are non-blocking, so a program must issue an `sfence` to ensure that a previous flush, write back, or non-temporal store is complete and persistent.

Performance Characterization

We find that Optane's performance characteristics are surprising in many ways, and more complex than the common assumption that Optane behaves like slightly slower DRAM.

LATTester

Characterizing Optane memory is challenging for two reasons. First, the underlying technology has major differences from DRAM but publicly available documentation is scarce. Secondly, existing tools measure memory performance primarily as a function of locality and access size, but we have found that Optane performance also depends strongly on memory interleaving and concurrency.

Consequently, we built a microbenchmark toolkit, *LATTester*. To accurately measure the CPU cycle count and minimize the impact from the virtual memory system, LATTester runs as a dummy file system in the kernel and accesses pre-populated (i.e., no page-faults) kernel virtual addresses. LATTester also pins the kernel threads to fixed CPU cores and disables IRQ and cache prefetcher. In addition to latency and bandwidth measurements, LATTester collects a large set of hardware counters from the CPU and NVDIMM.

Our investigation of Optane memory behavior proceeded in two phases. First, we performed a broad, systematic "sweep" over Optane configuration parameters, including access patterns (random vs. sequential), operations (loads, stores, fences, etc.), access size, stride size, power budget, NUMA configuration, and interleaving. Using this data, we designed targeted experiments to investigate anomalies. Across all our tests, we collected over ten thousand data points. The program and data set are available at https://github.com/NVSL/OptaneStudy, while the analysis was published as conference proceedings [5] and a longer technical report [2].

System Description

We performed our experiments on a dual-socket evaluation platform provided by Intel Corporation. The CPUs are 24-core Cascade Lake engineering samples with a similar spec as the previous-generation Xeon Platinum 8160. Each CPU has two iMCs and six memory channels (three channels per iMC). A 32-GB Micron DDR4 DIMM and a 256-GB Intel Optane DIMM are attached to each of the memory channels. Thus the system has 384 GB (2 socket × 6 channel × 32 GB/DIMM) of DRAM, and 3 TB (2 socket × 6 channel × 256 GB/DIMM) of Optane memory. Our machine runs Fedora 27 with kernel version 4.13.0 built from source.

Experimental Configurations

As the Optane DIMM is both persistent and byte-addressable, it can fill the role of either a main memory device (i.e., replacing DRAM) or a persistent device (i.e., replacing disk). In our paper, we focus on the persistent usage.

Our baseline (referred to as *Optane*) exposes six Optane DIMMs from the same socket as a single interleaved namespace (leaving the other CPU socket idle). In our experiments, we used local accesses (i.e., from the same NUMA node) as the baseline to compare with other configurations, such as access to Optane memory on the remote socket (*Optane-Remote*) or DRAM on the local or remote socket (*DRAM* and *DRAM-Remote*). To better understand the raw performance of Optane memory without interleaving, we also create a namespace consisting of a single Optane DIMM and denote it as *Optane-NI*.

Typical Latency

Read and write latencies are key memory technology parameters. We measured read latency by timing the average latency for individual 8-byte load instructions to sequential and random memory addresses. To eliminate caching and queueing effects, we empty the CPU pipeline and issue a memory fence (`mfence`) between measurements (`mfence` serves the purpose of serialization for reading timestamps). For writes, we load the cache line into the cache and then measure the latency of one of two instruction sequences: a 64-bit store, a `clwb`, and an `mfence`; or an `ntstore` and an `mfence`.

Our results (Figure 2) show the read latency as seen by software for Optane is 2×–3× higher than DRAM. We believe most of this difference is due to Optane's longer media latency. Optane memory is also more pattern-dependent than DRAM. The random-vs-sequential gap is 20% for DRAM but 80% for Optane memory, and this gap is a consequence of the Optane buffer. For stores, the instructions commit once the data reaches the ADR at the iMC, so both DRAM and Optane show a similar latency.

Bandwidth

Detailed bandwidth measurements are useful to application designers as they provide insight into how a memory technology will impact overall system throughput. Figure 3 shows the bandwidth achieved at different thread counts for sequential accesses with 256-byte access granularity. We show loads and

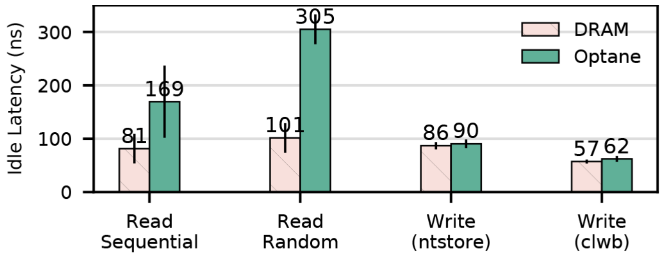


Figure 2: Typical latency. Random and sequential read latency, as well as write latency with `clwb` and `ntstore` instructions. Error bars show one standard deviation.

stores (`Write(ntstore)`), as well as cached writes with flushes (`Write(clwb)`). All experiments use AVX-512 instructions. The left-most graph plots performance for interleaved DRAM, while the center and right-most graphs plot performance for interleaved and non-interleaved Optane. In the non-interleaved measurements all accesses hit a single DIMM.

Figure 4 shows how performance varies with access size. The graphs plot aggregate bandwidth for random accesses of a given size. We use the best-performing thread count for each curve (given as "<`load` thread count> / <`ntstore` thread count> / <`store + clwb` thread count>" in the figure). The data shows that DRAM bandwidth is both higher than Optane and scales predictably (and monotonically) with thread count until it saturates the DRAM's bandwidth, which is mostly independent of access size.

The results for Optane are wildly different. First, for a single DIMM, the maximal read bandwidth is 2.9× the maximal write bandwidth (6.6 GB/s and 2.3 GB/s, respectively), where DRAM has a smaller gap (1.3×) between read and write bandwidth. Second, with the exception of interleaved reads, Optane performance is non-monotonic with increasing thread count. For the non-interleaved (i.e., single-DIMM) cases, performance peaks at between one and four threads and then tails off. Interleaving pushes the peak to 12 threads for `store + clwb`. Third, Optane bandwidth for random accesses under 256 bytes is poor.

Interleaving (which spreads accesses across all six local DIMMs) adds further complexity: Figure 3 (center) and Figure 4 (center) measure bandwidth across six interleaved NVDIMMs. Interleaving improves peak read and write bandwidth by 5.8× and 5.6×, respectively. These speedups match the number of DIMMs in the system and highlight the per-DIMM bandwidth limitations of Optane. The most striking feature of the graph is a dip in performance at 4 KB—this dip is an emergent effect caused by contention at the iMC, and it is maximized when threads perform random accesses close to the interleaving size. We return to this phenomenon later.

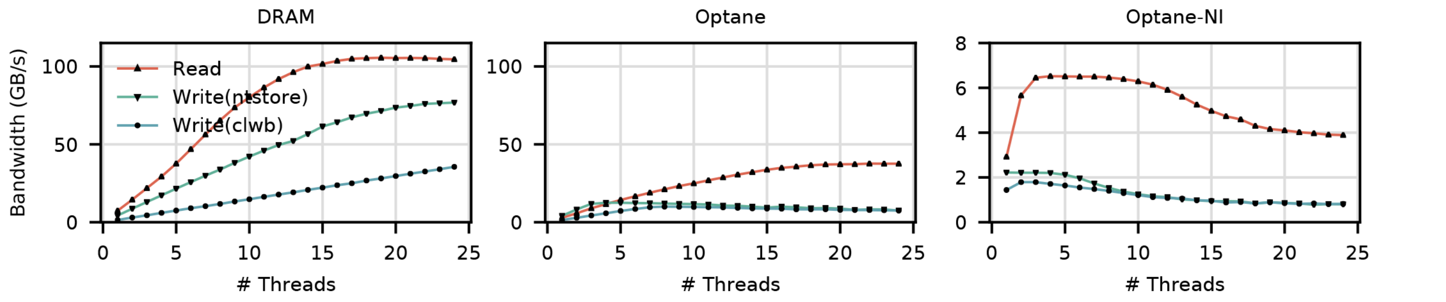


Figure 3: Bandwidth vs. thread count. Maximal bandwidth as thread count increases on local DRAM, non-interleaved, and interleaved Optane memory. All threads use a 256-byte access size.

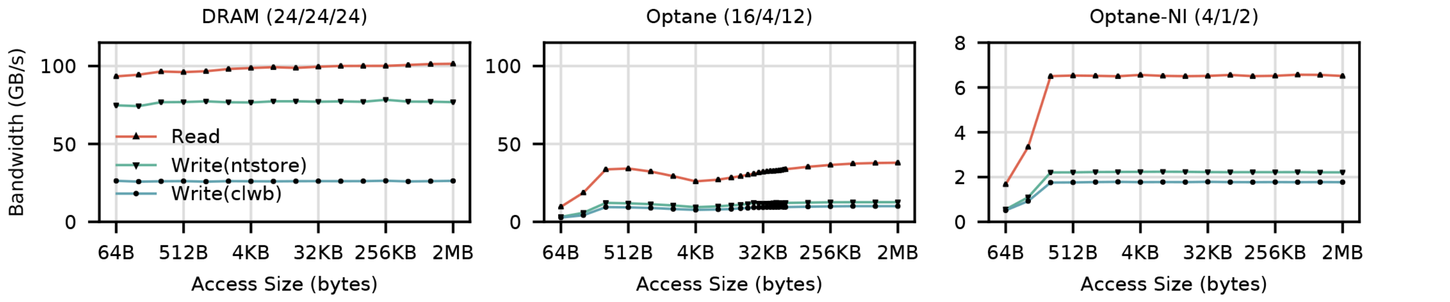


Figure 4: Bandwidth over access size. Maximal bandwidth over different access sizes on local DRAM, interleaved, and non-interleaved Optane memory. Graph titles include the number of threads used in each experiment (Read/Write (`ntstore`) / Write (`clwb`)).

Best Practices for Optane DIMMs

There are many differences between Optane and conventional storage and memory. These differences mean that existing intuitions about optimizing software do not apply directly to Optane. We distill our experiments into a set of four principles for building Optane-based systems.

1. Avoid random accesses smaller than 256 bytes.
2. Use non-temporal stores when possible for large transfers, and control cache evictions.
3. Limit the number of concurrent threads accessing an Optane DIMM.
4. Avoid NUMA accesses (especially read-modify-write sequences).

Avoid Small Random Accesses

Internally, Optane DIMMs update Optane contents at a 256-byte granularity. This granularity means that smaller updates are inefficient since they incur write amplification. The less locality the accesses exhibit, the more severe the performance impact.

To characterize the impact of small stores, we performed two experiments. First, we quantify the inefficiency of small stores using a metric we have found useful in our study of Optane DIMMs. The *Effective Write Ratio (EWR)* is the ratio of bytes issued by the iMC divided by the number of bytes actually written to the Optane media (as measured by the DIMM's hardware counters). EWR is the inverse of write amplification. EWR values below one indicate the Optane DIMM is operating inefficiently since it is writing more data internally than the application requested. Figure 5 plots the strong correlation between EWR and device bandwidth for a single DIMM for all measurements in our sweep of Optane performance. Maximizing EWR is a good way to maximize bandwidth.

Notably, 256-byte updates are EWR efficient, even though the iMC breaks them into 64 byte (cache-line sized) accesses to the DIMM—the Optane buffer is responsible for buffering and combining 64-byte accesses into 256-byte internal writes. As a consequence, Optane DIMMs can efficiently handle small stores, *if they exhibit sufficient locality*. To understand how much locality is sufficient, we crafted an experiment to measure the size of the Optane buffer. First, we allocate a contiguous region of N Optane blocks. During each "round" of the experiment, we first update the first half (128 bytes) of each Optane block. Then we update the second half of each Optane block. We measured the EWR for each round. Figure 6 shows the results. Below $N = 64$ (a region size of 16 KB), the EWR is near unity, suggesting the accesses to the second halves are hitting in the Optane buffer. Above 16 KB, write amplification jumps, indicating a sharp rise in the miss rate, implying the Optane buffer is approximately 16 KB in size. Together these results provide specific guidance for maximizing Optane store efficiency: avoid small stores or, alternatively, limit the working set to 16 KB per Optane DIMM.

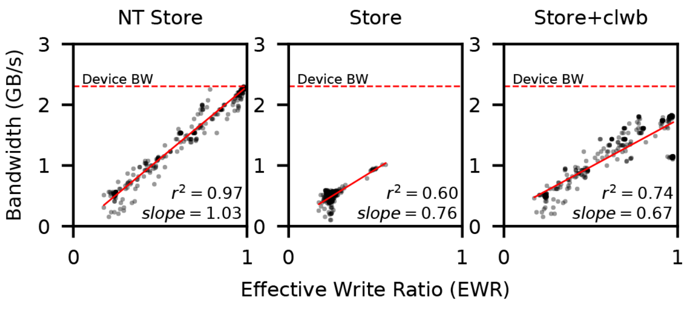


Figure 5: Relationship between EWR and throughput on a single DIMM. Each dot represents an experiment with different access size, thread count, and power budget configurations. Note the correlation between the metrics.

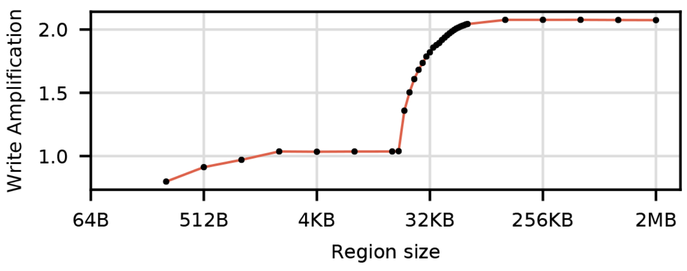


Figure 6: Optane buffer capacity. The Optane DIMM can use the Optane buffer to coalesce writes spread across 16 KB.

Use Non-Temporal Stores for Large Writes

When writing to persistent memory, programmers have several options, each with performance implications. After a regular `store`, programmers can either evict (`clflush`, `clflushopt`) or write back (`clwb`) the cache line. Alternatively, an `ntstore` writes directly to memory, bypassing the cache hierarchy. For all these instructions, a subsequent `sfence` ensures their effects are persistent.

In Figure 7, we compare bandwidth (left) and latency (right) for sequential accesses using AVX-512 stores with three different instruction sequences: `ntstore`, `store + clwb`, and `store` all followed by a `sfence`. Our bandwidth test used six threads since it gives good results for all instructions. The data show that flushing after each 64-byte store improves the bandwidth for accesses larger than 64 bytes. Letting the cache naturally evict cache lines adds nondeterminism to the stream that reaches the Optane DIMM, whereas proactively cleaning the cache ensures that accesses remain sequential. The EWR correlates: adding flushes increases EWR from 0.26 to 0.98.

The data also shows that non-temporal stores have lower latency for accesses over 512 bytes, and the highest bandwidth for accesses over 256 bytes. Here, the performance is due to the fact that a store must load the cache line into the CPU's local cache before execution, thereby using up some of the Optane DIMMs bandwidth. As ntstores bypass the cache, they avoid this extraneous read.

Limit the Number of Concurrent Threads Accessing an Optane DIMM

Systems should minimize the number of threads targeting a single DIMM simultaneously. We have identified two distinct mechanisms that contribute to this effect.

Contention in the Optane Buffer

Contention among threads for space in the Optane buffer will lead to increased evictions, driving down EWR. For example, using eight threads issuing sequential non-temporal stores achieves an EWR of 0.62 and 69% bandwidth compared to a single thread, which has an EWR of 0.98. Figure 3 (right) shows this contention effect in action.

Contention in the iMC

The limited queue capacity in the iMC also hurts performance when multiple cores target a single DIMM. On our platform, the WPQ buffer queues up to 256-byte data issued from a single thread. Since Optane DIMMs are slow, they drain the WPQ slowly, which leads to head-of-line blocking effects.

Figure 4 (center) shows an example of this phenomenon: Optane bandwidth falls drastically when doing random 4 KB accesses across interleaved Optane DIMMs. Due to the random access pattern, periodically all threads will end up colliding on a single DIMM, starving some threads. Thread starvation occurs more often as the access size grows, reaching maximum degradation at the interleaving size (4 KB). For accesses larger than the interleaving size, each core starts spreading their accesses across multiple DIMMs, evening out the load. The write data also show small peaks at 24 KB and 48 KB where accesses are perfectly distributed across the six DIMMs. This degradation effect will occur whenever 4 KB accesses are distributed nonuniformly across the DIMMs.

Avoid Mixed or Multithreaded Accesses to Remote NUMA Nodes

NUMA effects for Optane are much larger than for DRAM, so designers should avoid cross-socket traffic. The cost is especially steep for accesses that mix loads and stores or include multiple threads. Between local and remote Optane memory, the read latency difference is 1.79× (sequential) and 1.20× (random). For writes, remote Optane's latency is 2.53× (`ntstore`) and 1.68× higher compared to local. For bandwidth, remote Optane can achieve 59.2% and 61.7% of local read and write bandwidth at optimal thread count (16 for local read, 10 for remote read, and 4 for local and remote write).

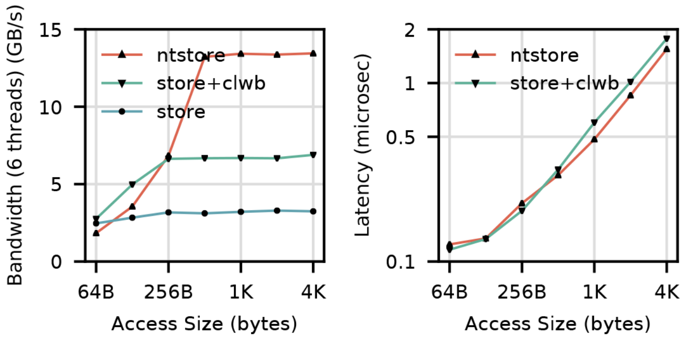


Figure 7: **Persistence instruction performance.** Flush instructions have lower latency for small accesses, but `ntstore` has better latency for larger accesses. Using `ntstore` avoids an additional read from memory, resulting in higher bandwidth.

The performance degradation ratio above is similar to remote DRAM to local DRAM. However, the bandwidth of Optane memory is drastically degraded when either the thread count increases or the workload is read/write mixed. Based on the results from our systematic sweep, the bandwidth gap between local and remote Optane memory for the same workload can be over 30×, while the gap between local and remote DRAM is, at max, only 3.3×.

Conclusion

Our guidelines provide a starting point for building and tuning Optane-based systems. By necessity, they reflect the idiosyncrasies of a particular implementation of a particular persistent memory technology, and it is natural to question how applicable the guidelines will be both to other memory technologies and to future versions of Intel's Optane memory. Ultimately, it is unclear how persistent memory will evolve. Several of our guidelines are the direct product of architectural characteristics of the current Optane incarnation. The size of the Optane buffer and iMC's WPQ might change in future implementations, which would limit the importance of minimizing concurrent threads and reduce the importance of the write granularity. However, expanding these structures would increase the energy reserves required to drain the ADR during a power failure.

The broadest contribution of our analysis and guidelines is that they provide a road map to potential performance problems that might arise in future persistent memories and the systems that use them. Our analysis shows how and why issues like interleaving, buffering, instruction choice, concurrency, and cross-core interference can affect performance. If future technologies are not subject to precisely the same performance pathologies as Optane, they may be subject to similar ones.

Acknowledgments

We thank Subramanya R. Dulloor, Sanjay K. Kumar, and Karthik B. Sukumar from Intel for their support in accessing the test platform. We would also like to thank Jiawei Gao, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Lu Zhang, and Jishen Zhao for technical suggestions. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] B. Beeler, "Intel Optane DC Persistent Memory Module (PMM)": https://www.storagereview.com/intel_optane_dc_persistent_memory_module_pmm.

[2] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. Joon Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," arXiv, August 9, 2019: http://pages.cs.wisc.edu/~yxy/cs839-s20/papers/optane_measurement.pdf.

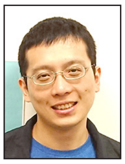
[3] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proceedings of the International Symposium on Computer Architecture (ISCA 2014)*, pp. 265–276: http://web.eecs.umich.edu/~twenisch/papers/isca14.pdf.

[4] A. Rudoff, "Deprecating the PCOMMIT Instruction," Intel, September 12, 2016: https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

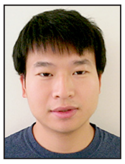
[5] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proceedings of the18th USENIX Conference on File and Storage Technologies (FAST '20)*, pp. 169–182: https://arxiv.org/pdf/1908.03583.pdf.

How to Not Copy Files

YANG ZHAN, ALEX CONWAY, NIRJHAR MUKHERJEE, IAN GROOMBRIDGE, MARTÍN FARACH-COLTON, ROB JOHNSON, YIZHENG JIAO, MICHAEL A. BENDER, WILLIAM JANNEN, DONALD E. PORTER, AND JUN YUAN


Yang Zhan recently completed his PhD at the University of North Carolina at Chapel Hill and now works as a senior engineer in the Operating Systems Kernel Lab at Huawei. yzhan@cs.unc.edu

Alex Conway recently completed his PhD at Rutgers University and is now a researcher at VMware. His interests focus on high-performance storage systems at the intersection of theory and practice. conway@ajhconway.com


Yizheng Jiao is a PhD student in the Computer Science Department at the University of North Carolina at Chapel Hill. He designs and implements efficient storage systems (e.g., in-kernel file systems and databases). yizheng@cs.unc.edu

Nirjhar Mukherjee is an undergrad at the University of North Carolina at Chapel Hill. nirjharm@gmail.com

Ian Groombridge is an undergrad at Pace University. igroombridge2010@gmail.com

Michael A. Bender is a professor of computer science at Stony Brook University. His research focuses on theory of algorithms and their use in storage systems. bender@cs.stonybrook.edu

Making logical copies, or clones, of files and directories is critical to many real-world applications and workflows, including backups, virtual machines, and containers. In this article, we explore the performance characteristics of an ideal cloning implementation; we show why copy-on-write induces a trade-off that prevents existing systems from achieving the ideal constellation of performance features; and we show how to achieve strong cloning performance in an experimental file system, BetrFS.

Many real-world workflows rely on efficiently copying files and directories. Backup and snapshot utilities need to make copies of the entire file system on a regular schedule. Virtual-machine servers create new virtual machine images by copying a pristine disk image. More recently, container infrastructures like Docker make heavy use of file and directory copying to package and deploy applications [5], and new container creation typically begins by making a copy of a reference directory tree.

Duplicating large objects is so prevalent that many file systems support *logical* copies of files or directory trees without making full *physical* copies. A physical copy is one where data blocks are duplicated, whereas a logical copy is one where data blocks may be shared. We call writable, logical copies ***clones***.

Writes to a logical copy should not modify the original file and vice versa. A classic way to maintain the content of a file is copy-on-write (CoW), where shared blocks are physically copied as soon as they are modified. Initially, this approach is also space efficient because blocks or files need not be rewritten until they are modified.

Many logical volume managers support CoW snapshots, and some file systems support CoW file or directory clones via `cp --reflink` or other implementation-specific interfaces. Many implementations have functional limitations, such as only cloning files, special directories marked as "subvolumes," or read-only clones. Nonetheless, we will refer to all these features as cloning.

Performance goal: nimble clones. An ideal clone implementation will have strong performance along several dimensions. In particular, clones should:

- be fast to create;
- have excellent read locality, so that logically related files can be read at near-disk bandwidth, even after modification;
- have fast writes, both to the original and the clone; and
- conserve space, in that the write amplification and disk footprint are as small as possible, even after updates to the original or to the clone.

We call a clone with this constellation of performance features ***nimble***.

Production clone implementations are not nimble. Nimble clones are the performance ideal, but CoW cloning does not yield nimble performance. This may seem surprising, especially given that CoW has been the de facto way to implement clones for decades.

Martín Farach-Colton is a professor of computer science at Rutgers University. His research focuses on theory of algorithms and their use in storage systems. martin@farach-colton.com

Bill Jannen is an assistant professor of computer science at Williams College. His research interests span a variety of topics, from computer science education to storage systems. jannen@cs.williams.edu

Rob Johnson is a senior staff researcher at VMware Research, where he works on the theoretical and applied aspects of high-performance storage systems. robj@vmware.com

Don Porter is an associate professor of computer science at the University of North Carolina at Chapel Hill. His research focuses on improving performance, security, and usability of computer systems. porter@cs.unc.edu

Jun Yuan is an assistant professor of computer science at Pace University. She is interested in building storage systems with solid theoretical foundation and with measured performance that matches the analysis. jyuan2@pace.edu

The Copy-on-Write Granularity Problem, or Why It's Hard to Achieve Nimble Clones

We begin by describing a simple implementation of CoW cloning in an inode-based file system. Although details will vary depending on the specifics of the file system, all existing production file systems share the CoW granularity trade-off illustrated in our simplified design below. This trade-off prevents these file systems from implementing nimble clones.

To clone a file from **a** to **b**, the file system can set up **b**'s inode to point to all the same data blocks as **a**'s inode, and both inodes are modified to mark all blocks as copy-on-write. With this approach, clones are cheap to create. In fact, if the file system uses extent trees to manage file blocks, it can mark entire subtrees of the extent tree as copy-on-write. This means that, to create the clone, the file system needs only to set up the old and new inodes to point to the same extent tree using copy-on-write.

This approach is also space efficient at first and preserves the locality of blocks within the file. If the blocks of the original file were laid out sequentially, then so are the clone's, so sequential reads from both will be fast. Note that this approach does not maintain inter-file locality: the blocks of clone **b** may be quite distant from the blocks of other files in **b**'s directory.

The challenge is to maintain space efficiency and good read locality as the files are edited.

Whenever the file system performs a write to a shared block of either file, the file system must allocate a new block and redirect the modified file's inode to point to the new data block.

This simple but representative implementation of CoW exhibits a trade-off among space conservation, read locality, and write throughput. The main tuning parameter for CoW is the copy granularity. Copy granularity is the size of the data block that is copied when a file is modified. At one extreme, the entire file can be copied, and at the other, the system might only copy a sector on the device—typically 512 bytes or 4 KiB.

File-granularity CoW can have poor write throughput and space efficiency. If one makes a small change to a large file, this small write will incur the cost of copying the entire file and miss a significant opportunity to share a large portion of identical contents. File-granularity CoW favors read locality, but even this goal isn't quite met: if a small file is modified and copied, its placement in storage can cause inter-file fragmentation and, thus, low read throughput for some workloads.

At the other extreme, fine-granularity CoW, say at block granularity, will struggle to conserve locality. Over time, the blocks of a file can scatter across the storage device as they are allocated 4 KiB at a time. For example, consider a large file that is initially placed in a physically contiguous run of blocks, cloned, and then a series of small, random writes are issued to both files. As soon as one block in the middle of this run is modified, the block must be rewritten out-of-place. This block is now far from its neighbors in either the original, the clone, or both. Many file systems have heuristics for placing logically related blocks near each other at allocation time, but, in practice, this is not enough to prevent *aging* over the lifetime of the file system [2].

Put differently, small, random writes force simple CoW schemes either (1) to choose performance at write time and space efficiency (with fine-grained CoW) at the cost of read performance in the future, or (2) to choose read locality in the future (with coarse-grained CoW) at a higher write and space overhead.

BetrFS overcomes this trade-off by (1) aggregating small random application-level writes into large sequential disk writes and (2) using large CoW blocks. By aggregating small random writes, BetrFS ensures that random writes are fast and space efficient. By using large CoW blocks, it ensures that locality is maintained even as sharing is broken. See section "Nimble Clones in BetrFS" for details.

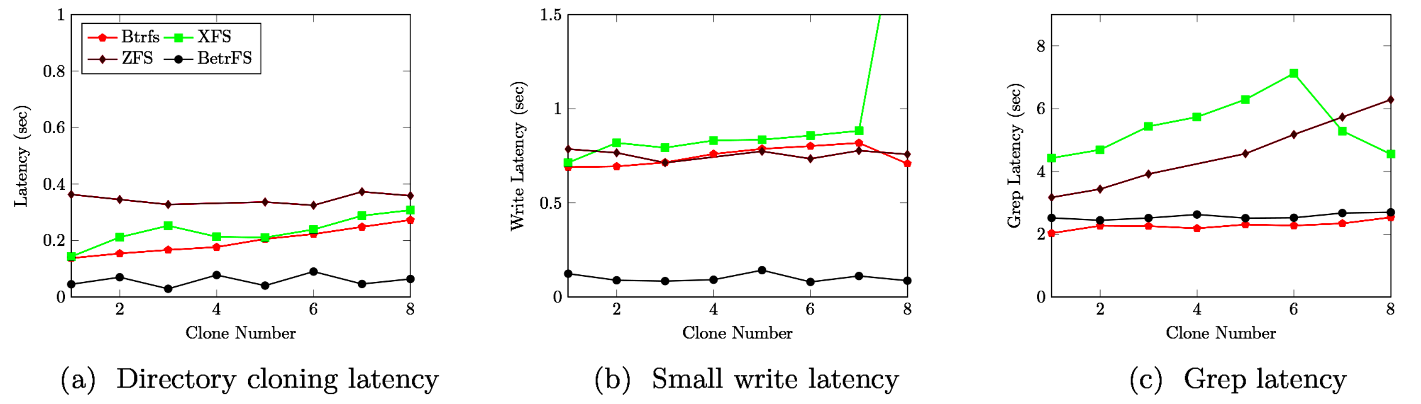


Figure 1: Latency to clone, write, and read as a function of the number of times a directory tree has been cloned. Lower is better for all measures.

Cloning Performance in Real File Systems

In this section, we use a microbenchmark to demonstrate the CoW-granularity trade-off in real file systems, and to show that BetrFS overcomes this trade-off. We then demonstrate how nimble clones can be used to accelerate real applications, such as container instantiation.

Dookubench: A Cloning Microbenchmark

To demonstrate the challenges to cloning performance in production file systems, we wrote a cloning microbenchmark, which we call Dookubench. Like its Star Wars namesake, it makes adversarial use of cloning. The benchmark begins by creating a directory hierarchy with eight directories, each containing eight 4-MiB files. Dookubench then proceeds in rounds. In each round, it creates a new clone of the original directory hierarchy and measures the clone operation's latency. It then writes 16 bytes to a 4 KiB-aligned offset in each newly cloned file—followed by a sync—in order to measure the impact of copy-on-write on writes. The benchmark then clears the file system caches and greps the newly copied directory to measure cloning's impact on read time. Finally, the benchmark records the change in space consumption for the whole file system at each step.

We use Dookubench to evaluate cloning performance in Btrfs, XFS, ZFS, and BetrFS. All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GiB RAM, and a 500 GB, 7200 RPM SATA disk, with a 4096-byte block size. The system runs 64-bit Ubuntu 14.04.5. Note that only BetrFS supports clones of arbitrary files and directories. We deal with the limitations of other file systems as follows. In Btrfs and XFS, we copy the directory structure in each round and use `cp --reflink` to create clones of all the files. For ZFS, we configure the root of the benchmark directory as a sub-volume, and use ZFS's volume snapshotting functionality to perform the clone.

FS	Δ KiB/round
Btrfs	176
XFS	32.6
ZFS	250
BetrFS	16.3

Table 1: Average change in space usage after each Dookubench round (a directory clone followed by small, 4 KiB-aligned modifications to each newly cloned file)

The write-granularity trade-off is illustrated clearly in Table 1 and Figure 1c. XFS uses relatively little space per round, suggesting it is using a small CoW block size. As would be expected of a CoW system with a small block size, Figure 1c shows that the amount of time required to scan through all the contents of the cloned directory degrades with each round of the experiment—after six clones, the grep time is nearly doubled. There appears to be some work that temporarily improves locality, but the degradation trend resumes after more iterations (not shown).

The Btrfs `grep` performance is much flatter, but this comes at the cost of much larger space usage per clone—Btrfs used an average of 176 KiB per clone, compared to 16.3 KiB for BetrFS and 32.6 KiB for XFS. Furthermore, its performance is not completely flat: Btrfs degrades by about 20% during the experiment. After 17 iterations (not presented for brevity), Btrfs read performance degrades by 50% with no indication of leveling off. ZFS is both space-inefficient, using 250 KiB per clone, and shows more than a 2× degradation in scan performance throughout the experiment.

Only BetrFS achieves low space per clone while maintaining locality, as shown by its flat performance on the `grep` benchmark. BetrFS uses 16 KiB per clone—half the space of the next-most-space-efficient file system (XFS)—and its read performance is competitive with the much less space-efficient Btrfs.

BetrFS excels at clone creation (Figure 1a) and small random writes to clones (Figure 1b). BetrFS's cloning time is around 60 ms, which is 33% faster than the closest data point from another file system (the first clone on XFS) and an order of magnitude faster than the worst case for the competition. Furthermore, BetrFS 's clone performance is essentially flat throughout the experiment. ZFS also has flat volume-cloning performance, but not as flat as BetrFS. Both Btrfs and XFS file-level clone latencies, on the other hand, degrade as a function of the number of prior clones; after eight iterations, clone latency is roughly doubled.

In terms of write costs, the cost to write to a cloned file or volume is flat for all file systems, although BetrFS can ingest writes 8–10× faster. This derives from BetrFS's write-optimized design.

In total, these results indicate that BetrFS supports a seemingly paradoxical combination of performance features: clones are fast and space-efficient, and random writes are fast, yet preserve good locality for sequential reads. No other file system in our benchmarks demonstrated this combination of performance strengths, and some also showed significant performance declines with each additional clone.

Cloning Containers

Linux Containers (LXC) is one of several popular container infrastructures that has adopted a number of storage back ends in order to optimize container creation. The default back end (dir) does an `rsync` of the component directories into a single, chroot-style working directory. The ZFS and Btrfs back ends use subvolumes and clones to optimize this process. We wrote a BetrFS back end using directory cloning.

Table 2 shows the latency of cloning a default Ubuntu 14.04 container using each back end. Container instantiation using clones on BetrFS is 3–4× faster than the other cloning back ends, and up to two orders of magnitude faster than the rsync-based back ends. Interestingly, BetrFS is also the fastest file system using the rsync-based back end, beating the next fastest file system (Btrfs) by more than 40%.

Back End	File System	lxc-clone (s)
Dir	ext4	19.514
	Btrfs	14.822
	ZFS	16.194
	XFS	55.104
	NILFS2	26.622
	BetrFS	8.818
ZFS	ZFS	0.478
Btrfs	Btrfs	0.396
BetrFS	BetrFS	0.118

Table 2: Latency of cloning a container

Nimble Clones in BetrFS

This section overviews the four key techniques BetrFS uses to realize nimble clones. The interested reader can find a detailed explanation, as well as related work, in our recent FAST '20 paper [4].

BetrFS [1, 3] is an in-kernel, local file system built on a key-value store (KVstore) substrate. A BetrFS instance keeps two KVstores. The metadata KVstore maps full paths (relative to the mountpoint, e.g., `/foo/bar/baz`) to `struct stat` structures, and the data KVstore maps {full path + block number} keys to 4 KiB blocks.

BetrFS is named for its KVstore data structure, the B^ε-tree [1]. A B^ε-tree is a write-optimized KVstore in the same family of data structures as LSM-trees (Log-Structured Merge-tree). Like B-tree variants, B^ε-trees store key-value pairs in leaves. A key feature of the B^ε-tree is that interior nodes buffer pending mutations to the leaf contents, encoded as *messages*. Messages are inserted into the root of the tree, and, when an interior node's buffer fills with messages, messages are *flushed* in large batches to one or more children's buffers. Eventually, messages reach the leaves and the updates are applied. As a consequence, random updates are inexpensive—the B^ε-tree effectively logs updates at each node. Note that these buffers are bounded in size to a few MiB, and buffers are never allowed to grow so large that they suffer from common pathologies in a fully log-structured file system. And since updates move down the tree in batches, the I/O savings grow with the batch size.

A key change needed to share data at rest is to convert the B^ε-tree into a B^ε-*Directed Acyclic Graph* (DAG). Nodes in the B^ε-DAG can be shared among multiple paths from the root to a leaf; sharing a sub-graph of the B^ε-DAG yields space-efficient clones. So far, this is a standard approach to copy-on-write. A nimble design is realized with four additional techniques.

Technique 1: Write Optimization. In order to avoid the granularity trade-off of CoW, we use buffers in a B^ε-DAG to accumulate small writes to a cloned file or directory. The key feature of write-optimization that contributes to nimble clones is "pinning" messages above a shared node in the B^ε-DAG. For example, if we clone a large file `foo` to `bar` and make a small modification to `bar`, that change is encoded in a message and written into the root of the tree, with destination `bar`. However, this message will not be flushed into a shared node in the B^ε-DAG, or else it would "leak" the change into the original file `foo`. Holding a small "delta" in the parent node is more space efficient than making a full copy for a small change, or even copying one leaf node. Instead, we wait until enough changes for `foo` or `bar` accumulate so that little of the remaining content is shared, and then we break that sharing by creating two unique, unshared copies of a node and repacking the contents, potentially recovering locality. We call this technique *Copy-on-Abundant-Write* (CAW).

Technique 2: Full-Path Indices. BetrFS maintains inter-file locality and supports arbitrary file and directory clones by using full-path indexing. BetrFS indexes all files and blocks by their full path, and paths are sorted in DFS (depth first search) traversal order. This means that all the paths for a sub-tree of the directory hierarchy are contiguous in the key space. As a result, a DFS traversal of the directory hierarchy will correspond to a linear scan of the key-space, which will translate into large sequential I/Os, since the BetrFS B^{ε}-tree uses 4 MiB nodes.

Furthermore, this means that cloning an entire sub-tree of the directory hierarchy corresponds to cloning a contiguous range of keys, all of which have a common prefix.

Technique 3: Lifting. As stated so far, key-value pairs encode full pathnames. So nodes or sub-graphs at rest will be shared but have incorrect, full-path keys along one of the shared paths. In our example above, nodes storing the key-value pairs backing `bar` will initially have `foo` keys. We observe that cloning a file or directory from **`a`** to **`b`** is essentially duplicating all the key-value pairs that start with **`a`** to new key-value pairs in which **`a`** has been replaced by **`b`** in each key.

Lifting removes a common prefix from the keys of a node and instead stores this prefix along with the pointer and pivot keys in the parent. For instance, if an entire B^{ε}-DAG leaf stores key-value pairs under directory `/home/user`, this common prefix would be removed from each key-value pair in that leaf, and instead the prefix is stored once in the parent, retaining only "short" pathnames in the child. With lifting, two parents with different directory prefixes can share a node, copy-on-write, and queries dynamically construct the full-path key based on the path taken through the graph to reach a given node.

Technique 4: Lazy Updates. In order to keep latency of a copy low, we must batch and amortize the cost of updates. First, we create GOTO messages that edit the B^{ε}-DAG itself as they are flushed. This is new; previously, all write-optimized dictionaries only batched changes to the *data*, not the data structure itself. Specifically, a GOTO message encodes a pointer that adds an edge to the graph, redirecting searches for a cloned key range to the source of the copy. These messages are flushed down the graph in a batch, and eventually become regular edges once they reach a target height.

The discussion to this point assumes that a cloned file or directory happens to be within a proper sub-graph in the B^{ε}-DAG; this may not be the case, as nodes in a B^{ε}-DAG do not have the same structure as the file system directory tree. Nodes in a B^{ε}-DAG pack as many keys (in key order) as needed to reach a target node size; thus a node may contain multiple small files packed into a 4 MiB node or a single 4 MiB chunk of a large file. Rather than immediately removing data outside of the cloned range, and making a proper sub-graph with the source prefix removed, we instead add additional bookkeeping to delay these edits.

Specifically, we augment lifted pointers with *translation prefixes*, which can specify both a prefix substitution for data at rest that has not already been handled by lifting and, implicitly, a range of keys in a child to ignore. In the example of cloning `foo`, the root of the sub-graph storing `foo` may also include keys for `fii` and `fuu`; a filter on the path for `bar` would specify that any query that follows this path should ignore keys without prefix `foo`. Similarly, if the sub-graph for `foo` has not yet lifted `foo` out of the children, a translation prefix along the path to `bar` would indicate that, when looking in the `foo` sub-graph, any keys that start with `foo` should be translated to have prefix bar.

Conclusion

This article demonstrates that a variety of file systems operations are instances of a *clone* operation, and the available implementations share the same copy-on-write-induced trade-off. This trade-off can be avoided by using write-optimization to decouple writes from copies, rendering a cloning implementation in BetrFS with the *nimble* performance properties: efficient clones, efficient reads, efficient writes, and space efficiency. The latency of the clone itself, as well as subsequent writes, are kept low by inserting a message into the tree. By making the changes in large batches, BetrFS conserves space. As data is copied on abundant writes, read locality is preserved and recovered by using full-path indexing to repack logically contiguous data into large, physically contiguous nodes. This unlocks improvements for real applications, such as a 3–4× improvement in LXC container cloning time compared to specialized back ends.

Acknowledgments

This research was supported in part by NSF grants CCF-1715777, CCF1724745, CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CCF1712716, CNS-1938709, and CNS-1938180. The work was also supported by VMware, by EMC, and by NetApp Faculty Fellowships.

References

[1] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan, "An Introduction to B$^\varepsilon$-trees and Write-Optimization," *;login;*, vol. 40, no. 5 (October 2015), pp. 22–28: https://www.usenix.org/system/files/login/articles/login_oct15_05_bender.pdf.

[2] A. Conway, A. Bakshi, Y. Jiao, Y. Zhan, M. A. Bender, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and M. Farach-Colton, "How to Fragment Your File System," *;login;*, vol. 4, no. 2 (Summer 2017), pp. 6–11: https://www.usenix.org/system/files/login/articles/login_summer17_02_conway.pdf.

[3] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. Porter, "BetrFS: Write-Optimization in a Kernel File System," *ACM Transactions on Storage*, vol. 11, no. 4 (November 2015), pp. 18:1–18:29: https://www.cs.unc.edu/~porter/pubs/a18-jannen.pdf.

[4] Y. Zhan, A. Conway, Y. Jiao, N. Mukherjee, I. Groombridge, M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, D. Porter, and J. Yuan, "How to Copy Files," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, pp. 75–89: https://www.usenix.org/conference/fast20/presentation/zhan.

[5] F. Zhao, K. Xu, and R. Shain, "Improving Copy-On-Write Performance in Container Storage Driver," 2016 Storage Developer Conference (SDC 2016): https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf.

Interview with Dick Sites

RIK FARROW

Richard L. Sites is a semi-retired computer architect and software engineer. He received his PhD from Stanford University several decades ago. He was co-architect of the DEC Alpha computers and then worked on performance analysis of software at Adobe and Google. His main interest now is to build better tools for careful, nondistorting observation of complex live real-time software, from datacenters to embedded processors in vehicles and elsewhere. dick.sites@gmail.com

Rik is the editor of *;login:*. rik@usenix.org

Part of editing *;login:* means reading the near-final page proofs. The authors have had their chance to correct mistakes that appeared after the production pipeline, and so I have a chance to read each article one last time prior to publication. While reading Dick Sites's article about his kernel tracing tool [1] and his bio, I decided I had some more questions about his article.

I also got to ask Dick about things he's done in his long career. In a three-hour interview at the Computer History Museum [3], Dick says that the summary of places he's worked spans seven pages. He started college (MIT) early and immediately started working as a programmer for IBM. I wasn't so much interested in Dick's early years, although they are fascinating, as I was in other more recent topics, things we covered by phone.

Rik Farrow: As I read your article [1] again, I wondered how you came up with these examples. Were they the results of prior work, or perhaps a lot of experimentation?

Dick Sites: I have been working on and teaching about KUtrace for several years now, and looking at the output from literally hundreds of traces.

As noted in the references to my article [1], Lars Nyland (Nvidia) did the initial scheduler comparison in the class I was teaching at the University of North Carolina in the fall of 2019. I redid it with a simpler program for this article.

Too-early `mwait` shows up in almost all Linux traces on Intel x86, which uses Intel-specific idle loop code, versus the less-aggressive generic code used for AMD chips. The idle loop is a kernel-mode process that does nothing but tries to do it slowly and with little power consumption.

I had seen unusually slow IPC (instructions per cycle) now and then over the last couple of years. I added IPC tracking to KUtrace in late 2017, but I only added the frequency tracing in 2020, which immediately revealed portions of code executing 5× too slowly. That explained the 5× drops in instructions per (constant) cycle, which really means instructions per 1/3.9 nsec on a 3.9 GHz chip.

The original 1972 Cray-1 cycle counter incremented once per CPU cycle and could be read in one cycle. I carried this idea into the first DEC Alpha chip in 1992, and it appeared across the industry by 1994. The 2001 introduction of Intel SpeedStep meant that the CPU clock frequency varied, creating problems for code that used the cycle counter to track elapsed time. Thus the so-called "constant TSC" was introduced in 2005 with a very simple implementation. A CPU clock is created by multiplying up some base clock frequency of say 100 MHz. Multiplying by 39 gives a 3.9 GHz clock; multiplying by eight gives an 800 MHz clock. SpeedStep and follow-ons just vary the multiplier. To produce a constant TSC on a chip advertised as 3.9 GHz, the chip always increments the cycle counter by 39 at a 100 MHz rate, independent of the actual CPU clock multiplier. The same chip advertised as 3.6 GHz would always increment by 36.

Page faults occur all over the place, usually in bursts, as shown in the Cost of Malloc section [1]. Even a trace on a vehicle board showed page fault bursts that were a complete surprise since no paging is done.

I am working on a paper to submit that focuses on explaining the 30× range of response times from 200 absolutely identical in-memory key-value lookup RPCs on a client-server pair of x86 desktops. Some of the underlying reasons for variation are the same as here, but the target audience is different—application programmers in response-time-constrained client-server environments.

RF: These days, eBPF, or just BPF, seems to be the favorite tool for profiling kernel events. I suspect that you wouldn't still be working on KUtrace unless each tool fulfilled different roles. BPF queries kernel structures, from what I understand, while KUtrace seems more focused on capturing timings of kernel events or system calls.

DS: It is all about speed. eBPF takes a bytecode program and interprets it to decide what to do and what to trace. Newer versions have a just-in-time compiler, but that is off by default in Linux. The JIT has been a source of security exposures.

eBPF is useful for tracking less common events or less common packets. The fact that the "F" means "filter" is the clue—it is not designed to track all packets or, in its extended form (the "e"), to track all of anything else. eBPF is not designed to track *all* system calls, interrupts, faults and context switches at full speed in a real-time environment. KUtrace is designed to do that and essentially nothing else, taking about 40 CPU cycles per transition.

The other clue is in your use of the word "profile"—a set of counts of how often something happened, with no timeline relating them. Profiles are useless for understanding variance between execution times of nominally similar tasks, because profiles simply average together all instances. That is what drove me to design KUtrace.

RF: You seem to be focused on Intel architectures? Have you looked at other CPU architectures?

DS: During March 2020 I ported KUtrace to the Raspberry Pi-4B and now have some interesting traces from the low end of the computing spectrum. I will be revising my book proposal, introduction, and some content to change the emphasis from just datacenter software to the entire span of datacenter to embedded computing.

RF: The article [1] you wrote for the Summer 2020 issue and your *ACM Queue* article [2] both feature some amazing graphs. Does KUtrace include tools to help produce such useful visualizations from the output of KUtrace?

DS: Yes, all the diagrams are produced by the KUtrace post-processing programs, posted on GitHub. The `rawtoevent` program turns raw binary trace files into text, `eventtospan` turns transitions into timespans expressed as a long JSON file, and `makeself` packages that and a JavaScript template (4200 non-comment lines) into an HTML/SVG file. The article diagrams are high-resolution screenshots or SVG. I have spent more development time on the diagrams than on the raw tracing.

References
[1] R. L. Sites, "Anomalies in Linux Processor Use," *;login:*, vol. 45, no. 2 (Summer 2020): https://www.usenix.org/system/files/login/articles/login_summer20_05_sites.pdf.

[2] R. L. Sites, "Benchmarking 'Hello World'," *ACM Queue*, vol. 16, no. 5 (November 2018): https://queue.acm.org/detail.cfm?id=3291278.

[3] "Oral History of Dick Sites": https://www.youtube.com/watch?v=A47a6Nqa2aM.

Understanding Transparent Superpage Management

WEIXI ZHU, ALAN L. COX, AND SCOTT RIXNER

Weixi Zhu is currently a fifth-year CS PhD student at Rice University who is advised by Professor Scott Rixner and works closely with Professor Alan Cox. His research area is memory systems aimed at improving their flexibility and performance for both generic and domain-specific architectures. He received his BS in the National Elite Program (computer science) at Nanjing University in 2016 and defended his MS thesis at the Computer Science Department of Rice University in 2018. wxchu@rice.edu

Alan L. Cox is a professor of computer science at Rice University and a long-time contributor to the FreeBSD project. Over the years, his research has sought to address fundamental problems at the intersection of operating systems, computer architecture, and networking. Prior to joining Rice, he earned his BS at Carnegie Mellon University and his PhD at the University of Rochester. alc@rice.edu

Scott Rixner is a professor of computer science at Rice University. His research spans virtualization, operating systems, and computer architecture, with a specific focus on memory systems and networking. His work has led to 11 patents and has been implemented within several open source systems. He is also well versed in the internals of the Python programming language, as he has developed Python interpreters for both embedded systems and web browsers. Prior to joining Rice, he received his PhD from MIT. rixner@rice.edu

Superpages (2 MB pages) can reduce the address translation overhead for large-memory workloads in modern computer systems. We clearly outline the sequence of events in the life of a superpage and explore the design space of when and how to trigger and respond to those events. We provide a framework that enables better understanding of superpage management and the trade-offs involved in different design decisions. Quicksilver, our novel superpage management system, is designed based on the insights obtained by using this framework to improve superpage management.

The memory capacity of modern machines continues to expand at a rapid pace. There is also a growing class of "large memory" data-oriented applications—including in-memory databases, data analysis tools, and scientific computation—that can productively utilize all available memory resources. These large memory applications can process data at scales of terabytes or even petabytes, which cannot fit in the memory. Therefore, they either use out-of-core computation frameworks or build their own heuristics to efficiently cache disk data to avoid the unexpected performance impacts of swapping. As a result, these applications have very large memory footprints, which makes address translation performance critical.

The use of *superpages*, or "huge pages," can reduce the cost of virtual-to-physical address translation. For example, the x86-64 architecture supports 2 MB superpages. Using these 2 MB mappings eliminates one level of the page walk traversal and enables more efficient use of TLB (translation lookaside buffer) entries. Intel's most recent processors can hold 1536 mappings in the TLB. The 2 MB superpages can therefore increase TLB coverage from around 6 MB (0.009% of the memory in a system with 64 GB of DRAM) to 3 GB (4.7%). While this is still a small fraction of the total physical memory capacity of a large machine, it is far more likely to capture an application's short-term working set.

The challenge, however, is for the operating system (OS) to transparently manage memory resources in order to maximize superpage use. Modern systems do not necessarily accomplish this well, which has led to many suggestions that transparent huge page (THP) support be turned off in Linux for performance-critical applications. A better solution, however, is to understand the benefits and limitations of existing superpage management policies in order to redesign and improve them.

We carefully explain and analyze the life cycle of a superpage and present several novel observations about the mechanisms used for superpage management. These observations motivate Quicksilver (https://github.com/rice-systems/quicksilver) [9], an innovative design for transparent superpage management based upon FreeBSD's reservation-based physical superpage allocator. The proposed design achieves the benefits of aggressive superpage allocation but mitigates the memory bloat and fragmentation issues that arise from underutilized superpages. The system is able to match or beat the performance of existing systems in both lightly and heavily fragmented scenarios. For example, when using synchronous page preparation, the system achieves 2× speedups over Linux on PageRank using GraphChi on a heavily fragmented system. On Redis, the system is able to maintain Redis throughput and

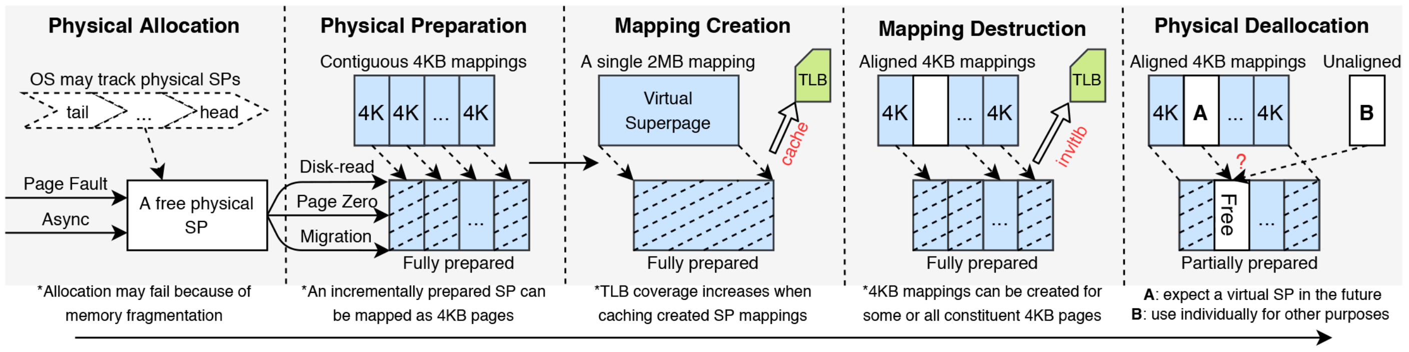


Figure 1: The five events in the life of a superpage (SP)

tail latency as fragmentation increases, whereas the throughput of other systems degrades and tail latency increases. Finally, the system is able to achieve these performance improvements without excessive memory bloat.

Transparent Superpage Management

Kernels manage superpages transparently via these five events:

1. *Physical superpage allocation*: acquisition of a free physical superpage
2. *Physical superpage preparation*: incremental or full preparation of the initial data for an allocated physical superpage
3. *Superpage mapping creation*: creation of a virtual superpage in a process's address space and mapping it to a fully prepared physical superpage
4. *Superpage mapping destruction*: destruction of a virtual superpage mapping
5. *Physical superpage deallocation*: partial or full deallocation of an allocated physical superpage

The five events follow an order that indicates their prerequisites. However, the triggers and handlers for each of these events are determined by the OS and vary across OSes. Figure 1 illustrates the lifetime of a superpage in terms of these five events.

As shown in the figure, the first step in the process is physical superpage allocation. The OS can choose to allocate a physical superpage to back any 2 MB-aligned virtual memory region. A physical superpage could be allocated synchronously upon a page fault or asynchronously via a background task. In order to allocate a physical superpage, the physical memory allocator must have an available, aligned 2 MB region. Under severe memory fragmentation, such regions may not be available.

The second step is to prepare the physical superpage with its initial data. A physical superpage can be prepared in one of three ways. First, if the virtual memory region is anonymous, that is, not backed by a file, then the superpage simply needs to be zeroed. Second, if the virtual memory region is a memory-mapped file, then the data must be read from the file. Finally, if the virtual memory region is currently mapped to independent 4 KB pages, then the contents of those existing pages must be copied into the physical superpage. In this case, the 4 KB pages within the superpage that were not already mapped would need to be prepared appropriately, either via zeroing or reading from the backing file.

Physical superpages can be prepared all at once or incrementally. As each 4 KB page is prepared, it can also be temporarily mapped as a 4 KB page. At a minimum, on a page fault, the 4 KB page that triggered the fault must be prepared immediately in order to allow the application to resume. However, upon a page fault, the OS can choose to prepare the entire physical superpage, only prepare the relevant 4 KB page, or prepare the relevant 4 KB page, allow the application to resume, and prepare the remaining pages later (either asynchronously or when they are accessed).

Once a physical superpage has been fully prepared, the third step is to map that superpage into a process's virtual address space in order to achieve address translation benefits. Before the superpage is mapped, the physical memory can still be accessed via 4 KB mappings; afterwards, the OS loses the ability to track accesses and modifications at a 4 KB granularity. Therefore, an OS may delay the creation of a superpage mapping if only some of the constituent pages are dirty in order to avoid unnecessary I/O in the future.

Superpage mappings are often created upon a page fault, on either the initial fault to the memory region or a subsequent fault after the entire superpage has been prepared. However, if the physical superpage preparation is asynchronous, then its superpage mapping can also be created asynchronously. Note that on some architectures—for example, ARM—any 4 KB mappings that were previously created must first be destroyed.

Fourth, superpage mappings can be destroyed at any time, but must be destroyed whenever any part of the virtual superpage is freed or has its protection changed. After the superpage mapping is destroyed, 4 KB mappings must be recreated for any constituent pages that have not been freed.

	Linux [3]	FreeBSD [6]	Ingens [4]	HawkEye [7]	Quicksilver [9]
Allocation	On first page fault (defragmenting if necessary) and asynchronously for regions with one 4 KB mapping	Created ("reserved") on the first page fault	Asynchronously for regions with 460 4 KB mappings, prioritizing processes with fewer superpages	Asynchronously for regions with one 4 KB mapping, prioritizing heavily utilized regions and processes with big memory usage and high TLB overheads	Created ("reserved") on the first page fault
Preparation	Immediately prepares entire superpage by zeroing or migration	Incrementally prepares in-place 4 KB pages on page faults	Immediately prepares entire superpage by zeroing and migration	Immediately prepares entire superpage by zeroing and migration	Incrementally prepares until a threshold is reached (e.g., 64 in-place 4 KB pages), then prepares the remainder entirely
Mapping	Immediately after allocation and full preparation	Upon the page fault that finishes all preparation	Immediately after allocation and full preparation	Immediately after allocation and full preparation	Upon the page fault that finishes all preparation
Unmapping	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part	When virtual memory is freed, or the mapping is changed, in whole or in part
Deallocation	As soon as the superpage is unmapped	Defers as long as possible	As soon as the superpage is unmapped	As soon as the superpage is unmapped	Defers until the superpage is inactive

Table 1: Comparison of modern superpage management designs

Finally, a physical superpage is deallocated when an application frees some or all of the virtual superpage, when an application terminates, or when the OS needs to reclaim memory. If a superpage mapping exists, it must be destroyed before the physical superpage can be deallocated. Then, either the entire 2 MB can be returned to the physical memory allocator or the physical superpage can be "broken" into 4 KB pages. If the physical superpage is broken into its constituent 4 KB pages, the OS can return a subset of those pages to the physical memory allocator. However, returning only a subset of the constituent pages increases memory fragmentation, decreasing the likelihood of future physical superpage allocations.

Superpage Management Designs

Table 1 presents a comparison of superpage management designs, showing how they handle the five events that occur in the lifetime of a superpage. The table shows two existing operating systems—Linux and FreeBSD—and three state-of-the-art research prototypes—Ingens, HawkEye, and Quicksilver.

Note that the primary differences among these systems are in how they allocate and prepare superpages. There are three key mechanisms that are used to allocate superpages: first-touch, reservations, and asynchronous daemons. The first-touch policy, used exclusively by Linux, allocates, prepares, and maps superpages on the first page fault to a 2 MB-aligned virtual memory region. Linux goes so far as to compact memory if a physical superpage is not currently available in order to attempt to obtain one. This maximizes address translation benefits, as memory is defragmented upon allocation and the superpage mapping is created immediately. However, this also increases page fault latency. In contrast, the reservation-based policy used by FreeBSD and Quicksilver simply *reserves* a physical superpage on the first page fault to a 2 MB-aligned virtual memory region. A physical superpage is allocated for that region, but it is not immediately prepared and mapped. This leads to faster page fault handling, but does not immediately achieve address translation benefits. However, there are benefits to delaying preparation and mapping. If not all of the constituent pages are accessed, then they can be

Workload	Linux-4 KB	Linux-noKhugepaged	Linux
Del-70	11.6 GB	11.7 GB	19.8 GB
Range-XL	14.4 GB	25.7 GB	30.7 GB

Table 2: Redis memory consumption. Linux-noKhugepaged disables khugepaged.

quickly reclaimed under memory pressure, and resources were not wasted on preparation for ultimately untouched pages.

Quicksilver strikes a balance between incremental and all-at-once preparation. Reservations are initially prepared incrementally. This minimizes the initial page fault latency, but loses immediate address translation benefits. Therefore, Quicksilver has an additional threshold, *t*. Once *t* 4 KB pages get prepared, it prepares the remainder of the superpage all-at-once, either synchronously (Sync-*t*) or asynchronously (Async-*t*). This design choice reduces memory bloat, as will be discussed in Observation 1 in the next section, because it does not immediately prepare and map the superpage. However, it enables address translation benefits sooner than waiting for the entire superpage to be accessed.

Linux, Ingens, and HawkEye all utilize asynchronous daemons to allocate, prepare, and map superpages in the background.

Linux's khugepaged is indiscriminate as it scans memory and creates superpages for any aligned 2 MB anonymous virtual memory region that contains at least one dirty 4 KB mapping. As with Linux's first-touch policy, if no free physical superpage exists, it will defragment memory in an attempt to create one. Ingens' and HawkEye's asynchronous daemons both improve upon Linux's indiscriminate allocation policy.

To prevent excessive memory bloat, Ingens increases the threshold of 4 KB pages used to trigger creation of a superpage from one single page to 90%, meaning there must be at least 460 4 KB mappings in a 2 MB region in order to create a superpage for that region. Ingens also prioritizes processes with fewer superpages in order to improve overall fairness. In addition, Ingens actively compacts non-referenced memory in the background.

HawkEye uses the same threshold as Linux: one dirty page. Under memory pressure, it scans mapped superpages and makes their zero-filled 4 KB pages copy-on-write to a canonical zero page to reclaim free memory. HawkEye also maintains a list of candidate 2 MB-aligned regions, but further weights them by the regions' spatial and temporal utilization and the processes' memory consumption and TLB overheads. HawkEye then creates a superpage mapping for the most heavily weighted region in an attempt to make the most profitable promotions first.

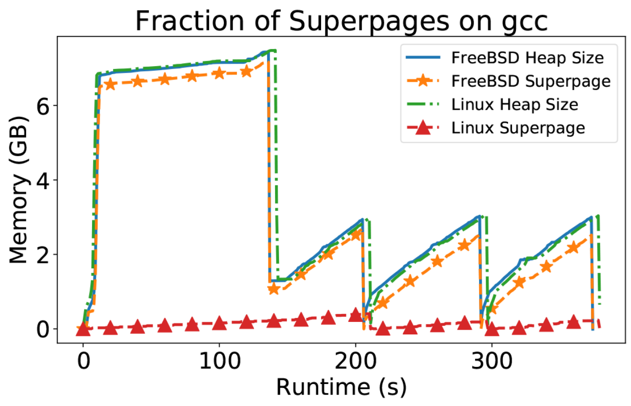


Figure 2: Linux's first-touch policy fails to create superpages.

Analysis of Existing Designs

In this section, we analyze the designs for transparent superpage management described in the previous section and present several novel observations about them. Details on the experimental setup can be found in [9].

Observation 1: Coupling physical allocation, preparation, and mapping of superpages leads to memory bloat and fewer superpage mappings. It also is not compatible with transparent use of multiple superpage sizes.

Linux's first-touch policy couples physical superpage allocation, preparation, and superpage mapping creation together. As a result, it enjoys two obvious benefits: it provides immediate address translation benefits, and it eliminates a large number of page faults. Therefore, it is usually the best policy when there is abundant contiguous free memory.

However, this coupled policy has several drawbacks. First, it can bloat memory and waste time preparing underutilized superpages. In a microbenchmark that sparsely touches 30 GB of anonymous memory, Linux's first-touch policy spends 1.4 sec and consumes 30 GB compared to 0.06 sec and 0.2 GB when disabling transparent huge pages. While such a case is rare when applications use malloc to dynamically allocate memory, it may still happen in a long-running server (for example, Redis). Table 2 shows Redis performance on two workloads: Del-70, which randomly deletes 70% of objects after inserting them, and Range-XL, which inserts randomly sized objects between 256 bytes and 1 MB. The table shows that Linux's first-touch policy bloats memory by 78% compared to Linux with superpages disabled (Linux-4 KB) on the workload Range-XL.

Second, it misses chances to create superpage mappings when virtual memory grows. During a page fault, Linux cannot create a superpage mapping beyond the heap's end, so it installs a 4 KB page, which later prevents creation of a superpage mapping when the heap grows. Figure 2 shows such behavior for gcc [2], which

Page Size	Anonymous	NVMe Disk	Spinning Disk
2 MB	91 µs	**1.7 ms**	**11 ms**
1 GB	**46 ms**	**0.9 sec**	**7.7 sec**

Table 3: Page fault latency. Bold numbers are estimates.

includes three compilations. Linux's first-touch policy creates a few superpage mappings early in each compilation but fails to create more as the heap grows. Instead, promotion-based policies can create more superpages, as seen with FreeBSD and Linux's khugepaged.

Third, it cannot be extended to larger anonymous or file-backed superpages. Table 3 estimates the page-fault latency on both 1 GB anonymous superpages and 2 MB and 1 GB file-backed superpages. Faulting a 2 MB file-backed superpage on the NVMe disk costs 1.7 ms and faulting a 1 GB anonymous superpage takes 46 ms. These numbers may cause latency spikes in server applications. Furthermore, it cannot easily determine which page size to use on first touch. This is arguably more of an immediate problem on ARM processors, which support both 64 KB and 2 MB superpages.

Observation 2: Asynchronous, out-of-place promotion alleviates latency spikes but delays physical superpage allocations.

Promotion-based policies can use 4 KB mappings and later replace them with a superpage mapping. This allows for potentially better-informed decisions about superpage mapping creation and can easily be extended to support multiple sizes of superpages. Specifically, there are two kinds of promotion policies, named out-of-place promotion and in-place promotion. They differ in whether previously prepared 4 KB pages require migration when preparing a physical superpage.

Under out-of-place promotion, a physical superpage is not allocated in advance; on a page fault, a 4 KB physical page is allocated that may neither be contiguous nor aligned with its neighbors. When the OS decides to create a superpage mapping, it must allocate a physical superpage, migrate mapped 4 KB physical pages, and zero the remaining ones. At this time, previously created 4 KB mappings are no longer valid.

Linux, Ingens, and HawkEye perform asynchronous, out-of-place promotion to hide the cost of page migration. As discussed in the previous section, Linux includes khugepaged as a supplement to create superpage mappings. The steady, slow increase of Linux's superpages in Figure 2 is from khugepaged's out-of-place promotions. However, khugepaged can easily bloat memory. Table 2 shows a memory bloat from 11.6 GB to 19.8 GB on workload Del-70. On workload Range-XL, it bloats memory from 25.7 GB to 30.7 GB. Ingens and HawkEye disable Linux's first-touch policy and instead improve the behavior and functionality of khugepaged. Under memory fragmentation, Linux tries to compact memory when it fails to allocate superpages, which blocks the ongoing page fault and leads to latency spikes. Ingens and HawkEye enhance khugepaged and use it as their primary superpage management mechanism.

However, out-of-place promotion delays physical superpage allocations and, ultimately, superpage mapping creations, because the OS must scan page tables to find candidate 2 MB regions and schedule the background tasks to promote them. Table 4 compares in-place promotion (FreeBSD) with out-of-place promotion (Ingens and HawkEye) on applications where superpage creation speed is critical. Both PageRank using GraphChi (GraphChi-PR) [5] and BlockSVM [8] represent important real-life applications, using fast algorithms to process big data that cannot fit in memory. To better illustrate the problem, in Table 4 Ingens* and HawkEye* were tuned to be more aggressive, so that all 2 MB regions containing at least one dirty 4 KB mapping are candidates for promotion. Specifically, Ingens* uses a 0% utilization threshold instead of 90%, and HawkEye* uses a 100% maximum CPU budget to promote superpages. However, Table 4 shows that FreeBSD consistently outperforms both of them. In other words, the most conservative in-place promotion policy creates superpage mappings faster than the most aggressive out-of-place promotion policy.

Observation 3: Reservation-based policies enable speculative physical page allocation, which enables the use of multiple page sizes, in-place promotion, and obviates the need for asynchronous, out-of-place promotion.

In-place promotion does not require page migration. It creates a physical superpage on the first touch, then incrementally prepares and maps its constituent 4 KB pages without page allocation. Therefore, the allocation of a physical superpage is immediate, but its superpage mapping creation is delayed. To bypass 4 KB page allocations, it requires a bookkeeping system to track allocated physical superpages: for example, FreeBSD's reservation system.

FreeBSD's reservation system immediately allocates physical superpages but delays superpage mapping creation, sacrificing some address translation benefits. Navarro et al. reported negligible overheads from the reservation system [6]. Table 4 shows that Linux consistently outperforms FreeBSD when memory is unfragmented, though Linux and FreeBSD both created similar numbers of anonymous superpage mappings.

However, FreeBSD aggressively allocates physical superpages for anonymous memory. Upon a page fault of anonymous memory, it always speculatively allocates a physical superpage,

Workloads	Ingens	Ingens*	HawkEye	HawkEye*	FreeBSD
GraphChi-PR	0.58	0.58	0.53	0.60	0.77
BlockSVM	0.81	0.79	0.73	0.81	0.96

Table 4: Speedup over Linux with unfragmented memory. All systems have worse performance than Linux. The Ingens* and HawkEye* versions are aggressively tuned.

	Linux-4 KB	Linux
Frag-0	1.04 GB/s (5.6 ms)	1.34 GB/s (4.1 ms)
Frag-50	1.04 GB/s (5.7 ms)	0.92 GB/s (10.2 ms)

Table 5: Mean throughput and 95th latency of Redis Cold workload. Frag-X has X% fragmented memory.

expecting the heap to grow. This eliminates one of the primary needs for khugepaged in Linux. In Figure 2, FreeBSD has most of the memory quickly mapped as superpages, because most speculatively allocated physical superpages end up as fully prepared pages.

Observation 4: Reservations and delaying partial deallocation of physical superpages fight fragmentation.

Superpages are easily fragmented on a long-running server. A few 4 KB pages can consume a physical superpage, which benefits little if mapped as a superpage. Existing systems deal with memory fragmentation in three ways.

Linux compacts memory immediately when it fails to allocate a superpage. It tries to greedily use superpages but risks blocking a page fault. Table 5 evaluated the performance of Redis on a Cold workload, where an empty instance is populated with 16 GB of 4 KB objects. Under fragmentation (Frag-50), Linux obtains slightly higher throughput but much higher tail latency than Linux-4 KB.

FreeBSD delays the partial deallocation of a physical superpage to increase the likelihood of reclaiming a free physical superpage. When individual 4 KB pages get freed sooner, they land in a lower-ordered buddy queue and are more likely to be quickly reallocated for other purposes. Therefore, performing partial deallocations only when necessary due to memory pressure decreases fragmentation.

Ingens actively defragments memory in the background to avoid blocking page faults. It preferably migrates non-referenced memory, so that it minimizes the interference with running applications. As a result, Ingens generates fewer latency spikes compared with Linux [4]. These migrations, however, do consume processor and memory resources.

Evaluation

This section provides a brief evaluation of several variants of Quicksilver (Sync-*t* and Async-*t*) against Linux, FreeBSD, Ingens, HawkEye, and their aggressively tuned variants. A more detailed evaluation can be found in [9].

Unfragmented Performance

Sync-1 uses the same superpage preparation and mapping policy for anonymous memory as Linux. With no fragmentation, they perform similarly. However, there are two notable differences. First, Sync-1 speculatively allocates superpages for growing heaps, which allows it to outperform Linux on canneal [1] and gcc [2]. Their similar speedups on reservation-based systems validate Observation 3. Second, Sync-1 creates file-backed superpages and outperforms Linux on GraphChi-PR.

With no fragmentation, FreeBSD outperforms Ingens and HawkEye. This validates Observation 2, as the issue is that out-of-place promotion is slower. Furthermore, on the Redis Cold workload, Ingens and HawkEye even show a degradation over Linux without using superpages.

Sync-64 typically outperforms Async-64 because Async-64 zeros pages in the background, which can cause interference. The comparable performance of Sync-64 and Sync-1 shows that less aggressive preparation and mapping policies can achieve comparable results to immediately mapping superpages on first touch.

Performance under Fragmentation

Linux has a higher tail latency on a Redis Cold workload under fragmentation than Linux without superpages because its on-allocation defragmentation significantly increases page fault latency. In contrast, FreeBSD does not actively defragment memory, so it generates no latency spikes. Ingens and HawkEye offload superpage allocation from page faults and compact memory in the background, so they reduce interference and generate few latency spikes on the Redis Cold workload. Furthermore, their speedup over Linux increases as fragmentation increases.

The four variants of Quicksilver all consistently perform well under fragmentation because their background defragmentation not only avoids increasing page fault latency, but also succeeds in recovering unfragmented performance. Specifically, on the Redis Cold workload with Frag-100, Sync-1 maintained the

	GraphChi-PR	canneal	DSjeng	XZ
Ingens	1.13	1.00	1.01	1.02
HawkEye	1.11	1.01	0.97	1.02
FreeBSD	1.10	1.05	1.04	1.02
Sync-1	2.18	1.12	1.10	1.14
Sync-64	2.11	1.12	1.11	1.14
Async-64	1.68	1.12	1.11	1.13
Async-256	1.65	1.16	1.08	1.13

Table 6: Performance speedup over Linux in a fully fragmented system (Frag-100)

highest throughput (1.31 GB/s) while providing low (4.5 ms) tail latency. This outperforms Linux, the second best system, which only achieved 1.07 GB/s with 5.6 ms tail latency.

Table 6 shows some select results across the systems discussed in the paper in a fully fragmented system (DSjeng and XZ are from SPEC CPU2017 [2]). Note that Quicksilver outperforms the other systems under high fragmentation across a wide range of workloads, but these applications show some of the greatest benefits.

GraphChi-PR is an important real-world workload, and Sync-1 is able to achieve a 2.18× speedup over Linux, far greater than any of the other systems. To better understand that speedup, consider the other variants of Quicksilver on GraphChi-PR. First, in a fully fragmented system, Async-256 performs well because its preemptive and asynchronous superpage deallocation allows many more superpage allocations than the non-Quicksilver systems. Quicksilver is able to defragment memory more efficiently by identifying inactive fragmented superpages. Furthermore, the in-place promotions contribute to the 1.65 speedup of Async-256, which is already much higher than all of the other non-Quicksilver systems. The more aggressive promotion threshold of Async-64 leads to a slightly higher 1.68 speedup.

Second, Sync-64 outperforms Async-64 with a speedup of 2.11. Again, the asynchronous deallocation is beneficial. However, in addition, the synchronous all-at-once preparation implemented by bulk zeroing in Sync-64 efficiently removes the delay of creating superpages. With the same number of superpages created, Sync-64 is able to reduce page walk pending cycles by 76%. Finally, Sync-1 obtains the highest speedup of 2.18 with a more aggressive promotion threshold. While the speedups on the other applications are not as dramatic, the underlying trends are the same.

Conclusion

The solution to perceived performance issues with transparent superpages is not to disable them. Rather it is to carefully understand how superpage management systems work so that they can be improved. The explicit enumeration of the five events involved in the life of a superpage provides a framework around which to compare and contrast superpage management policies. This framework and analysis yielded several key observations about superpage management that motivated Quicksilver's innovative design. Quicksilver achieves the benefits of aggressive superpage allocation, while mitigating the memory bloat and fragmentation issues that arise from underutilized superpages. Both the Sync-1 and Sync-64 variants of Quicksilver are able to match or beat the performance of existing systems in both lightly and heavily fragmented scenarios, in terms of application performance, tail latency, and memory bloat.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 72–81: https://dl.acm.org/doi/10.1145/1454115.1454128.

[2] J. Bucek, K.-D. Lange, and J. V. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*, pp. 41–42: https://dl.acm.org/doi/pdf/10.1145/3185768.3185771.

[3] M. Gorman and P. Healy, "Supporting Superpage Allocation without Additional Hardware Support," in *Proceedings of the 7th International Symposium on Memory Management (ISMM '08)*, pp. 41–50: https://dl.acm.org/doi/10.1145/1375634.1375641.

[4] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 705–721: https://www.usenix.org/system/files/conference/osdi16/osdi16-kwon.pdf.

[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, (OSDI '12)*, pp. 31–46: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-126.pdf.

[6] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pp. 89–104: https://www.usenix.org/legacy/events/osdi02/tech/full_papers/navarro/navarro.pdf.

[7] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient Fine-Grained OS Support for Huge Pages," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, pp. 347–360: https://dl.acm.org/doi/10.1145/3297858.3304064.

[8] H.-F. Yu, C.-J. Hsieh, K.-W. Chang, and C.-J. Lin, "Large Linear Classification When Data Cannot Fit in Memory," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pp. 2777–2782: https://www.ijcai.org/Proceedings/11/Papers/462.pdf.

[9] W. Zhu, A. L. Cox, and S. Rixner, "A Comprehensive Analysis of Superpage Management Mechanisms and Policies," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, pp. 829–842: https://www.usenix.org/system/files/atc20-zhu-weixi_0.pdf.

Interview with Ion Stoica

RIK FARROW

Ion Stoica is a professor in the EECS Department at the University of California, Berkeley, and the director of RISELab (https://rise.cs.berkeley.edu/). He is currently doing research on cloud computing and AI systems. Past work includes Apache Spark, Apache Mesos, Tachyon, Chord DHT, and Dynamic Packet State (DPS). He is an ACM Fellow and has received numerous awards, including the Mark Weiser Award (2019), SIGOPS Hall of Fame Award (2015), SIGCOMM Test of Time Award (2011), and ACM doctoral dissertation award (2001). He also co-founded three companies: Anyscale (2019), Databricks (2013), and Conviva (2006). istoica@berkeley.edu

Rik is the editor of *;login:*. rik@usenix.org

I came across "Cloud Programming Simplified: A Berkeley View on Serverless Computing" via The Morning Paper website and started reading [2]. It turns out that this paper is a follow-up to a 2009 technical report on cloud computing [1]. I started asking people I knew in the authors' list, and Ion Stoica agreed to answer some questions I had about the two reports.

Rik Farrow: Cloud computing brought real advantages, but it left some things essentially unchanged. Organizations no longer needed to buy and maintain hardware, and virtualization meant that hardware could be better utilized. But system administrators still needed to manage their virtual systems, and networking had become more complex. The technical report written by a group at UC Berkeley in 2009 [1] covered these issues in great detail, along with conjectures about how things would evolve over time. How well did this group do with their future projections?

Ion Stoica: Cloud computing succeeded beyond our highest expectations. When we wrote the paper, cloud computing was still a curiosity. Outside of research groups and startups, few organizations bet on cloud. Fast-forward to today and almost every company either uses cloud or is planning to do so. Cloud evolved into a huge market. During the last quarter alone, Microsoft Azure's revenue (including other as-a-service products) passed $11B, AWS exceeded $10B, and Google Cloud reached $2.8B. Even companies like Oracle, who were skeptical of cloud computing at that time, are now putting the cloud at the center of their strategy.

In part, this happened because many of the challenges we listed in our paper were addressed or at least alleviated. Here are just a few examples. The availability of cloud services such as S3 has increased dramatically from two 9s in 2008 (as we reported in our original paper) to four 9s. The performance has increased considerably as well. Today the majority of instances use SSDs instead of HDDs, and there are instances that offer terabytes of RAM and up to 40-Gbps connections. These are at least one order of magnitude improvements over the last decade.

Cloud security made big strides. Today, every major cloud provider offers a myriad of security certifications (e.g., HIPAA, SOC 2, FedRAMP) and even supports new certifications such as GDPR, which were just research proposals a decade ago. Furthermore, cloud providers have started to provide support for hardware enclaves (e.g., Azure Confidential Computing), as well as software enclaves (e.g., AWS Nitro). This allows developers to deploy security protocols and applications not possible before. As a result, virtually every industry is migrating to the cloud, including the ones with stringent security requirements, such as health care, financial services, and retail.

Cloud providers have also improved the ability to scale quickly. In particular, with the advent of serverless computing, customers can instantiate new (function) instances in sub-seconds.

Finally, data locking is more of a mixed bag. On one hand, cloud providers have pushed for proprietary solutions to support data analytics (e.g., BigQuery, RedShift, CosmosDB), machine learning (e.g., SageMaker, Azure ML, Google AutoML), and resource orchestration and management (e.g., Cloud Formation, Azure Factory). On the other hand, virtually every cloud provider hosts virtually every major open source software system, including Hadoop, Spark, Kafka, Redis, Kubernetes, and many more.

Furthermore, a new generation of companies has been successful in providing multi-cloud services, such as Databricks, Confluent, MongoDB, Snowflake, and many more. Part of their success stems from the desire of many enterprises to avoid cloud provider lock-in. I am confident that this will accelerate the standardization of the cloud.

RF: There wasn't just a single step from cloud to serverless. Instead, large cloud providers had already started providing some API-based services, such as storage (S3) and Google App Engine. While these are still important today, except for back-end-as-a-service (BaaS), they don't seem to have become dominant in the move to cloud functions. Do you see an increasing role for BaaS going forward, or have most niches already been filled?

IS: Yes, I expect an increasing role for BaaS. We are already seeing this. For example, Google's Biquery and AWS's Aurora and Athena are rapidly growing in popularity and are supporting more and more traditional database workloads. In addition, we are seeing an increase of BaaS offerings in machine learning, such as Amazon Elastic Inference and Google AutoML.

One reason I expect BaaS to grow in popularity is because the cloud providers have every incentive to push for such services, as they provide higher levels of functionality, which translates to higher revenue and increased "stickiness."

RF: When cloud functions first appeared, cloud providers would provision containers within virtual machines for security purposes. That appears to have shifted over the last several years, with the replacement of VMs with sandboxed container runtimes like gVisor and Firecracker. While these are lighter weight and faster to start up and shut down than VMs, they still appear heavyweight to me. Comments?

IS: Yes, it is true that these are more heavyweighted compared to a simple process or a container. At the same time, as you mentioned, they are significantly lighter and faster than VMs. And I am sure they will improve over time, as researchers and practitioners are continuously optimizing these abstractions.

At the same time, when we are talking about the startup time, we need to look at the big picture. In many cases, the real startup overhead is not to start these containers but to initialize them. For example, the Python environment (e.g., libraries) can easily take hundreds of MBs. Even assuming all data is local and stored on a fast SSD, it might take many seconds to load the libraries and initialize the environment. This can take significantly more time than starting a container. So at least from the startup time point of view, and at least for some applications, the existing sandboxed containers might be already good enough.

RF: Elasticity is one of the most important aspects of cloud functions: both the automatic scaling of function containers as necessary, as well as only having to pay for the resources used instead of reserving those speculatively. But you mention that there are still very real limitations to elasticity in the current support for cloud functions. What are those limitations and how might they be satisfied?

IS: The big challenge with elasticity is that it is at odds with virtually every requirement desired by developers. Each of these requirements adds constraints to where the cloud function can run, which fundamentally limits elasticity. In particular, users want specialized hardware support (e.g., GPUs), they want to run arbitrarily long cloud functions, they want better performance (e.g., co-location), they want fast startup times (e.g., run on nodes which cache the code), and they want security (e.g., do not share the same physical nodes with other tenants when running sensitive code).

Two approaches to address these challenges are (1) relaxing these constraints and (2) workload prediction. One example of relaxing these constraints is developing a low-latency high-throughput shared storage system to store the cloud function's code and environment. Such a system can obviate the need to run a cloud function on a node that has already cached the function's environment. Such a storage system could also be used to efficiently take checkpoints, preempt cloud functions, and restart them on a different node. This could allow cloud providers to relax the running time limits of the functions without hurting elasticity.

Another example is improving the security of cloud functions, which could remove the need to avoid sharing nodes across different tenants running sensitive code.

The other approach to improve elasticity is predicting the workload or application requirements. For instance, if it takes more time to acquire the resources than the application affords, the natural solution is to predict when the application needs these resources and allocate them ahead of time. This will likely require a combination of the application itself providing some hints about its workloads, and machine learning algorithms accurately predicting the application's workload and communication patterns.

RF: One of the biggest advantages of cloud functions is that they put programmers in control, turning operations largely over to the provider's automation. The downside of cloud functions for programmers is that offerings differ widely from provider to provider: there is no standardization. That means customers get locked in to a particular provider, and migration means refactoring entire services. Do you see a way forward here?

IS: This is an excellent point. Cloud providers have the natural incentive to provide differentiated serverless APIs, which can lead to locking.

However, we are starting to see early efforts to provide cross-cloud open source serverless platforms, such as PyWren or Open Lambda, and Apache OpenWhisk. While a dominant open source platform has still to emerge, previous developments give us hope. In particular, at the lower layer of resource orchestration, Kubernetes has already become the de facto standard for container orchestration, and all major cloud providers are supporting it (in addition to their own proprietary offerings).

RF: Programmers must learn new programming paradigms for cloud functions. One function doesn't call another. Instead, programmers must use RPCs, temporary storage, events/queueing, all things that are likely unfamiliar to many programmers. Recently, companies have started to talk about No-code as a way of hiding even more lower-level details, making the use of cloud functions and BaaS even easier. I first heard of this idea around 1989, as "Fifth Generation Programming Languages," an idea that never went anywhere. What do you consider the best way to overcome the barriers to programming using cloud functions?

IS: This is an excellent question. I believe that we will see the emergence of new programming systems that will simplify distributed programming. One example is Ray, a system we have developed in RISELab at UC Berkeley over the past several years. Ray provides support not only for stateless functions, but also for stateful computations (i.e., actors) as well as an in-memory object store for efficient data sharing. In addition, there are many other research projects at Berkeley and elsewhere that aim to provide distributed shared memory abstractions for serverless: for example, Anna [3].

This being said, there are several hard challenges which we will need to address. These challenges stem from the physical characteristics of the underlying infrastructure: the latency of accessing data remotely can be orders of magnitude higher than accessing data locally; the throughput to access data on GPUs is 10× the throughput of local RAM, which is in turn >10× the throughput to a remote node. As a result, the overhead of executing a function remotely can be orders of magnitude higher than executing the function locally. Addressing these challenges calls for new research in compilers that can automatically decide whether a function should be executed locally or remotely and, if remotely, where.

Another challenge, and one of the holy grails of the programming languages, is automatically parallelizing a sequential program. This is a very hard problem which has not been fully solved despite decades of research. This being said, I expect the emergence of serverless computing will spur new efforts that will push the state of the art. In the shorter term, I expect to see tools that target automatic parallelization of specialized workloads, such as big data and ML, as well as tools that assist developers with parallelizing their applications (instead of automatically parallelizing them).

RF: In section 3 of the 2019 paper, you cover five applications that serve to illustrate the current limitations to cloud functions. Summarizing Table 5, these are: object store latency too high, IOPS limits, network broadcast inefficient, lack of fast storage, and lack of shared memory. What, if anything, has changed since your report was written?

IS: It's just a bit over one year since we published our report on serverless computing. Many challenges still remain, but we are already seeing some technologies being developed to alleviate these challenges. These developments are both in the serverless space and in adjacent areas (which I expect will likely impact the serverless space down the line).

In the serverless space, one interesting announcement at the last AWS reinvent was "provision concurrency for lambdas." In a nutshell, this enable users to predefine a number of instances (e.g., concurrency level) of lambdas that can start executing developers' code within a few tens of milliseconds of being invoked. This can go a long way toward making the process of scaling up predictable.

Outside serverless space, an exciting development is the Nitro enclave announced at the same event. This enclave provides both better security and better performance than existing instances. In particular, Nitro provides CPU and memory isolation for EC2 instances, as well as integration with the AES Key Management system. This enables new applications to protect highly sensitive data such as personally identifiable information (PII) and healthcare and financial data. In addition, they improved the bandwidth to EBS (Elastic Block Storage) by 36%, from 14 Gbps to 19 Gbps. Lambdas can already use EBS, and I expect some of the secure technologies in Nitro will later migrate to serverless.

References
[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," Technical Report, 2009: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzales, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," arXiv, February 9, 2019: https://arxiv.org/pdf/1902.03383.pdf.

[3] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling Tiered Cloud Storage in Anna," in *Proceedings of the VLDB Endowment,* vol. 12, no. 6 (February 2019), pp. 624–638: http://www.vldb.org/pvldb/vol12/p624-wu.pdf.

OPEN SOURCE

Open Source Project Health

GEORG J.P. LINK

Georg J.P. Link, PhD, is an open source strategist. During 15 years in open source, Georg experienced the importance of open source project health when the OpenOffice.org community forked the project to start LibreOffice and founded The Document Foundation. This impressive experience inspired Georg's PhD research focus. Today, Georg's mission is to help open source projects, communities, and organizations become more professional in their use of project health metrics and analytics. Georg cofounded the Linux Foundation CHAOSS Project. As the Director of Sales at Bitergia, Georg helps organizations and communities with adopting CHAOSS metrics and technology. In his spare time, Georg enjoys reading fiction and hot-air ballooning. Find Georg online at https://georg.link/ and on Twitter: @GeorgLink, email: georg@chaoss.community

Open source project health describes the potential of projects to continue developing and maintaining quality software, an issue that has long been overlooked. Recently, open source software failures have negatively affected millions of people (e.g., OpenSSL, Equifax), raising the question about the health of open source projects that develop these critically important pieces of software. Measuring and determining the health of open source projects that develop and maintain open source software is a difficult task and has been hard to do well. In this article, I describe issues that make open source project health difficult to measure and what the CHAOSS project has been doing to help with measuring the health of open source projects.

Failures of Open Source Project Health

Software development is often done piecemeal, relying heavily on existing software libraries. For example, the OpenSSL library provides highly specialized encryption algorithms that require expert cryptography knowledge and makes these features available to any developer. This piecemeal approach to software development is fueled by open source software. Increasingly, software libraries are made available through an open source license which encodes the rights for anyone to use, modify, and share the software for any purpose. This licensing model enables developers to collaborate in software production, avoiding duplicate work and improving the software for the benefit of everyone. But despite all the advantages that open source software brings, there are also challenges.

The challenge I explore in this article is in measuring and understanding the health of open source projects. The absence of traditional software project and market indicators makes understanding open source project health quite difficult. The health of proprietary software projects can be measured by revenue from sales that will support future development for the software. Sales figures are nonexistent, and open source licensing means that open source software can be distributed and used by anyone without paying a license fee. Open source project health needs different metrics. This challenge used to be an academic exercise, but today it has the attention of open source foundations, large corporations, and governments. This is because open source projects are a critical part of our digital infrastructure, empowered by projects like OpenSSL, Linux, and Apache Web Server. Many governments, organizations, and individuals depend on open source projects.

Considering the widespread use of open source software, project health failures can have significant impacts. For example, the Heartbleed vulnerability existed in the open source software library OpenSSL [1]. OpenSSL was used by most web servers to secure Internet traffic. Heartbleed allowed a malicious user to get sensitive information from a server, endangering the data of millions of Internet users. This vulnerability was introduced in 2012 and publicly disclosed in 2014. The baffling part of this story is the mismatch between the widespread use of OpenSSL and its very small project community of a few unpaid developers. In hindsight, OpenSSL had poor open source project health, which should have served as a warning signal if only we had paid attention to it.

Heartbleed was a wakeup call to organizations relying on open source software. The Ford Foundation research report *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure* by Nadia Eghbal [2] was very influential in the following conversations. Eghbal had interviewed open source project maintainers and discovered that Heartbleed was merely a very visible open source project health failure while many more open source projects face similar challenges. Some maintainers of open source projects reported suffering burnout from the challenge of securing critical software with little help in their spare time while earning a living in an unrelated job. Several solutions were proposed in response to this realization. For example, the Linux Foundation established the Core Infrastructure Initiative to give money and developer resources to open source projects that were critical for the digital infrastructure but were lacking a healthy project community. Similarly, Mozilla has the Open Source Support (MOSS) program. However, open source project health is more complex than just a matter of lacking financial resources.

OpenSSL's Heartbleed example highlights the need for open source project health to ensure the production of quality open source software. This is not sufficient when users of open source software do not pay attention to changes in the health of open source projects. Equifax, for example, was using the open source software Apache Struts and failed to respond to an update announcement that a vulnerability (CVE-2017-5638) had been fixed in a new version of Struts. Two months after the fix was released, Equifax became subject to a data breach because it was still using a vulnerable and outdated version that hackers exploited [3], and 143 million US consumers were affected. This example highlights that users of open source software have to not only evaluate open source project health once but monitor it continuously and actively for all software and infrastructure components they rely on.

Long-time members of open source projects will tell you that they have developed a sense for open source project health and make decisions based on past experience. However, this sense may not scale to organizations without tools for automation. The open source ecosystem is growing rapidly as more first-time contributors are participating in open source projects. A formalized understanding of how to measure open source project health can transfer this critical knowledge and allow it to be embedded in supporting software.

Measuring Open Source Project Health

Before we can assess open source project health, we need to have clarity on definitions and assumptions. Open source software is at the core of this discussion and is defined as software licensed under an open source license. The Open Source Initiative (https://opensource.org/) is the steward of the Open Source Definition and decides which software licenses are valid open source licenses. The production of open source software is organized in open source projects, which have a technical and a social component. The technical component includes the tools used in software production: source code repositories, issue trackers, mailing lists, CI/CD toolchains, and so on. The social component includes the people involved and how they organize their collaboration: governance, leadership, membership, events, and working groups. Open source community refers to the people involved in an open source project. Just like most people have fingers but unique fingerprints, open source projects have common technical and social components but are not alike. The unique context of each open source project makes it difficult to measure open source project health in a standard and consistent way.

Open source project health is the potential that an open source community will continue developing and maintaining quality software [4]. This assumes that an open source project has the goal of producing software and that the user of the software wants good quality. Because project health is forward looking, an assessment can only speak to the potential and not about a precise probability or likelihood that a community will continue to develop and maintain quality software.

Open source project health can be assessed along three dimensions [5]:

1. Community
2. Code
3. Resources

The *community* dimension captures the idea that open source projects rely on people to contribute. An assessment could look at the diversity of active community members, the size of the community—both contributors and users—and the governance of the community. The *code* dimension captures the idea that open source projects should produce and maintain quality software. An assessment could look at vulnerabilities, code quality, and activity in code review processes. The *resources* dimension captures the idea that open source projects can develop quality software using their own resources, including an infrastructure of specialized hardware, continuous integration systems, testing facilities, and financial resources. An assessment could look at the availability of resources, number of sources providing resources, and how resources are managed within a project. Each of these dimensions focuses on a different aspect of open source project health and can be understood through more metrics than are listed here.

There are two types of data for metrics about open source project health: qualitative and quantitative. Qualitative data can be collected through surveys and interviews with open source community members to understand their perception of a project's health. These valid data collection methods are time-consuming and are rarely done. Recent examples are the Apache Community Survey 2020 and the OpenStack Gender Diversity Report 2018. Quantitative data is typically easier to process and can be automatically collected. A great source of data about open source projects is the trace data that is created as community members collaborate in the creation of software using computer-mediated technology. This includes the Git log, the mailing list archive, and the issue tracker history. Easy-to-collect metrics include quantifying events, such as the numbers of commits, emails, issues, comments, and functions or lines in the source code. While we know that some metrics are easier to obtain than others, the important question is which metrics are most indicative of open source project health.

To date, there is no canonical set of metrics that are most indicative of open source project health. Several studies analyzed historic metrics and correlated them with the continued existence and development of open source projects. In such a setup, a healthy project was one that was developing and maintaining software at the time of the study, and unhealthy projects had stopped development [6]. However, these studies have failed to determine metrics that will be useful. My work has explored these failures through many conversations with open source practitioners in open source projects, organizations, foundations, and government. The unique ways in which each open source project works influence the interpretation of metrics and have so far thwarted all efforts to develop quality models and definitive open source project health metric guidelines.

Building Shared Understanding of Open Source Project Health

Despite the challenges, many open source communities, open source foundations, organizations, and researchers want to determine the health of open source projects. Many lessons have been learned but numerous attempts at measuring open source project health started from scratch because a common language and tool set was missing. The CHAOSS project is seeking to level the playing field and get everyone a head start for understanding the importance of open source project health and how to determine it.

We founded the CHAOSS project, which is an acronym for Community Health Analytics Open Source Software, at the Linux Foundation in 2017. The mission of CHAOSS is to define metrics and software that can help everyone with measuring open source project health. CHAOSS focuses on the basics, such as describing data sources for collecting data about open source projects, defining metrics that can be calculated from that data, and developing a shared language for talking about open source project health. We provide a central location in the open source ecosystem where anyone who is interested in open source project health can come to learn more, discuss ideas, get feedback, and build on existing solutions.

The CHAOSS project has working groups that define related metrics. The five working groups are Diversity and Inclusion, Evolution, Risk, Value, and Common Metrics. To learn more about the metrics in each working group, visit https://chaoss.community/metrics. The key point here is that these working groups think through a variety of issues related to measuring open source project health. For example, the Common Metrics working group describes lower-level metrics that can be used by other working groups for higher-level metrics. One such metric is Organizational Diversity, which can be used by the Risk working group to assess the risk of a single-vendor dependency or by the Evolution working group to assess the growth, maturity, or decline of organizational engagement. The metric Organizational Diversity describes core challenges around identifying which organizations contributors affiliate with, taking into account job changes, contributors using @gmail and not their work email addresses, or combining identities of contributors who use different usernames and email addresses across different collaboration tools. Through these metric definitions, CHAOSS provides a starting point for anyone interested in determining the health of an open source project.

Open source project health metrics can be divided into leading metrics that change rapidly and lagging metrics that are slower to change. On the one hand, we have a fair amount of influence on leading metrics, such as the number of commits or the time to close issues. Setting a goal to increase a leading metric can directly lead to behavior changes in the community. On the other hand, we cannot easily influence lagging metrics, such as the number of long-term contributors or active users of the software. We have so far not found a relationship between leading and lagging metrics that would allow us to say: if you want to improve open source project health as measured by lagging metric X, you need to focus community activities that change leading metric Y and Z. Maybe such a relationship cannot exist because when setting goals for leading metrics, project members may change their behavior to "game" the metric. Gaming of metrics describes a situation in which behavior is targeted to improve a metric while possibly working against the original goals for which the metric was chosen. An example of this is the Number of Commits metric, which measures developer contributions, but developers can easily split a commit into many smaller commits, creating more managerial overhead instead of producing more contributions. Nevertheless, leading metrics can be used in tactical decisions for improving the health of our projects while lagging metrics may be better for tracking long-term goals, of course, taking into account the context of the project.

The CHAOSS project stays neutral about the interpretation of metrics and what they mean in the determination of open source project health. This approach to determining open source project health accommodates the fact that metrics are highly context-sensitive, and open source projects have many different contexts. Projects use a different mix of technical and social components. Even when using the same collaboration tools, projects have different patterns of collaboration and expected behaviors. Whereas some projects are run by volunteers, others are run by organizational employees. Some projects have benevolent dictators who make many decisions, while others have committees or governing boards who collectively make decisions. Some projects have CI/CD pipelines and automated tests that facilitate feedback on code contributions, and others rely more on human reviewers. These are just examples of the large variety of contexts that open source projects create and that make it difficult to interpret the meaning of metrics. One approach to overcoming this challenge is to have an expert on an open source project interpret the metrics specific to that context and tell a story of the project's health, informed and supported by metrics. Determining open source project health is therefore storytelling supported by metrics and evidence.

Improving Open Source Project Health

Having an honest assessment of open source project health can inform data-driven decisions. Following this idea, I discuss thoughts on how open source project health can inform different stakeholders. My opinion has been shaped by conversations in the CHAOSS project, the SustainOSS.org community, my PhD research, and my current job at Bitergia.

Open source communities can observe open source project health to learn about themselves. Since metrics are not absolute indicators of project health, changes over time can be helpful to identify when to take action. For example, when core contributors to a project are leaving, then the community may have a project health issue as indicated by a decline in issue tracker activity. Conversely, a spike in issue tracker activity may indicate that more users are asking questions about the software, and engaging them strategically can draw them in to grow the community. However, context matters because a spike in activity could be the result of outside factors. I recently experienced this in the CHAOSS project when the number of issue comments tripled over the course of one month because of students interested in applying for the paid Google Summer of Code mentoring program.

Organizations can observe open source project health to mitigate risk when relying on open source software in their operations and value creation. Project health can also inform organizations' strategic decisions regarding which projects to engage in and how to maximize value extraction from open source software. For example, a decline of development activity in an open source project can be an early indicator of risk, and an organization can dedicate employee time to such a project to make sure it stays maintained and compatible with new technology developments, standards, and regulatory requirements.

Open source foundations can observe open source project health to identify best practices and learn from open source projects that are doing very well to then help other projects achieve similar outcomes. Foundations can also use the same metrics to help themselves by observing, for example, who active members in the open source projects are and recruiting them as new foundation members, strengthening the relationships between open source project members and thereby improving project health. Foundations are stewards of open source projects and need to have early indicators of changes in order to intervene when needed.

Contributors to open source projects can use open source project health to make decisions about which projects they want to be part of and how to have the most impact. Contributors prefer healthy open source projects because they are easier to engage in. For example, an increasing number of contributors pay attention to diversity and inclusion as an important aspect in the community dimension of open source project health. Contributors can learn from healthy open source projects with high code-quality standards and improve their job market opportunities.

Conclusion

Project health is an important topic for many open source stakeholders. Open source projects, organizations, foundations, and contributors need to look for ways to better tell open source project health stories that will help stakeholders form an accurate picture of the health of an open source project. The CHAOSS project is an important collaboration for the creation of a shared understanding of open source project health. It provides many resources to understand open source project health and is a vibrant community where project health is discussed, defined, and measured. The CHAOSS project releases project health standards in the form of metrics definitions, creates tooling to measure metrics, and creates community reports to understand project health. CHAOSScast, the CHAOSS project podcast, is a great source of inspiration because the community shares use cases and experiences that are highly contextualized for specific open source projects. As a member of an open source community, ask yourself these two questions: (1) how healthy is my project? and (2) how can I tell my project health story? Join us in the CHAOSS project so we can help tell your story.

References

[1] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The Matter of Heartbleed," in *Proceedings of the 2014 Internet Measurement Conference (IMC '14),* pp. 475–488: https://doi.org/10.1145/2663716.2663755.

[2] N. Eghbal, *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure* (Ford Foundation, 2016): https://www.fordfoundation.org/media/2976/roads-and-bridges-the-unseen-labor-behind-our-digital-infrastructure.pdf.

[3] E. Weise and N. Bomey, "Equifax had patch 2 months before hack and didn't install it, security group says," *USA Today,* September 14, 2017: https://www.usatoday.com/story/money/2017/09/14/equifax-identity-theft-hackers-apache-struts/665100001/.

[4] D. Naparat, P. Finnegan, and M. Cahalane, "Healthy Community and Healthy Commons: 'Opensourcing' as a Sustainable Model of Software Production," *Australasian Journal of Information Systems,* vol. 19 (2015): https://doi.org/10.3127/ajis.v19i0.1221.

[5] G. J. P. Link and M. Germonprez, "Assessing Open Source Project Health," in *Proceedings of the 24th Americas Conference on Information Systems (AMCIS 2018)*: http://aisel.aisnet.org/amcis2018/Openness/Presentations/5.

[6] C. M. Schweik and R. C. English, *Internet Success: A Study of Open-Source Software Commons* (MIT Press, 2012).

SRE

Beneath the SURFace
An MRI-like View into the Life of a 21st-Century Datacenter

ALEXANDRU UTA, KRISTIAN LAURSEN, ALEXANDRU IOSUP, PAUL MELIS, DAMIAN PODAREANU, AND VALERIU CODREANU

Alexandru Uta is an assistant professor in the computer systems group at LIACS, Leiden University. He received his PhD in 2017 from Vrije Universiteit Amsterdam on topics related to distributed storage systems for scientific workloads. His current research interests are in taming large-scale infrastructure—from designing reproducible experiments to understanding and evaluating performance, as well as designing efficient large-scale computer systems. a.uta@liacs.leidenuniv.nl

Kristian Laursen is pursuing a BSc in computer science at Vrije Universiteit Amsterdam and interns at the SURFsara offices. His professional interests span the field of computer science with a focus on datacenter technologies and distributed systems. kristianvalurlaursen@gmail.com

Alexandru Iosup is a full professor at Vrije Universiteit Amsterdam, chairing the research group on massivizing computer systems and SPEC-RG's cloud group. He is a member of the Netherlands Young Royal Academy of Arts and Sciences. He received a PhD in computer science from Technische Universiteit Delft. His distributed systems work received prestigious recognition: the 2016 Netherlands ICT Researcher of the Year, the 2015 Netherlands Teacher of the Year, and several SPEC community awards, including the SPECtacular Award. A.Iosup@vu.nl or @AIosup

Real-world data is crucial in understanding and improving our world, from health care to datacenters. To help the computer systems community with data-driven decisions, we open-source a collection of fine-grained, low-level operational logs from the largest public-sector datacenter in the Netherlands (SURFsara). In this article, we describe the infrastructure providing the data, give examples of some of this data, and perform thorough statistical analysis to indicate that this ongoing collection not only reflects the ground truth but will be useful to designers and maintainers of large clusters, and generally to computer systems practitioners.

Medical professionals employ MRI images to look inside our bodies, thus gaining a deeper understanding of the spread and effects of diseases. Open-source collections [1] of medical images enable building or improving analysis tools and training new professionals. In contrast, for computer systems, we do not yet fully benefit from MRI-like views on datacenters. Open source operational traces are scarce and bereft of low-level metrics. Absent such metrics, large-scale systems experts and infrastructure developers are currently forced to design, implement, and test their systems using unverified, sometimes even unrealistic, assumptions. The operational traces we propose help alleviate this problem. Moreover, low-level details of MRI images also offer clinicians predictive capabilities on the evolution of diseases. Similarly, our operational traces would allow for predictive analysis of systems behavior.

Real-world data can be instrumental in answering detailed questions: How do we know which assumptions regarding large-scale systems are realistic? How do we know that the systems we build are practical? How do we know which metrics are important to assess when analyzing performance? To answer such questions, we need to collect and share operational traces containing real-world, detailed data. The presence of low-level metrics is not only significant, but they also help researchers avoid biases through their variety. To address variety, there exist several types of archives, such as the Parallel Workloads Archive, the Grid Workloads Archive, and the Google or Microsoft logs (the Appendix gives a multi-decade overview). However, such traces mostly focus on higher-level scheduling decisions and high-level, job-based resource utilization (e.g., consumed CPU and memory). Thus, they do not provide vital information to system administrators or researchers analyzing the full-stack or the OS-level operation of datacenters.

The traces we are sharing have the finest granularity of all other open-source traces published so far. In addition to scheduler-level logs, they contain over *100 low-level, server-based metrics, going to the granularity of page faults or bytes transferred through a NIC*. The metrics presented in this article are gathered every *15 seconds* from a GPU cluster at SURFsara, totaling over 300 servers. This cluster includes high-speed networks and storage devices, and it is being used for scientific research in the Netherlands in areas such as physics, chemistry, weather prediction, machine learning, and computer systems.

This archive is a valuable resource for many professionals: software developers, system designers, infrastructure developers, machine learning practitioners, and policy-makers. During 2020, we will release monthly on Zenodo the trace data gathered in the previous 30

Paul Melis is group leader of the visualization group at SURFsara, which supports users of its computing infrastructure with visualization expertise and software development, on topics such as data visualization, 3D modeling and rendering, and virtual reality. He has an MSc in computer science from the University of Twente in the Netherlands, and worked on topics in scientific visualization and VR at the University of Groningen and University of Amsterdam before joining SURFsara in 2009. Paul.Melis@surfsara.nl

Damian Podareanu is a senior consultant in the High Performance Machine Learning Group from SURFsara. He studied mathematics and computer science at the University of Bucharest and the Polytechnic University of Bucharest and artificial intelligence at the University of Groningen. He worked as a software developer and scientific programmer for multiple companies before joining SURFsara in 2016. He is currently involved in several classical machine learning projects: IPCC, Examode, ReaxPro. damian@surfsara.nl

Valeriu Codreanu studied electrical engineering at the Polytechnic University of Bucharest, following up with a PhD in computer architecture at the same institute. Valeriu continued as a researcher at Eindhoven University of Technology and University of Groningen, working on GPU computing, computer vision, and embedded systems. He joined SURFsara in 2014, and in 2016 became PI of an Intel Parallel Computing Center project on scaling up deep learning. Valeriu is currently leading the High Performance Machine Learning Group at SURFsara. valeriu.codreanu@surfsara.nl

days, as FAIR (see https://www.go-fair.org/fair-principles/) open data. In this article, we provide a high-level overview of the metrics and data we gather, and a high-level characterization of the first three months of operation in 2020.

Three Months in the Life of a Datacenter

The SURFsara datacenter is used mostly by researchers from the Netherlands, running workloads in areas such as physics, chemistry, weather forecasting, machine learning, and computer systems. Users run primarily HPC-like workloads and deep-learning training, using combinations of regular CPU-, and multi-GPU-servers. A minority of the workloads run on big-data-like systems.

Figure 1 and Table 1 present a summary of several metrics computed over all the GPU servers in the LISA cluster over three months. The individual data points in Figure 1 represent the maximum value for a given hour over all servers, normalized to the maximum value of that metric for the whole period. Table 1 presents the range of values we encountered. We depict here only 10 metrics out of the 100+ collected. Even this high-level summary can be useful to datacenter engineers. For example, the alternation of the five colors for the metric *GPU Fanspeed* shows that the maximum fan speed for a GPU in the LISA cluster varies significantly during the three months analyzed, suggesting that there are very different levels of load in the system over this period. Engineers have to be alert, especially when the load is extreme, either very high or very low.

Our logs register all the interactions of user workloads with the datacenter itself. They also register maintenance events (e.g., adding or replacing servers—these are events which can be derived from the metrics), and unusual events (e.g. job failures, server failures, reboots). Last,

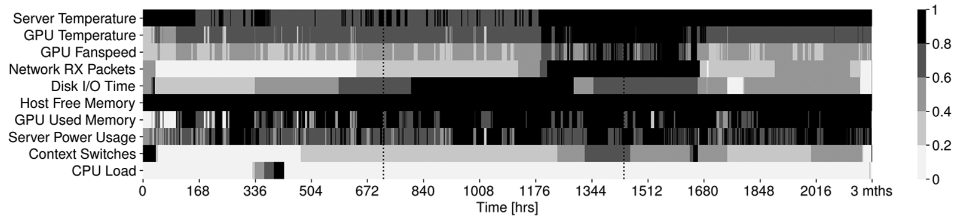


Figure 1: Metric variety and server load variability of the GPU-enabled servers in the LISA cluster over three months (January 1–March 31, 2020). Each data point represents the maximum value a server has encountered for that metric, normalized to the highest encountered value for that metric. The online version of this article shows this heat-map in color.

Metric	Min	Max	Median	Mean	CoV
Server Temperature (Celsius)	24	35	26	26	0.08
GPU Temperature (Celsius)	23	91	31	36	0.38
GPU Fanspeed (Percentage)	0	100	0	8	1.93
Network RX Packets (# packets x 1^6)	0.000003	18	0.460	1	1.73
Disk I/O Time (ms x 1^6)	0.0008	82	9	12	1.01
Host Free Memory (GB)	0.602	268	256	222	0.31
GPU Used Memory (GB)	0	12	0	1	1.96
Server Power Usage (Watt)	0	1400	312	401	0.59
Context Switches (# switches x 1^9)	0.0000052	216	12	23	1.2
CPU Load (Run-queue length)	0	7000	1	12	18.1

Table 1: Value range, median, mean, and coefficient of variation for each metric depicted in Figure 1.

they capture phenomena, such as sudden drops in activity, or low system load. Figure 1 depicts such a phenomenon: at almost all times, the majority of the GPU-enabled servers have their host-CPU underutilized, but simultaneously their GPUs, their networking, and their I/O subsystems experience high utilization. System designers could leverage this empirical observation.

Performing the analysis exemplified in Figure 1 shows that there is ample load variability inside the LISA system. Yet, explaining it is complex. This load variability stems from load imbalances due to varying user-demand, occasionally poor scheduling decisions, and lacking load balancing over the entire set of servers. Only after understanding this complexity can we hope to tame the large design-space for resource management and scheduling decisions in modern datacenters. Moreover, Figure 1 depicts a recent trend we perceive in datacenter operations: GPU-servers are underutilized in terms of CPU load, so here it may be more cost-effective to equip the servers with less powerful (and cheaper) CPUs.

The SURF Archive

Datacenters already exhibit unprecedented scale and are becoming increasingly more complex. Moreover, such computer systems have begun having a significant impact on the environment: for example, training some machine learning models has sizable carbon footprints [2]. As our recent work on modern datacenter networks shows [3], low-level data is key to understanding full-stack operation, including high-level application behavior. We advocate it is time to start using such data more systematically, unlocking its potential in helping us understand how to make (datacenter) systems more efficient. We advocate that our data can contribute to a more holistic approach, looking at how the multitude of these systems work together in a large-scale datacenter.

SURFsara operates several systems inside their datacenter. In this archive, we release operational data from two of SURFsara's largest production clusters: LISA and Cartesius. The former is a 300+ server cluster containing more than 200 GPUs, interconnected with 40-Gbps and 10-Gbps networks. The latter is a 2000+ server cluster containing 132 GPUs and 18 Intel last-generation KNLs. The rest of the servers are a combination of thin (24 cores and 64 GB memory) and fat machines (32 cores and 256 GB memory). The total number of cores in Cartesius is roughly 47K, amounting to 1.8 PFLOPS double precision. Most servers are connected by an FDR InfiniBand network, ensuring 56 Gbps peak bandwidth, with a subset (18 Intel KNL and 177 Intel Broadwell) connected by an EDR InfiniBand network that enables 100 Gbps peak-bandwidth.

We gather metrics, at 15-second intervals, from several data sources:

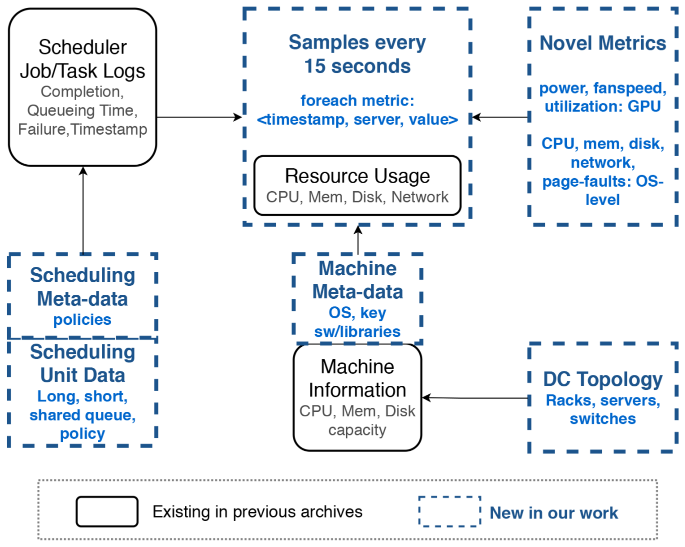


Figure 2: The schema for our collection of datacenter metrics. The figure highlights the novel components we propose, compared to state-of-the-art datacenter archives.

- **Slurm**: all job, task, and scheduler-related data, such as running time, queueing time, failures, servers involved in the execution, organization in partitions, and scheduling policies.
- **NVIDIA NVML**: per GPU, data such as power metrics, temperature, fan speed, or used memory.
- **IPMI**: per server, data such as power metrics and temperature.
- **OS-level**: from either `procfs`, `sockstat`, or `netstat` data: low-level OS metrics, regarding the state of each server, including CPU, disk, memory, network utilization, context switches, and interrupts.

We also release other kinds of novel information, related to datacenter topology and organization.

The audience we envision using these metrics is composed of systems researchers, infrastructure developers and designers, system administrators, and software developers for large-scale infrastructure. The frequency of collecting data is uniquely high for open-source data, which could allow these experts unprecedented views into the operation of a real datacenter.

Our traces will benefit multidisciplinary teams in building better schedulers, better co-locating workloads to improve resource utilization and minimize interference. Recently, systems experts started teaming up also with machine-learning experts to produce AI-enhanced systems such as learned database indexes (work done by Tim Kraska et al.). All these stakeholders could benefit from our many low-level server metrics, which uniquely complement scheduler logs. Uniquely, our traces could help experts to understand how specific workloads interact with the

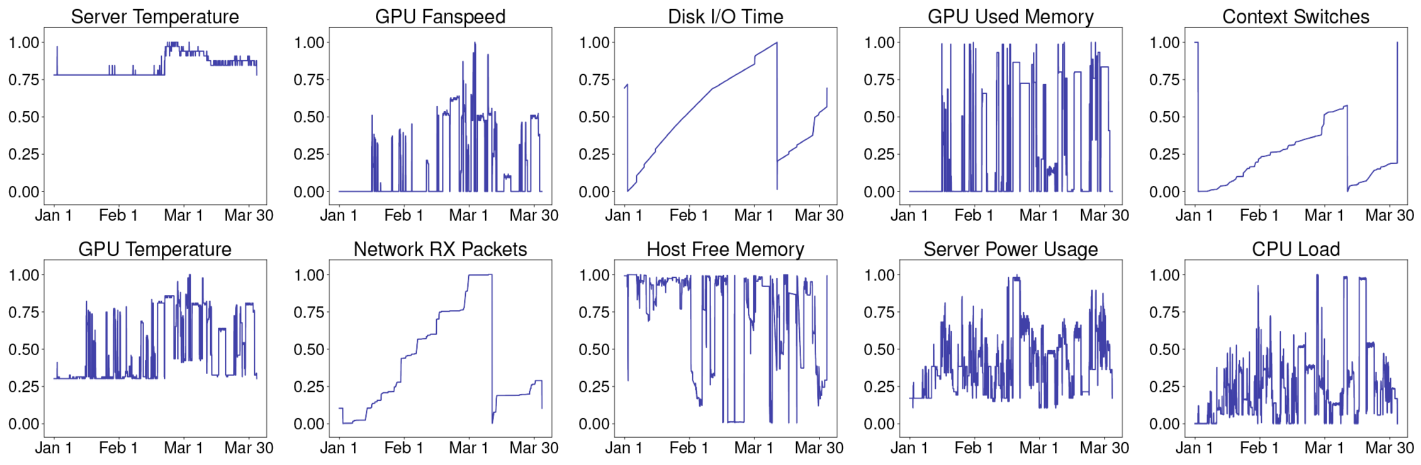


Figure 3: A dashboard to visualize 10 metrics for a single GPU-enabled server in the LISA datacenter. Each metric is normalized by the maximum value encountered during the three months recorded for this server. For all but the "Host Free Memory," higher means more loaded.

hardware, with each other, and where faults and performance issues originate.

Figure 2 presents a high-level view of the schema of the archive we propose. The structure resembles a snowflake schema, with the central *fact* table representing the low-level, high-resolution metrics we collect every 15 seconds from our datacenters. The *dimension* tables represent all other data that we can use to interpret and analyze the fact table. As SURFsara users run jobs, a data set of job-related metrics records scheduler logs (e.g., from Slurm). Simultaneously, many independent tools (e.g., Nvidia Management Layer (NVML)) gather data from each server and push them into the fact table. We keep a separate table containing the list of metrics we collect, enabling easy addition of metrics in the future. Moreover, we explicitly include in the data both server-level and topology information.

Our archive is online: https://doi.org/10.5281/zenodo.3878142.

What Our Archive Offers

There are many types of analyses one could do using the data we open source, such as the typical sysadmin dashboards exemplified by Figure 3. From utilization-level metrics, sysadmins can identify interesting points or correlations that could be examined in more detail, thus improving the daily operation of the datacenter. Using the data in this figure, one could easily correlate temperature increases with, for example, data received over networks, increase in I/O time, and context switches.

Other kinds of analyses are more complex, requiring data science techniques to delve deeper into possible meaning and correlations in our time-series data. Time series in datacenters often display sequential dependencies, meaning the value of a data point is statistically dependent on a previous one. One of the possible steps in analyzing time series is performing regression analysis, which assumes independence of observations. To ascertain the practical usefulness of our data, we perform some basic analytics.

We first investigate whether the time series is linearly correlated to a lagged version of itself, using the Pearson correlation for two independent variables, or, in time series terminology, *autocorrelation*. Figure 4 plots this autocorrelation to provide an insight into the possibility to reduce the amount of data [4]. We use the metric *Server Power Usage* averaged over the GPU-enabled nodes. The confidence interval, depicted in light gray/blue in the figure, lies between -0.2 and 0.2. The figure shows that high correlation values occur for small lags, which is reasonable considering the 15 second sampling frequency.

To further assess the usefulness of the collected metrics, we evaluate a first-order autoregression model, a parametric technique for fitting the observations. As Figure 5 depicts,

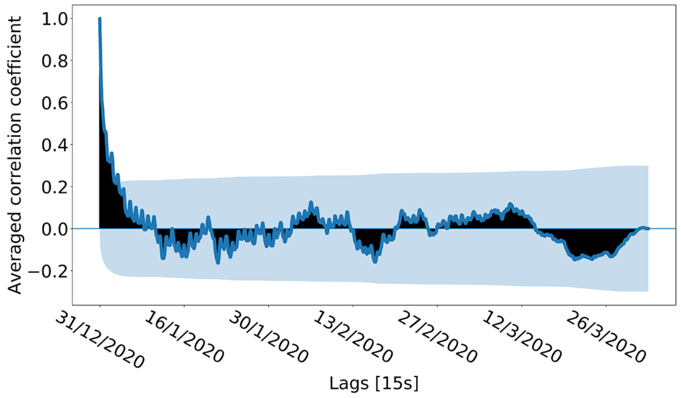


Figure 4: Pearson autocorrelation plot for the server power usage metric. Each point represents a period of 15 seconds. The light gray/blue shaded area represents confidence intervals. The horizontal axis shows 15 second lags, the vertical axis shows correlation values.

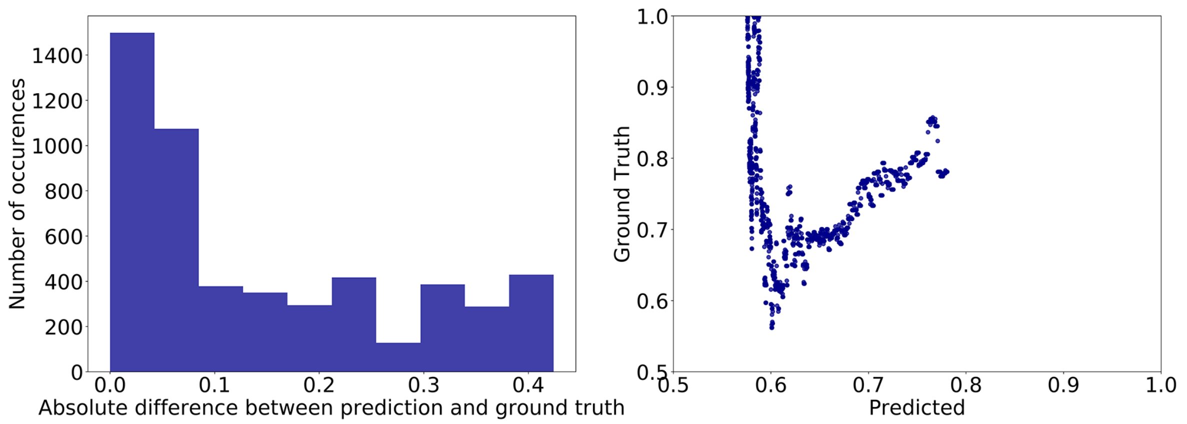


Figure 5: Autoregression histogram (left), predictions vs. ground truth values (right) for the Server Power Usage metric. On the right-hand plot we clip values to [0.5, 1.0], since there are no values below 0.5.

we measure the solution quality by computing the absolute distance between predictions and the ground truth. We chose 518,918 points for fitting the model and tested on 5,242 values, normalized between 0 and 1, by subtracting from each value the minimum and then dividing the result by the difference between maximum and minimum. We did no additional filtering. The autoregression histogram in Figure 5 (left) suggests a reasonable fit for the Server Power Usage metric. However, in Figure 5 (right), we see a possible overfitting behavior when scatter-plotting the predictions against the ground truth. It seems that this simple technique is only capable of predicting the limited interval between 0.6 and 0.8, which is close to the normalized average (0.57), with the whole range being 0.17, 1 for this metric. This is an example of how data scientists could start analyzing our data. More in-depth analyses are certainly possible. We leave this for future work and invite others to run their analyses on the data we open-source.

Conclusion

Realistic assumptions are at the core of building and operating computer systems. Ideally, experts derive these assumptions from data gathered long-term from datacenters in the wild, with the finest of granularities and at the deepest levels of system information. Unfortunately for the computer systems community, only a few organizations currently have access to such data. Existing data sets and trace archives are bereft of such metrics, limiting their ability to support deeper insights.

We offer, as open-source and FAIR data, over 100 low-level metrics gathered at fine granularity from the largest public datacenter in the Netherlands, hosted by SURFsara. In this article, we gave examples and provided an initial analysis over a GPU-enabled cluster inside this datacenter. We showed there are large amounts of variability and imbalances, and correlations between several low-level metrics. Thus, there is value in performing data science analysis over our time-series data. We invite all researchers, practitioners, system designers, and datacenter operators to download and put to good use our open-source archive.

References

[1] R. A. Poldrack, D. M. Barch, J. Mitchell, T. Wager, A. D. Wagner, J. T. Devlin, C. Cumba, O. Koyejo, and M. Milham, "Toward Open Sharing of Task-Based fMRI Data: The OpenfMRI Project," *Frontiers in Neuroinformatics*, vol. 7, no. 12 (July 2013): https://dash.harvard.edu/bitstream/handle/1/11717560/3703526.pdf?sequence=1.

[2] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Deep Learning in NLP," arXiv, June 5, 2019: https://arxiv.org/pdf/1906.02243v1.pdf.

[3] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup, "Is Big Data Performance Reproducible in Modern Cloud Networks?" in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pp. 513–527: https://www.usenix.org/system/files/nsdi20-paper-uta.pdf.

[4] P. M. Broersen, *Automatic Autocorrelation and Spectral Analysis* (Springer Science & Business Media, 2006).

Appendix—On the Elusive Pursuit of Sharing Trace Archives

This work follows in the footsteps of major achievements. The importance of tracing was becoming apparent to the broad systems community at least since the mid-1960s, when instrumentation for collecting operational traces was made available as part of OS/360. By the early 1970s, the systems community was already discussing the importance of using real traces in performance engineering, and by the beginning of the 1990s this practice had already become commonplace.

Until the advent of the Internet, the sharing of traces seemed at best haphazard. The mid-1990s have witnessed the birth of trace archives, with the most prominent being the Internet Trace Archive (ITA, 1995). Focusing on the operation of the Internet, the ITA exhibits many modern features such as data collection and processing tools, and, most importantly, data shared with policies that today would be labeled as FAIR.

Established in the late 1990s, the Parallel Workloads Archive (PWA) [1] is perhaps the most successful example of how shared traces can help shape a community. The PWA started with just a few traces but a good format for sharing, and today it shares traces collected from about 35 environments, mostly from parallel production supercomputers and clusters, but also from research and production grids. Since the mid-2000s, sustained efforts have led to the creation of the Grid Workloads Archive [2] (2006), the Failure Trace Archive [3] (FTA, 2010), the Peer-to-Peer Trace Archive (2010), the Workflow Trace Archive [7] (2019), and the Computer Failure Data Repository, hosted by USENIX.

In the 2010s, the computing industry was transformed by the move to cloud services and by the advent of big data. Unsurprisingly, studies of how such systems operate have led to sharing of characteristics (notably, from Facebook, Yahoo, IBM, Taobao) and, rarely, of traces such as the multi-day trace from a large cluster at Google [4] or Microsoft [6]. Although sharing traces has been very useful for the community, the presence of only one or a few traces cannot account for the tremendous diversity of traces present "in the wild" as reported periodically by analytical studies [5].

References

[1] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with Using the Parallel Workloads Archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10 (October 2014), pp. 2967–2982: https://www.cse.huji.ac.il/~feit/papers/PWA12TR.pdf.

[2] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, and D. H. Epema, "The Grid Workloads Archive," *Future Generation Computer Systems*, vol. 24, no. 7 (July 2008), pp. 672–686: https://doi.org/10.1016/j.future.2008.02.003.

[3] B. Javadi, D. Kondo, A. Iosup, and D. Epema, "The Failure Trace Archive: Enabling the Comparison of Failure Measurements and Models of Distributed Systems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 8 (August 2013), pp. 1208–1223: https://doi.org/10.1016/j.jpdc.2013.04.002.

[4] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4 (March 2010), pp. 34–41: http://pages.cs.wisc.edu/~akella/CS838/F12/838-CloudPapers/Appworkload.pdf.

[5] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the Diversity of Cluster Workloads and Its Impact on Research Results," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18)*, pp. 533–546: https://www.usenix.org/conference/atc18/presentation/amvrosiadis.

[6] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pp. 153–167: https://dl.acm.org/doi/pdf/10.1145/3132747.3132772.

[7] L. Versluis, R. Matha, S. Talluri, T. Hegeman, R. Prodan, E. Deelman, and A. Iosup, "The Workflow Trace Archive: Open-Access Data from Public and Private Computing Infrastructures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9 (May 2020), pp. 2170–2184: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9066946.

Practical Mitigation Guidance for Hardware Vulnerabilities

ANTONIO GÓMEZ-IGLESIAS, EVAN PEREGRINE, AGATA GRUZA, NEELIMA KRISHNAN, AND PAWAN KUMAR GUPTA

Antonio is a software engineer at Intel where he focuses on security software mitigations. He holds a PhD in computer science and has worked in different roles in the areas of performance, computer architecture, parallel programming, and security for the last 15 years.
antonio.gomez.iglesias@intel.com

Evan Peregrine is a software ecosystem engineer at Intel, specializing in long form technical documentation. He contributed to the ACPI specification, Clear Linux, 01.org, and Zephyr Project before joining Intel's software security communications team in 2018.
evan.c.peregrine@intel.com

Agata Gruza has been at Intel for over five years working on performance optimizations of big-data frameworks like Cassandra, Spark, and Hadoop for Intel Architecture. Currently, she is a lead performance engineer and focuses on Linux kernel software mitigation. Agata is a Google (Android Developer) and Facebook AI (Secure and Private AI) scholarship recipient. She holds double MS in computer science and mathematics from Montana State University and The John Paul II Catholic University of Lublin, Poland, respectively. She is an open source contributor and a founder of Women in Big Data NorthWest Chapter. In her free time Agata enjoys hiking and outdoor activities.
agata.gruza@intel.com

Transient execution attack methods and their mitigations have been subject to much scrutiny in recent years. While new hardware platform designs are built to mitigate these methods, existing systems may need to implement microcode or software mitigations. But due to the complexity and variety of these methods, system administrators may wonder what, when, and how to mitigate their systems. We examine common mitigation approaches for the Microarchitectural Data Sampling (MDS) and Transactional Asynchronous Abort (TAA) methods, how these mitigations help prevent attackers from leaking data, how they work to prevent attackers from leaking data, and how sysadmins can configure the mitigations depending on the needs of their environment.

Hardware Vulnerabilities and Transient Execution Methods

In recent years, researchers have demonstrated a novel set of methods known as *transient execution attacks* (TEA, formerly termed *speculative execution side channel*), which target some of the hardware designs introduced in many modern processors, in particular speculative execution. The leading researchers have detailed several variants of this new class of methods that target different hardware components and instructions that execute transiently under various conditions. The hardware industry has responded by issuing microcode updates for affected platforms, developing software techniques to mitigate affected instructions, and changing the designs of new processors. These efforts help ensure that by the time new variants are disclosed, users can protect their systems against potential implementations of these methods. This is a common process that the industry has used to mitigate other hardware issues and errata in the past [1].

Demystifying Microcode

Hardware manufacturers have been using microcode (µcode) since the mid-1990s, among other things to fix bugs found on existing processors. µcode is a way to modify the behavior of hardware without changing the silicon itself by changing how the CPU translates instructions into micro-operations (µops). For example, when a CPU executes x86 instructions, parts of the CPU decode each instruction into a sequence of machine-readable µops that defines what the instruction does. Microcode updates allow hardware manufacturers to modify how particular instructions translate into µops, thereby changing the instruction's behavior.

Software Stack

As seen in Figure 1, there are many different elements in the software stack. Depending on the issue, different components of this stack might change to accommodate new optimization, hardware functionality or to complement µcode changes with additional features. For example:

Neelima Krishnan is a software engineer at Intel. She leads the validation of the security mitigations in the Linux kernel with a special focus on hardware vulnerabilities. neelima.krishnan@intel.com

Pawan Gupta is a software engineer at Intel working on software mitigations for hardware vulnerabilities. He is the author of the TSX Asynchronous Abort mitigation in the Linux kernel. His areas of interest are embedded systems, kernel programming, device drivers, micro-controllers, and real-time operating systems. pawan.kumar.gupta@intel.com

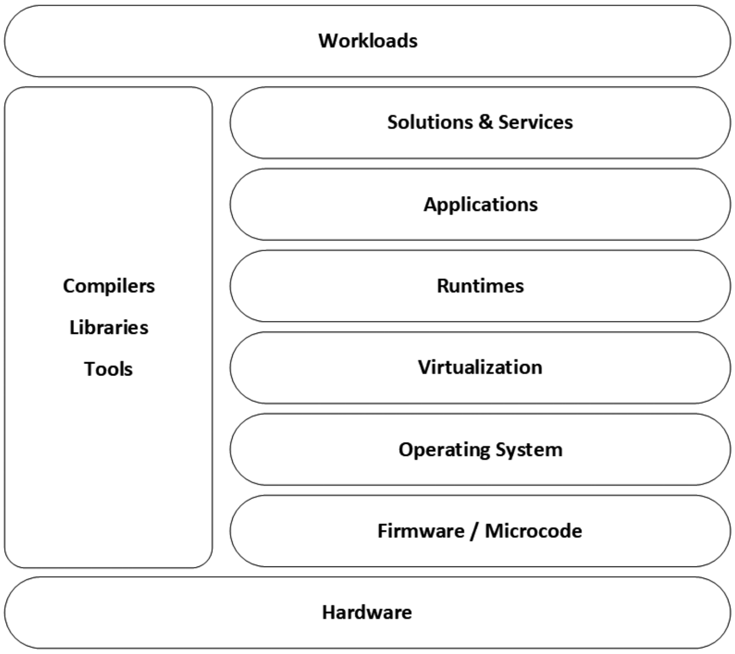


Figure 1: Modern software stack

- The operating system (OS) can include methods that make it more difficult for potential malicious actors to target other processes, other users, or the OS itself.
- Over the years, we have seen how some of these methods target some popular libraries, particularly cryptographic libraries. Popular and well-maintained cryptographic libraries are regularly updated to include programming techniques that make these attacks more difficult. For example, constant-time implementations of crypto algorithms increase their protection against timing methods.
- In certain cases, compilers have introduced changes so that the code generated includes constructs to increase the protection against potential malicious actors. For example, we saw how compilers like gcc included options to protect code against certain Spectre attacks [2].

But the list of software mitigations for these methods does not end here. Software developers regularly update virtual machine managers, web browsers, libraries, tools, and middleware to help mitigate issues originating in hardware [3].

Characteristics of Transient Execution Methods

We focus here on the recently disclosed Microarchitectural Data Sampling (MDS) [4] and Transactional Asynchronous Abort (TAA) [5] methods. In these transient execution attacks (TEA), both the victim (process, kernel, etc.) and the malicious actor must share some physical computing resources. This means that these methods have several inherent restrictions:

- Remote attacks are difficult or not possible. A malicious actor will typically require having local access to a system.
- Any data is accessed in read-only mode. Malicious actors cannot change or roll back a system's data.
- There is no direct privilege escalation. A malicious process cannot give itself root access.
- In some methods, attackers have little or no control over what data they can access. Sophisticated analysis techniques are required to parse secret data out of system noise.
- Both victim and attacker must run on the same physical core.

In addition to these limitations, most TEA share the following procedure:

1. Access target data
2. Send data through a covert channel
3. Receive data from the covert channel
4. Analyze the data for secrets

To demonstrate, consider the following typical scenario: a malicious actor wishes to extract data from a public cloud system where multiple users can access the same machine and run any type of code. In this type of system, an orchestrator or another piece of software will assign a user to a machine according to the user's specified requirements, and the user has little to no control over which machine they are assigned to. The assigned system will typically also be running other users or processes that have been allocated in the same manner, which means a malicious actor has little to no control over which users or processes they can attack. Because these other users and processes can run arbitrary code, a malicious actor needs to work really hard to find a way to force a victim to run a workload that may be of interest to the attacker, and the attacker must also devise a way to infer what code the victim is running. Finally, if the attacker wishes to implement a data sampling method, the malicious process must share those key computational resources for an extended period of time with the victim process to establish certain data access patterns that the attacker can analyze to infer the data that the victim process was using.

Design and Implementation of Mitigations

While there is not a single recipe to follow when mitigating these issues, this section describes the general process used to mitigate MDS and TAA. The mitigations for both issues require changes at the μcode level and the software level and, therefore, are good case studies of the mitigation process for TEA.

Step 1: New Microcode

Let's review an example of how μcode defines how instructions translate into μops executed by the CPU. The MDS and TAA methods try to leak stale data from small microarchitectural buffers inside the CPU, and the mitigations for these methods consist of clearing the affected buffers before their contents can be sent through a covert channel. This raises the question of when and how those buffers are flushed. We cannot clear the buffers in a disorganized fashion, since that could have undesirable effects, such as cross-thread attacks, stalls, or performance implications. One option we do have is to provide a mechanism so software elements higher up in the stack (such as the OS or applications) can decide when to clear the buffers. For that reason, Intel redefined an existing instruction (`VERW`, Verify Segment for Writing) that was deprecated and not in use. On affected systems, after the μcode update, `VERW` can be used to flush and clear the content of the buffers affected by MDS/TAA.

Step 2: How to Invoke the New Functionality Provided by the Microcode (if Required)

Now we have a tool (`VERW`) that software can invoke to clear those buffers. An example of a C function that calls this instruction in the Linux kernel is shown below:

```
static inline void mds_clear_cpu_buffers(void) {
        static const u16 ds = __KERNEL_DS;
        asm volatile("verw %[ds]" : : [ds] "m" (ds) : "cc");
}
```

Listing 1: Linux kernel function to invoke `VERW`

We mentioned the Linux kernel since the OS invokes this functionality. The OS is the only component of the software stack that can protect different users from user-to-user attacks, as well as protect the kernel itself from potential attacks originating in userspace.

Step 3: Implementing the New Functionality

Now that we have a function like `mds_clear_cpu_buffers()`, the next step is to identify the right places to clear the buffers. The most appropriate location to flush the buffers is during a ring transition. Ring transitions occur when the system changes the privilege level at which code can execute. For example, if a user application performs a system call to the kernel, a ring transition from ring 3 (userspace) to ring 0 (kernel space) occurs. To mitigate these issues, the `VERW` instruction is invoked before the system call returns from kernel space back to userspace.

As another example, `VERW` should be invoked if there is a context switch between different processes, regardless of the owner of those processes. This prevents attacks on systems that disable simultaneous multithreading (SMT), since only one process can run on a physical core at any given time, and the buffers are cleared before another process runs in the same core.

Step 4: Options to Configure the Mitigation and Report Mitigation Status

The last step when it comes to reducing the severity of these security issues is to provide mitigation options so that those with the right privileges can configure them at boot time, as well as a mechanism to detect the status of those mitigations. Sysadmins can control mitigations from the kernel command line. A full list of available options based on the hardware vulnerability is available at kernel.org [6]. In most cases, because transient execution attacks require a malicious actor to be able to execute locally on a system, machines that run only known, controlled, and trusted software may be very difficult or even impossible to target with these methods. Also, due to the nature of the code and users they support, certain systems might not be the target of transient execution attacks and may not need the mitigations. For example, if after a detailed risk analysis where the usage of the system and the characteristics of TEA are considered, the sysadmin decides that the risk of TEA is very low, they may choose to disable the mitigations.

In cases where all the userspace applications are trusted and don't execute untrusted code, then mitigations can be disabled. System administrators may want to disable the mitigations on such systems, as different mitigation options can have performance implications. In other cases, when programs have secret data that needs to be protected (for example, crypto keys), the kernel should provide a full mitigation for the same issue. Also, other components of the software stack, like compilers, might also put in place options to enable or disable the mitigations. System administrators should evaluate their environment and workloads and make an educated decision whether security mitigations are needed.

Sysadmins can use simple tools to check if a given system is mitigated against certain CPU vulnerabilities. In Linux, hardware security issues are associated with a report log, which resides in `sysfs`. The output of `sysfs` indicates if a system is affected by a specific method, and whether the system is mitigated or presently vulnerable.

To reflect recently disclosed vulnerabilities the OS needs to be up-to-date, either by updating the existing kernel or upgrading to the most current one. Updating the OS might seem like a daunting task that takes a significant amount of time. To ease the burden, security researchers created system vulnerability checkers, such as a tool for Linux and BSD available at GitHub [7] to detect whether a given machine is affected by TEA methods. Those tools provide detailed information about hardware support for mitigation techniques if a system is vulnerable to TEA, whether vulnerable systems can be mitigated with a µcode or OS update, or if software changes are required.

Other Techniques for Preventing Attacks

We have seen how these methods take advantage of hardware resources that are shared among different processes. It is possible to limit this resource sharing and thereby reduce the attack vector, with some caveats.

The first challenge here is system load. Some systems are configured to run many more processes at any given time than currently available physical cores on the system. In these cases, resource sharing is unavoidable. However, on systems where the total number of user processes doesn't exceed the total number of physical cores, it is possible to schedule processes to always run on the same physical core and never share that core with other processes, thereby reducing the chances for a malicious actor to implement one of these methods. Linux tools like `numactl` and `taskset` can be used to set the affinity of processes and implement this type of process scheduling. Also, `cgroups` can be an alternative to create process isolation.

The open source community is working on a Linux kernel scheduling technique to implement a similar solution. This technique, called *core scheduler*, allows system administrators to tag specific processes. Processes sharing a tag can run simultaneously on the same physical core (when SMT is enabled), while processes with different tags are prevented from running concurrently on the same physical core. When one process stops running, either because the process is finished or because the OS schedules a different process, the hardware resources (such as buffers and branch predictors) are cleared before another process can use them. Other operating systems and virtualization tools might also implement similar techniques.

The second caveat is interruptions. By default, interrupts can run on any core of the system as decided by the OS. So, if that is the case, then interrupts might be a target of a potential malicious actor. Particularly in systems with SMT on, a malicious actor may be able to target the data accessed by the interrupt while this interrupt is executing on the same core. However, system administrators can now choose to specify cores in the system to handle all system interrupts, preventing any user processes from running on those cores [8]. System administrators should carefully consider the implications of this approach (how it affects the overall throughput of the system, the number of system calls that are normally handled, etc.).

Conclusion

While transient execution methods have affected many modern CPUs, the industry has collaborated to ensure that mitigations for these methods are available by the public disclosure date. This requires understanding the implications of these methods and the optimum solution for mitigation. Since new hardware includes mitigations against these methods, an approach might be to update the hardware. However, because changing hardware takes time, is costly and challenging, other alternatives (like updating microcode and making changes to the software stack) are needed to mitigate existing vulnerabilities. It's crucial for system administrators to understand that the technical mitigations are just one component of the security process. Enabling sysadmins to choose what mitigation approach works best for their environment and workloads is key. Providing alternatives, explaining how the different mitigation methods work, and outlining the factors to be considered for each mitigation approach, all help enable system administrators to choose the most appropriate actions.

References

[1] Microsoft, "Host Microcode Update for Intel Processors to Improve the Reliability of Windows Server": https://support.microsoft.com/en-us/help/2970215/host-microcode-update-for-intel-processors-to-improve-the-reliability.

[2] GCC 7.3 release notes: https://lwn.net/Articles/745385/.

[3] Intel, "Deep Dive: Managed Runtime Speculative Execution Side Channel Mitigations": https://software.intel.com/security-software-guidance/insights/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations.

[4] Intel, "Microarchitectural Data Sampling": https://software.intel.com/security-software-guidance/software-guidance/microarchitectural-data-sampling.

[5] Intel, "Intel Transactional Synchronization Extensions Asynchronous Abort": https://software.intel.com/security-software-guidance/software-guidance/intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort.

[6] Linux Kernel, "Hardware Vulnerabilities": https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/.

[7] Spectre & Meltdown checker: https://github.com/speed47/spectre-meltdown-checker.

[8] Linux Kernel, "SMP IRQ Affinity": https://www.kernel.org/doc/Documentation/IRQ-affinity.txt.

Using OpenTrace to Troubleshoot DB Performance

ANATOLY MIKHAYLOV

Anatoly is a keen enthusiast in observability and performance troubleshooting. He works as a Staff SRE engineer at Zendesk in Dublin, Ireland, where he is part of a global team that builds and maintains next generation observability tools for dozens of high-traffic microservices/databases. He contributes to the Zendesk Engineering blog. He is also a runner, an avid hiker, and nature photographer. Before Zendesk, Anatoly worked as a DBA, DevOps and software engineer for over 10 years. mikhailov.anatoly@gmail.com

You cannot fix any of the problems you cannot see. I will outline how the Zendesk SRE team monitors database performance and how you can apply it to your own observability challenges. Our approach considers low-level database performance data, proxy logs, and application performance monitoring (APM) in order to expose the meaningful context behind an individual slow SQL query.

Database performance is central to users' experiences, so having excellent observability is critically important. Many of the observability tools that we build or buy are focused on ensuring optimal customer experience, or determining the extent of customer impact during outages and service degradations. These are challenges that many in the industry experience, and I hope that the work I have done at Zendesk will help you build your own observability dashboards. This approach leads to improved back-end performance and happier customers.

Ideally, what we want is a way to track just the single user request that resulted in bad performance. Imagine being able to complete an incident's root cause analysis that takes seconds rather than minutes or hours. API traffic from a large set of customers can not only be traced to relevant database internal performance metrics at a given time, but also be visualized and presented in a readable format. Why is a given SQL query fast in one case and slow in another? When does database performance degradation lead to an outage and when does it not? Is the query execution plan alone enough to understand and address performance issues?

Over the past year we substantially improved database observability, and this improved overall stability and reliability of the system. We built tools to help engineers see and understand performance issues quicker. This has also helped to prevent outages.

I will go through key elements of the observability stack we have built to create meaningful context around requests, linking SQL queries to APM distributed traces and even proxy log events. In this context the proxy event is the entry point, the time elapsed between when an individual request enters the system and once the response is ready to be sent back. SQL query analysis, proxy log event, and APM tracing are the three key elements. To support and enhance their integration we collect database internal information and link that to the rest of the system. We also collect data from information schema to have information about data-set size, which is very important for profiling SQL queries and understanding how data-set size impacts the overall performance. Each individual element provides information, and their integration helps to traverse from one to another using Open Tracing.

OpenTracing

OpenTracing is a vendor-neutral, cross-language standard for tracing distributed applications. Datadog offers OpenTracing implementations for many APM tracers, including the Ruby on Rails version we use for demo purposes. According to the official documentation (https://opentracing.io/docs/overview/spans/):

> The *span* is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system. Each component of the distributed system contributes a span—a named, timed operation representing a piece of the workflow. Spans can (and generally do) contain *References* to other spans, which allows multiple Spans to be assembled into one complete Trace—a visualization of the life of a request as it moves through a distributed system.

A *trace_id* is the unique identifier we propagate from one service to another in order to keep the context. While it can be relatively easy to connect APM application requests with a proxy log event, it's much more difficult to propagate a trace_id to other services like the database process list; I will show how we use SQL comments to do so. We can reuse this approach to connect a background `cron` task job with a relevant SQL query by generating trace_id outside of the HTTP request life cycle. Any service that communicates to a database can benefit by propagating the necessary context with SQL query and tracing libraries that help to abstract complexity and use higher level objects: span and trace.

Database Observability

According to *High Performance MySQL* (https://www.highperfmysql.com):

> Performance is response time. We measure performance by task and time, not by resource. Performance optimization is the practice of reducing response time as much as possible.

MySQL Performance Schema provides a way to inspect database performance and find out why a SQL query runtime takes longer. Or saying it another way: why an SQL query is slow. This level of instrumentation is critical to address performance issues. MySQL 5.6+ supports the sys schema (https://www.percona.com/blog/2014/11/20/sys-schema-mysql-5-6-5-7/), which is a set of objects that interprets data collected by the Performance Schema in a manageable format. I will describe how we take snapshots of relevant queries from the schema with 15-second resolution and learn how to use tracing to connect SQL queries, including trace_id, with the application traces and proxy logs. This tool will not only give you a great instrument to jump from slow query to proxy logs but will also filter out HTTP requests with high database runtime, and so we will focus on these.

According to *High Performance MySQL:*

> [A] common mistake is to observe a slow query, and then look at the whole server's behavior to try to find what's wrong. If the query is slow, then it's best to measure the query, not the whole server....Because of Amdahl's law, a query that consumes only 5% of total response time can contribute only 5% to overall speedup, no matter how much faster you make it.

According to *Site Reliability Engineering* (https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/):

> Your monitoring system should address two questions: what's broken, and why? The "what's broken" indicates the symptom; the "why" indicates a (possibly intermediate) cause....When pages occur too frequently, employees second-guess, skim, or even ignore incoming alerts, sometimes even ignoring a "real" page that's masked by the noise.

"What" and "Why" have different meanings for DBA and for SRE.

- Relational storage and SRE observability worlds are somewhat disconnected. We have to close the gap between a slow HTTP request and what the DB was doing at that very moment.
- SRE teams view high-volume traffic that often has high cardinality. High-cardinality monitoring tools allow connecting with APM/logs.
- DBA teams focus on database performance and the portion of inefficient SQL queries that make it to the DB slow query log.

Improving Observability with Database Signal

Four golden signals (saturation, latency, traffic, errors) make up a well-known approach in web service monitoring, but how can we apply these signals to database performance? Is there anything unique about database performance?

According to *High Performance MySQL:*

> `Threads_running` tends to be very sensitive to problems, but pretty stable when nothing is wrong. A spike of unusual thread states in SHOW PROCESSLIST is another good indicator....If everything on the server is suffering, and then everything is okay again, then any given query that's slow isn't likely to be the problem.... Pileups typically result in a sudden drop of completions, until the culprit finishes and releases the resource that's blocking the other queries. The other queries will then complete.

We will follow the advice from this book to pick the most important performance metrics:

> The essence of this technique is to capture...[it] at high frequency...and when the problem manifests, look for *spikes* or *notches* in counters such as `Threads_running`, `Threads_connected`, `Questions` and `Queries`.

Each of the key metrics carries the signal. For this purpose we choose very low thresholds as service level indicators: seven threads connected, five threads running, DB runtime below two seconds and queries/second (QPS) not higher than 20. When the threshold is exceeded it indicates a signal (1); otherwise, there's absence of the signal (0). We will use the bitmask OR operation to calculate the resulting database signal (Table 1).

Bit Signal SLI		
0001	Threads connected	7 sec
0010	Threads running	5 sec
0100	Database runtime	2 sec
1000	Database queries per second	20

Table 1: Bitmasks for signaling exceeded SLIs

Each individual SQL query will be instrumented with an SQL comment that contains a unique identifier trace_id—for example:

```
SELECT * from users /* 1541859401495831 */
```

For visualization purpose we use Datadog and its APM (https://docs.datadoghq.com/tracing/connect_logs_and_traces/Logs_integration):

> The correlation between Datadog APM and Datadog Log Management is improved by the injection of trace IDs, span IDs, `env`, `service`, and `version` as attributes in your logs. With these fields you can find the exact logs associated with a specific service and version, or all logs correlated to an observed trace (https://docs.datadoghq.com/tracing/visualization/#trace).

Full Circle Observability

Disclaimer: code snippets shared below are open source (MIT license) and are not used at Zendesk but are created exclusively for this article for the purpose of illustration.

A tracing library automatically generates a trace_id on the application side. When the trace_id is generated we propagate context via the HTTP header to the downstream and upstream dependencies, so when the two services communicate to each other, the HTTP header `X-Trace-ID` is the key element needed to bring the context up the stack, from the application to the proxy layer (see Figure 1). In the Ruby on Rails application, the simplified version of the middleware looks as follows:

```
class DdtraceMiddleware
  def call(env)
    result = @app.call(env)
    result[1]['X-Trace-Id']
                        Datadog.tracer.active_span.trace_
id.to_s
    result
  end
end
```

Then the trace_id can be part of Nginx proxy logs, application log, all dependent microservices and external services that were called to serve the original requests. For example, the Nginx access log may appear as follows:

```
log_format    json '{"dd":{"trace_id":"$upstream_http_x_
trace_id"}}'
```

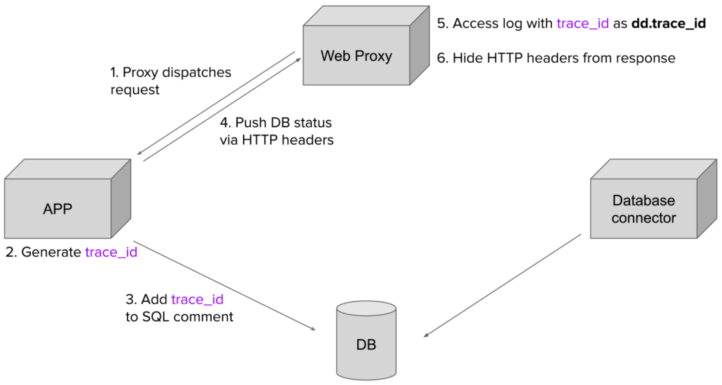

Web Proxy
5. Access log with trace_id as dd.trace_id
6. Hide HTTP headers from response
1. Proxy dispatches request
4. Push DB status via HTTP headers
APP
2. Generate trace_id
Database connector
3. Add trace_id to SQL comment
DB

Figure 1: The trace_id gets added by the application and pushed back upstream to the web proxy and downstream to the database.

We can bring more information from the application up to the proxy layer, store it in `access_log` for observability purposes, and then remove the service information from the HTTP response. For example, if we collect the information about DB runtime, connected and running threads, as well as QPS and calculated Database signal, then the proxy configuration will look as follows:

```
log_format    json '{'
  '"dd":{'
    '"trace_id":"$upstream_http_x_trace_id"'
  '},'
 '"http":{'
    '"performance":{'
      '"queueing_delay":$upstream_http_x_queueing_delay_
digits,'
      '"total_runtime":$upstream_http_x_total_runtime_
digits,'
      '"db_runtime":$upstream_http_x_db_runtime_digits,'
      '"db_signal":$upstream_http_x_db_signal_digits,'
      '"db_threads_running":
               $upstream_http_x_db_threads_running_digits,'
      '"db_threads_connected":
               $upstream_http_x_db_threads_connected_
digits,'
      '"db_qps":$upstream_http_x_db_qps_digits'
    '}'
  '}'
'}';
```

Database signal calculation can be another middleware layer with the following code:

```
def get_db_signal
  db_threads_connected_slo \
             = Thread.current[:db_threads_connected] > 7
  db_threads_running_slo   \
             = Thread.current[:db_threads_running] > 4
  db_runtime_slo           \
             = Thread.current[:db_runtime] > 2
  db_qps_slo               \
             = Thread.current[:db_qps] > 20

  db_threads_connected_bit = db_threads_connected_slo ? 1 :
0
```

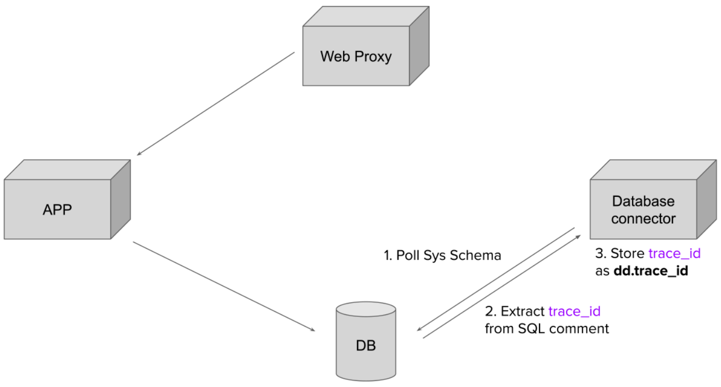

Web Proxy
APP
Database connector
1. Poll Sys Schema
3. Store trace_id as dd.trace_id
2. Extract trace_id from SQL comment
DB

Figure 2: A second way of using the trace_id is for the database connector to query sys schema.

```
    db_threads_running_bit    = db_threads_running_slo ? 2 : 0
    db_runtime_bit            = db_runtime_slo ? 4 : 0
    db_qps_bit                = db_qps_slo ? 8 : 0

    (db_threads_connected_bit | db_threads_running_bit \
        | db_runtime_bit | db_qps_bit).to_s
  end
```

Both the middleware layers and enhanced proxy log configuration help to traverse and debug slow SQL queries in either direction: from proxy to the database process list data and also from process list up to the proxy log. Figure 2 shows a communication between an asynchronous process and the database to collect performance information. Step 1 polls the sys schema process list, extracts the individual SQL query, parses the SQL comment with trace_id, and constructs the JSON event with the `dd.trace_id` identifier. This is a very important step to connect asynchronous data collection with request/response events later on and in being able to create context around slow SQL queries.

Process list aggregation can be done via a bash script:

```
    function process_list_json() {
      trace_id=$(echo "$1" |grep -Eo '/\* [0-9]{16,20} \*/' \
         | awk '{print $2}')

      if [ -z "$trace_id" ]
      then
        echo "{\"mysql\": \"process_list\", \"process_list\":\
  $1}"
      else
        echo "{\"mysql\": \"process_list\", \"process_list\":\
  $1, \
           \"dd\": {\"trace_id\": ${trace_id}}}"
      fi
    }

    function process_list() {
      /usr/bin/mysql -h127.0.0.1 -uroot -s -r -e "SELECT
      JSON_OBJECT(
        'thd_id', thd_id,
        'conn_id', conn_id,
        'command', command,
        'state', state,
```

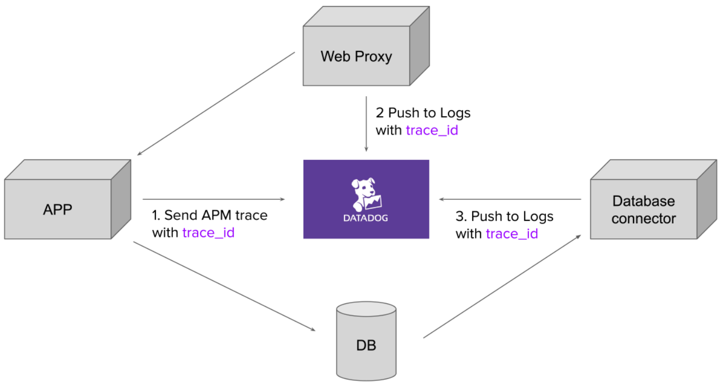

Web Proxy
2 Push to Logs with trace_id
APP
1. Send APM trace with trace_id
DATADOG
3. Push to Logs with trace_id
Database connector
DB

Figure 3: Full observability circle. The unique identifier trace_id gets propagated from the application to database and proxy logs.

```
        'current_statement', current_statement,
        'statement_latency', statement_latency / 1000,
        'progress', progress,
        'lock_latency', lock_latency / 1000,
        'rows_examined', rows_examined,
        'rows_sent', rows_sent,
        'rows_affected', rows_affected,
        'tmp_tables', tmp_tables,
        'tmp_disk_tables', tmp_disk_tables,
        'full_scan', full_scan,
        'last_statement', last_statement,
        'last_statement_latency', last_statement_latency /
  1000
      )
      FROM sys.x\$processlist
      WHERE pid IS NOT NULL
      AND db = 'db'
      LIMIT 25;"
    }

  while true; do
      process_list | while read -r item; do
        process_list_json "$item" "$threads" > /proc/1/fd/1
      done
      sleep 1;
    done
```

This script contains two key functions: `process_list()` to collect SQL queries, and `process_list_json()` to extract trace_id from the SQL comment; it also contains a loop to keep these two functions running once per minute. This script is running in a docker container; output gets redirected to `STDOUT` and is collected by the OpenTracing agent: in this case, the datadog-agent.

An OpenTracing agent receives an APM event from the application and a log event from the proxy and JSON log events. Log events get sent to the log intake endpoint separately. Note: Datadog is used for illustration purposes, but database performance monitoring can be done by any alternative OpenTracing provider.

Conclusion

With comprehensive instrumentation and distributed tracing, we created an observability basis to detect database performance degradation and have the necessary context for further investigation. A database signal can help to address the following questions:

- How much time a database spent processing an SQL query for a given HTTP request
- How saturated the database resources have been during the time of request
- Where the database spent most of the time processing database requests
- How many customers are impacted and what their user experience was

You can find more information with related visualizations at https://medium.com/@unknown_runner.

usenix
SRECON® AMERICAS

DECEMBER 7–9, 2020 • VIRTUAL EVENT

PROGRAM CO-CHAIRS

Nora Jones
Jeli.io

Mike Rembetsy
Bloomberg

The full program and registration will be available soon.

www.usenix.org/srecon20americas

SRE and Justice

LAURA NOLAN

Laura Nolan is an SRE and software engineer by day and a postgraduate student in Ethics at Dublin City University by night. She is a contributor to *Site Reliability Engineering: How Google Runs Production Systems* and *Seeking SRE*, published by O'Reilly, and is a member of the USENIX SREcon Steering Committee. laura.nolan@gmail.com

Not yet at its midpoint, 2020 is already an unforgettable year. This article will appear in the fall edition of *;login:* but was drafted in June, three weeks after the tragic death of George Floyd. The Black Lives Matter movement is at the forefront of current events, eclipsing even the ongoing pandemic. By the time you read this, the news cycle may have changed again (to what is anyone's guess), but right now, Black Lives Matter is at the top of everyone's mind, including mine. I sat down to write this column with the intention of drawing out often overlooked nuances of health-checking in distributed systems, but that will have to wait until a later column. There are more pressing matters at hand.

Black people are incredibly underrepresented in the technology industry, and the percentages have barely moved in the last several years [1]. Black technologists are even more underrepresented when you break out engineering staff from the rest of the business. I don't have statistics, but based on my own experiences in this discipline, site reliability engineering as a sub-field includes very few Black people. None of this is OK.

We have not welcomed Black people into our field, and too many are leaving, or choosing never to enter, because of that [2]. Avoiding use of offensive language such as "master/slave" (use "leader/follower" or "primary/replica") and "whitelist/blacklist" (use "allowlist/denylist" or "blocklist") is table stakes. We in senior roles also need to "give away our legos" [3] to members of underrepresented groups by supporting them through projects that help them grow and by sponsoring them.

Big tech often speaks of diversity as a means to an end, and this is a problem. For example, training sessions intended to reduce unconscious bias usually tell us to value diverse teams because those teams are more effective and creative and therefore better for business. I have always viewed this approach as incredibly dehumanizing. The people who work for any organization, and indeed, those who might aspire to work there, are not commodities. They do not exist as a means to benefit your business or to increase your key performance indicators. People should be treated well (and fairly) simply because they are human beings and intrinsically valuable. It is our obligation and our duty to our Black colleagues. It is a matter of justice.

Justice is a complicated topic, and different thinkers have different approaches to it, but the twentieth-century American philosopher John Rawls's contributions have been the most influential in recent times [3]. Rawls proposes a thought experiment: what if we designed the rules of society from behind a "veil of ignorance," without knowledge of what our eventual social position would be? Rawls thinks that we'd choose two key principles for a just society: the first and overriding principle being civil liberties for all, such as freedom of speech and the right to equal treatment under the law; and the second principle being that the only social and economic equalities that exist should work to the advantage of the least well off—so, for example, a business owner can fairly make more income than average because that business provides affordable services and employment, lifting others.

Black Lives Matter is a call for justice for Black people in their dealings with the police. It is also a matter of justice that Black people deserve to be able to work in the technology industry on an equal basis to anyone else, and to achieve their full career potential. Black people also deserve to have more voice and influence in tech than they currently do, and this is vital as technology now has significant bearing on political issues and on civil liberties.

An incomplete list of the places where justice currently meets technology includes:

- Predictive policing technologies
- Use of automated surveillance and facial recognition technology by authorities (including at protests)
- Targeted political advertising
- Software expert systems in the public realm, including in social welfare decision-making and criminal justice
- Collection, use, and sharing of personal information of all kinds
- Determining credit scores and conducting background checks

Black people in the United States (and in many other countries) have never truly had equal civil liberties in practice. This makes the dearth of Black representation in technology at a time when technology is impacting civil liberties in such profound ways deeply troubling. Shalini Kantayya's new documentary, *Coded Bias*, about Joy Buolamwini's research at the MIT Media Lab on racial bias in AI, discusses how flawed facial recognition technologies disproportionately impact Black people [5]. Cathy O'Neil's Euler prize-winning book *Weapons of Math Destruction* describes many more examples, ranging from the impact of technology on workers' rights to bias in predictive policing technology [6].

I am not suggesting that SRE (or operations-focused engineers in general) can solve all of these problems. However, I do think that we have valuable perspectives on the systems that we work with. For example, we tend to have a broad view of system architectures and a good understanding of what data exists in our systems and how it is managed. We ought to know how reliable and robust our systems are, and if they are fit for purpose. We know whether appropriate security and privacy measures are in place. We have access to metrics and logs. In short, we know a lot about our systems and are thus well positioned to spot many potential ethics concerns. For instance, it's feasible that operations engineers at Facebook could have spotted Cambridge Analytica's excessive API use to harvest personal information in order to influence voters ahead of the 2016 US elections and Brexit referendum.

In recent years we have seen many engineers and technologists speaking out about ethical concerns in the technology industry. This is an important development—vigilant engineers can provide an essential counterbalance to the reduction in transparency, oversight, and accountability that normally goes hand-in-hand with the automation of any process. SREs and other kinds of production-focused engineers have a role to play here, and Black engineers and others from underrepresented groups ought to be part of that.

What are the service level objectives (SLOs) and service level indicators (SLIs) for our democracies and civil liberties, and how do we do our part to uphold them as a profession? With our current demographic makeup, there is no way we can justly answer these questions.

References

[1] S. Harrison, "Five Years of Tech Diversity Reports—and Little Progress," *Wired*, October 1, 2019: https://www.wired.com/story/five-years-tech-diversity-reports-little-progress/.

[2] A. Scott, F. K. Klein, U. Onovakpuri, *Tech Leavers Study* (The Kapor Center, 2017): https://www.kaporcenter.org/tech-leavers/.

[3] T. Reilly, "A Grab-Bag of Advice for Engineers," No Idea Blog, March 3, 2018: https://noidea.dog/blog/a-grab-bag-of-advice-for-engineers.

[4] J. Rawls, *A Theory of Justice* (Belknap Press, 1971).

[5] S. Kantayya, *Coded Bias* (7th Empire Media, 2020): https://www.codedbias.com/.

[6] C. O'Neil, *Weapons of Math Destruction* (Crown Books, 2016).

Systems Notebook
Socially Distant Projects

CORY LUENINGHOENER

Cory Lueninghoener makes big scientific computers do big scientific things, mainly looking at automation, scalability, and large-scale system design. If you don't see him hanging out with the LISA and SREcon crowd, he's probably out exploring the mountains of northern New Mexico. cluening@gmail.com

I don't know about the rest of you, but for me the last several months have been really weird. At the start of March, my daily routine stopped being one that involved getting up, riding my bike to my office, talking to my coworkers, and hopefully getting some technical work done. Instead, I started walking into my garage every morning, sitting at my workbench-become-desk, and interacting with all of my coworkers via WebEx, Skype, BlueJeans, Zoom, and just about any other online meeting package that's ever been invented. While being socially distant has resulted in fewer interruptions, and I feel like I have gotten a lot more done each day, it's also made it clear that projects frequently require socialization to make progress. It turns out that most technical projects benefit from some level of social closeness.

Getting Stuff Done, Together

Let's take a look at a project I have been recently working on, one that started back in the days when we could sit closer than two meters apart from each other. This project, which is still ongoing, is a long-term effort to replace the aging software stack we use to manage many of our scientific computing clusters with something more modern. To say the system management stack that we started with was outdated would be an understatement: one of the main tools we have been using for a long time last had a public release in 2012, and the domain name of the company that was founded to support it was recently for sale—$2999 (CHEAP) and it could be yours! But the stack, which also included Cfengine 2 and SVN, was solid and well known on our production teams, so despite its age making it a liability, there was reluctance to change.

Anybody who has worked in computing long enough has faced the same decision we had to make around a year ago: do we continue dragging our current software stack forward, hoping that it can continue to serve us for a few more generations of systems? Or do we start the long process of updating, knowing that we will face unexpected challenges and potentially introduce instability during the process? While we have faced this question in the past and have always decided to wait a little longer, this time we decided to attack it head on.

Now, to be honest, our environment isn't *that* complex, and this column isn't going to be about the technical details of our solution. I will mention that it involves Git, Ansible, and a yet-to-be-determined provisioning tool, but the work we are doing with them is pretty standard. Standard enough that a motivated team of three or four people could probably have replaced most of the aged components with about six months of solid work. But if those four people had hidden away in their offices for those six months, only eating cheese and pizza that we slid under their doors for them, and they emerged at the end of their metamorphosis with a beautiful new software stack that was perfect in every way, the project would have been a total failure. The problem we had was partly a technical one, but also a social one. We needed to move an entire organization of technical people from one software stack to another, and we needed to do it in a way that respected the fact that some teams wanted to be involved in the development, but a lot of the teams just wanted to be the end users of a stable product.

Cha-cha-cha-cha-changes!

How do you introduce a big change to a big organization? It involves transparency, iteration, and building trust. It involves being social. This starts out all the way at the start of the project, when you need to sell the idea to your immediate coworkers, and continues through selling that same idea to members of other teams, managers, program managers, and everybody else who might be affected by the change. It involves sharing your code, whether that is actual code in Go, Python, or some other language, or it is a set of YAML configuration files. And it involves two-way conversations: presenting your ideas and your code for review, and accepting feedback that others give in return.

We used that recipe to great effect with this project, and we started out small in the beginning. Our initial social circle was just a few of us who had been thinking about the project for a long time, and we started by merging our ideas into an initial project plan. But instead of acting on the plan, creating a new system management stack, and then trying to get others on board, we started out by talking about our plan with our managers and fellow tech leads to make sure we wouldn't create something that would be dead on arrival. Meanwhile, we started a proof of concept where we could try out ideas and incorporate feedback from our colleagues, developing it in an open way that built understanding and trust. Once we knew we had the backing of a sufficient number of stakeholders, we built a small development team with motivated members from each of the teams that needed input, and we started working on the real project.

If you just read that and thought, "Wow, that must have taken a while," then you are totally correct. But by doing a lot of the socialization work up front, we were saving time along the way and preventing failure at the end. As we started the technical work on the project, we knew we needed to find ways to keep the project collaborative. Since the development team was made up of representatives from a variety of other teams, we needed to build ourselves up as a meta-team that could work on this together. How did we do that?

Let's Get Together

To start, we had meetings. No, really! A well-managed meeting is a *very* effective way to share information with multiple people at once. While we were still able to meet in person, we met once a week as a team. Around once a month, we used those meetings as "broadcast" meetings—making announcements, working through administrative details, and generally keeping everybody on the team up-to-date. The rest of the meetings were used for social coding activities: group code reviews, giving presentations on recent work, and triaging development tasks. Two important aspects that made these meetings successful were having agendas and finishing on time. Both of these aspects are based on the same idea: respect others' time. By ensuring we had agendas (and that we stuck to them!), we made it easier for team members to prioritize their time and be ready for the topics that would be discussed that day. By finishing on time, we kept our discussions bounded and didn't steal time from other work.

This model hit a snag in the middle of March, when the world changed and we all started working remotely. No longer were we able to follow our normal routine of getting together weekly to talk about the details of an Ansible deployment. Since our meetings were designed around in-person interaction, we decided to cancel them until things got better. However, as of late May, we recognized that we would likely be working under social distancing restrictions for a longer term than initially anticipated, and as I am writing this (June 2020) we are starting to spin the project back up. Luckily, we had another way to work collaboratively at a distance waiting in the wings.

Enter GitLab

Very early in the process, we had started hosting our work on a local GitLab instance. While our existing system management stack was backed by SVN and we used a separate issue tracking system for our day-to-day work, we recognized early on that adopting an integrated repository browser, issue system, and code review system would provide a new level of insight into our initial coding project as well as the changes that were happening in our systems.

Git has spawned a variety of collaboration tools, from full-featured services like GitHub to locally hosted tools like Gitea. In between is GitLab, which can be used as a remote service or hosted locally. All of these tools promote working on projects in the open, and all of them follow the same general concept of a "merge request" workflow: to make a change, you create a branch, make your changes, push the branch up for review, and then merge the results into the master code branch. These tools provide tight integration with an internal issue tracking system and a web-based front end, providing a great deal of transparency into a team's development process. In our case, GitLab most closely met our needs, and we enthusiastically embraced its use.

As we have started spinning this project back up, we have begun using GitLab's integrated features in earnest. Our weekly in-person meetings have moved online, and we now use GitLab as the main driver of our meetings. Whereas we had previously used a separate meeting agenda to decide on discussion topics, we now use our existing tasks and issues to drive the meetings. While we have replaced the meeting room projector with a shared WebEx screen, more people tend to interact with GitLab on their laptops during the meetings than before. The tooling has stayed the same, but the way we use it to interact with our code and with each other has changed to meet our new needs.

But Wait, There's More!

One final note about the benefits of making a project that is strong both technically and socially: an unexpected outcome of this effort was finding other teams that were starting down the same path on similar projects at about the same time we were doing this. We had originally set out to build a repository that could manage scientific computing clusters, but as we socialized our plans, the core team working on this project started picking up members of other teams who wanted to build on our work. We took this into stride as a group, and used the opportunity to make our work more flexible and accessible to more teams.

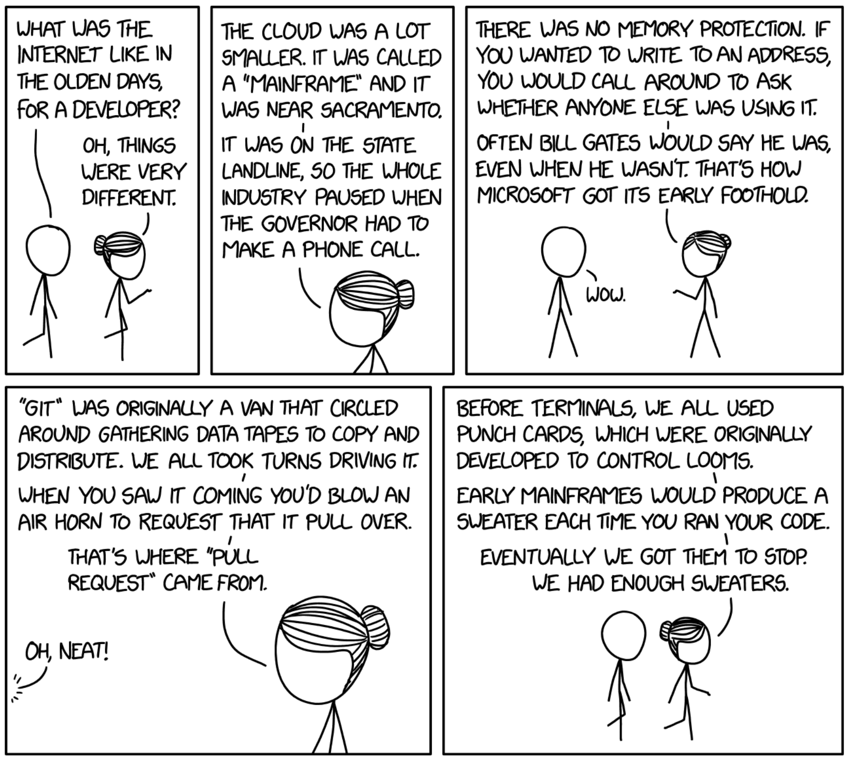
To do this, we split our Ansible repository into two parts. Each individual team has their own Ansible *inventory* directory, which contains their team-specific host definitions, variable definitions, and playbooks. Meanwhile, all teams share an Ansible *roles* directory, which contains reusable building blocks that install and configure things like NTP, `rsyslog`, and authentication in a standard way across our environment. Had we done this project in isolation, none of us would have recognized the utility in splitting the repository out like this until it was much too late to implement it. And by using GitLab as a central collaboration point, we have a very social roles repository that multiple teams can edit and review, but also the flexibility for each team to build their own team-specific work on top of that.

And the Beat Goes on

So where are we currently with this project? As I mentioned at the start, this is an ongoing project, and we are only partway through its implementation. I'm happy we started the project out socially, as it has benefited from that, especially when we had to start doing it remotely. We've begun to start the project up again after we paused it for a while, and as I am writing this, we're just beginning to see how the project will work using text chat, WebEx sessions, and GitLab's integrated tooling. So far, it is very promising. It was a large and sudden change to our workflow, and I don't think it would have worked out as well had we not started out with a social and collaborative approach to this project.

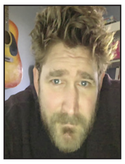
Being socially close despite being physically distant is important beyond this time of isolating ourselves for the sake of society. Most of the USENIX community spends some amount of time working remotely with colleagues, whether they are employees of the same company, salespeople who live in different cities, contributors to open source projects, or any number of other people we benefit from working with without sharing physical space. And as we start migrating back to our normal office life, keeping projects social will help keep them running smoothly, especially when they involve large changes that we need to convince lots of people to make.

XKCD xkcd.com


WHAT WAS THE INTERNET LIKE IN THE OLDEN DAYS, FOR A DEVELOPER?
OH, THINGS WERE VERY DIFFERENT.
THE CLOUD WAS A LOT SMALLER. IT WAS CALLED A "MAINFRAME" AND IT WAS NEAR SACRAMENTO.
IT WAS ON THE STATE LANDLINE, SO THE WHOLE INDUSTRY PAUSED WHEN THE GOVERNOR HAD TO MAKE A PHONE CALL.
THERE WAS NO MEMORY PROTECTION. IF YOU WANTED TO WRITE TO AN ADDRESS, YOU WOULD CALL AROUND TO ASK WHETHER ANYONE ELSE WAS USING IT.
OFTEN BILL GATES WOULD SAY HE WAS, EVEN WHEN HE WASN'T. THAT'S HOW MICROSOFT GOT ITS EARLY FOOTHOLD.
WOW.
"GIT" WAS ORIGINALLY A VAN THAT CIRCLED AROUND GATHERING DATA TAPES TO COPY AND DISTRIBUTE. WE ALL TOOK TURNS DRIVING IT.
WHEN YOU SAW IT COMING YOU'D BLOW AN AIR HORN TO REQUEST THAT IT PULL OVER.
THAT'S WHERE "PULL REQUEST" CAME FROM.
OH, NEAT!
BEFORE TERMINALS, WE ALL USED PUNCH CARDS, WHICH WERE ORIGINALLY DEVELOPED TO CONTROL LOOMS.
EARLY MAINFRAMES WOULD PRODUCE A SWEATER EACH TIME YOU RAN YOUR CODE.
EVENTUALLY WE GOT THEM TO STOP.
WE HAD ENOUGH SWEATERS.

iVoyeur
BPF—Part Three

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Fastly. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

It is a little known fact [1] that as pre-teens Romeo and Juliet, both by nature predisposed to notions of impossible love and emo anti-parental overreaction, independently happened upon and fell in love with Ovid's *Metamorphoses*, wherein is related the tale of the OG suicidal power couple, Pyramus and Thisbe.

Neighbors, whose dwellings were built upon a common center wall in the lovely city of Babylon, Pyramus and Thisbe were cruelly forced apart by their respective families, who shared not only a foundation wall, but also a bitter long-running feud. So close, and yet so far; the phrase itself might have been invented to literally describe their specific predicament, for although their love burned so bright the gods took notice, they might as well have been separated by an ocean.

Until one day, a crack formed low in the basement wall that separated their dwellings. Each noticing separately, and then by degrees stealing into the basement in the night to hear each other whisper their love through the crack in the wall, and to sometimes pass messages and precious tokens of love as opportunity allowed. Eventually they both stabbed themselves. A lion was somehow involved—the precise details escape me, but in probably humanity's earliest example of negative media influence on youth [2], Romeo and Juliet followed in kind some 1500 years later.

Anyway, I know exactly what you're thinking. The basement wall is a textbook perfect metaphor for the memory-enforced separation of kernel space and userspace in monolithic kernel architecture! I know, right? Each side yearning for and depending upon the other?! Each sharing a common heartbeat but never an embrace! Doomed forever to content themselves with whispered secrets and messages passed through cracks in the wall forever holding them apart. Sigh.

Passing Messages

In my last article [3], we took a first look at the `biolatency.py` source code and dove into the kernel source to get a basic understanding of the block I/O layer and what requests at that layer of Linux look like. In this article, as promised, we're going to talk about message passing and the three mechanisms BPF gives us to whisper precious data through the wall between kernel space and our userspace Python runtime. I'll briefly cover all three, though the third and final method is the one we really care about, as it's the one used by biolatency itself.

The first method we have to send ourselves a message-in-a-bottle from kernel space is the `bpf_trace_printk()` function. For an example of its use, consider the BCC tools' one-liner "Hello, World!" program [4]:

```
from bcc import BPF
BPF(text='int kprobe__sys_clone(void *ctx) { bpf_trace_printk("Hello, World!\\n");
return 0; }').trace_print()
```

The C portion of this program attaches to the `sys_clone()` system call and uses `bpf_trace_printk()` to print the string "Hello, World!" to the system "common trace pipe" (`/sys/kernel/debug/tracing/trace_pipe`) whenever a new process is created. On the Python side, we slurp it from the pipe with the `trace_print()` method, which opens the file and prints whatever it finds within [5].

This approach is straightforward and makes for easy one-liner style tool development, but it has a few problems that make it unusable for anything but light testing and one-off tools. Primarily, it's called the "common" trace pipe because it's shared by every BPF filter that uses `bpf_trace_printk()`.

Ignoring the other technical limitations for a moment, this mechanism doesn't even work well with my diligently constructed *metaphor*—more akin to shouting our messages out the window than surreptitiously passing notes in the classroom; making `bpf_trace_printk()` not just a technical but, more importantly, a *literary* non-starter. I think you'll agree, if Romeo and Juliet had to depend on world-readable sockets for message passing, their love would never have survived long enough to result in tragic mutual suicide.

Obviously, to write tasteful trace programs, we'll need a way to get data from our probe without the pollution of a system-common datapath.

Let's therefore abandon `printk` and move on to the second means of data-transfer from a kernel-side probe: `BPF_PERF_OUTPUT()`. This is a ring-buffer of shared memory that contains a pointer to some data that you want to pass from kernel space into your Python program. A proper piece of shared memory, safe from prying eyes. Let's take a look at how the C-side (kernel-side) code uses `BPF_PERF_OUTPUT()`; this snippet is from the `hello_perf_output.py` example in the BCC tools repo [6]:

```
// define output data structure in C
struct data_t {
    u32 pid;
    u64 ts;
    char comm[TASK_COMM_LEN];
};
BPF_PERF_OUTPUT(events);

int hello(struct pt_regs *ctx) {
    struct data_t data = {};

    data.pid = bpf_get_current_pid_tgid();
    data.ts = bpf_ktime_get_ns();
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    events.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
```

Now this is more like it. At the top of this probe, we define `data_t`, an arbitrary data structure whose contents are controlled by us. This is the envelope we will press through the crack in the wall between kernel and userspace. Its secret contents, completely our discretion. In this example, we have three bits of info: the PID of the process that triggered the probe (`pid`), the current system time in nanoseconds (`ts`), and the name of the current process (`comm`).

Each of these three tantalizing intimacies is retrieved by a `bpf_get` function and packed into an instance of `data_t` called, unimaginatively, `data`. There is a small number of these helper-functions [7] available in BPF to retrieve various pieces of context from the kernel at the time the probe was fired. `bpf_ktime_get_ns()` is an extremely common bit of passed data, given that we are almost always timing system calls, or system-call frequency, with BPF. Once packed into our data envelope, we deliver our message with a method call on the `BPF_PERF_OUTPUT` ring buffer, which we've named *events*:

```
events.perf_submit(ctx, &data, sizeof(data));
```

I need to call a quick time out here, before we head back to the Python side, to more closely examine the call to `BPF_PERF_OUTPUT(events);` and talk about variable scope in your C-side probe code. `BPF_PERF_OUTPUT(events);` is the call that creates the ring-buffer we need to pass our `data` struct into userspace (and gives it the name `events`), and I want to explicitly point out *where* in the code it's being called, namely, above our `hello()` function, making it a *globally scoped* variable within the context of our probe. That is, `events` persists between invocations of our `hello()` function, so every time the kernel calls `sys_clone()` and wakes up our probe, the new invocation of `hello()` will reuse the same `BPF_PERF_OUTPUT` instance.

Stated more explicitly, our `hello()` function will only be in scope for a single triggering of a `sys_clone()` system call. It fires and exits with each new process created by the kernel, and then it returns, its context sacrificed to the reallocation gods. This is fine if we just want to blurt a "hello" into the world per invocation, but what if we want to do something more stateful? Like, for example, counting the total number of `sys_clone()` calls throughout the lifetime of our probe's invocation?

The globally scoped `events` ring buffer implies the answer. Because it's scoped outside our `hello()` function, it remains in memory as long as our Python script is running. Hence, if BPF provided something more like a map than a ring buffer (spoiler alert; it does), we could use that map to store data between probe invocations and slurp it up on a timer, or when we catch a keyboard-interrupt on the Python side.

Speaking of the Python side, let's return there now, where we use a blocking call to `perf_buffer_poll()` inside an unbounded loop to check for new data from our `events` ring buffer, like Pyramus constantly slipping downstairs to check for a message from his cherished neighbor. This polling method is called on the top-level BPF object, once we've explicitly opened the ring buffer with `open_perf_buffer()`, the first line of the blurb below:

```
b["events"].open_perf_buffer(print_event)
while 1:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

There are two important things to note about this `open_perf_buffer()` call. The first is its argument, in this case `print_event`; this is a function pointer or "callback." It tells `perf_buffer_poll()` where to send the love letters gleaned from the far side of the wall. The second and more important is how we're dereferencing the `events` ring buffer itself, as a dictionary entry from the top-level BPF object `b["events"]`.

This brings us to the third means we have of smuggling sweet nothings through the wall between our kernel space probe and our userspace Python script: *Maps*. As I implied above, BPF provides myriad Map-like data-structures [8] that you can use to capture stateful information like invocation counts and timings between the system calls captured by your probe. These data structures can all be accessed on the Python side as dictionary values attached to the top-level BPF object, in the same way we're accessing the `events` ring buffer in the code blurb above: `b["events"]`.

Let's take a moment to think about `biolatency.py`'s requirements. From my last article, you'll remember that we're inserting not just one but *two* block I/O layer probes. The first (depending on whether we care about queue-time or not) fires on the `blk_start_request()` system call and invokes our probe's `trace_req_start()` function. The second fires on the kernel's `blk_account_io_done()` and invokes our `trace_req_done()` probe function. In other words, one probe fires when the block I/O event starts, and the other fires when it ends.

Here's the code [9]:

```
if args.queued:
    b.attach_kprobe(event="blk_account_io_start", \
       fn_name="trace_req_start")
else:
    if BPF.get_kprobe_functions(b'blk_start_request'):
        b.attach_kprobe(event="blk_start_request", \
           fn_name="trace_req_start")
    b.attach_kprobe(event="blk_mq_start_request", \
           fn_name="trace_req_start")
b.attach_kprobe(event="blk_account_io_done",
    fn_name="trace_req_done")
```

If you've inferred, without needing to look at the C-side `trace-req` functions, that we're going to be using `bpf_ktime_get_ns()` to capture the "start" system time, and again to capture the "end" system time, and then subtract them to derive an elapsed time from `trace_req_start` to `trace_req_done`, you are absolutely correct. We'll use a globally scoped `BPF_HASH` data structure to store the start times until they can be matched up to their respective "done" events. The invocation to create the hash in the biolatency code looks like this:

```
BPF_HASH(start, struct request *);
```

The map structures provided by BPF are sort of reminiscent of Java generics in that you specify their type as arguments. The first argument in the call above is its name: `start`. The second argument specifies the type of the key value in the hash. Here, we're specifying that the hash will be keyed by a `struct` pointer, literally a number that represents the memory address where a block I/O request struct is stored. This is a pretty clever value for a hash key because it's terse and will always uniquely identify a given I/O request between the start syscall and done syscall. The third argument, which would define the value-type of the hash, is omitted here, so it defaults to a u64, which happens to be exactly the return type of `bpf_ktime_get_ns()`.

This `BPF_HASH` structure is only used to hold the timestamps of each `start` probe firing. It doesn't communicate anything to userspace since its values are set by the start probe and dereferenced by the `done` probe to compute an elapsed time for the I/O request. This means we need another structure to store the elapsed times and communicate these through the wall to the Python side.

You might remember from my first article on eBPF tools [10] that `biolatency.py` presents these values in the form of a histogram, keyed in various ways based on user-provided options (overall summary, per-disk, per I/O-type (read/write etc.)). The use of a histogram here makes a lot of sense because, as you can probably imagine, a busy box may produce a high cardinality of I/O request syscalls. If we tried to shove a note through the wall for every I/O request as we did in the previous examples, we might undermine the wall and send the house collapsing down on top of our heads.

Instead, `biolatency` keeps the data kernel-side, using globally scoped `HISTOGRAM` data-structure to collect the timings computed by our probe's `done` function, as a series of counters within a distributed series of "buckets" representing the range of their values. This is easy on kernel memory (since we're merely storing 64 counters) as well as on the userspace boundary (since we only need to transfer these values once, when we tear down the probe).

Unfortunately, things get a little muddled here since `biolatency.py` needs to use a few different storage back ends and techniques depending on end-user options. Rather than glossing over the interesting details in the space I have left, I will see you in the next issue, where we will take a brief tour of histogram theory, base-two logarithms and the "powers of two rule," and decode `biolatency.py`'s series of substitution choices for the different kinds of block I/O histograms it can depict.

Take it easy.

References

[1] Not a fact. Completely made up.

[2] Romeo and Juliet were imaginary characters who never read Ovid, and that's not how media influence works.

[3] D. Josephsen, "iVoyeur: eBPF Tools," *;login:*, vol. 45, no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/josephsen.

[4] https://github.com/iovisor/bcc/blob/master/examples/hello_world.py.

[5] https://github.com/iovisor/bcc/blob/10603c7123c4b2157190151b63ea846c04c76037/src/python/bcc/__init__.py#L1214.

[6] https://github.com/iovisor/bcc/blob/master/examples/tracing/hello_perf_output.py.

[7] https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#data.

[8] https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md#maps.

[9] https://github.com/iovisor/bcc/blob/master/tools/biolatency.py#L135-L142.

[10] D. Josephsen, "iVoyeur—eBPF Tools: What's in a Name?" *;login:*, vol. 45, no. 1 (Spring 2020): https://www.usenix.org/publications/login/mar20/josephsen.

Programming Workbench

Hand-Over-Hand Locking for Highly Concurrent Collections

TERENCE KELLY

Terence Kelly studied computer science at Princeton and the University of Michigan, followed by a long stint at Hewlett-Packard Laboratories. Kelly now writes code and documentation promoting persistent memory programming and other programming techniques. He usually avoids falling off the monkey bars on the playground by remembering to grab the next bar before letting go of the previous one. His publications are listed at http://ai.eecs.umich.edu/~tpkelly/papers/ and he welcomes feedback at tpkelly@eecs.umich.edu.

Welcome to "Programming Workbench," a new column that will delve into interesting programming problems and solve them with working software. All code is available in machine-readable form at [7]. I welcome feedback from readers, the best of which I may discuss in future columns.

This first installment of "Programming Workbench" reviews a concurrent programming pattern that every developer should know: hand-over-hand locking. Over the past year, I've been surprised more than once to meet well-educated, experienced, proficient programmers who aren't familiar with this versatile and powerful technique. After a bit of digging I began to understand why it's underappreciated: hand-over-hand locking isn't mentioned at all in numerous places where I'd expect a detailed treatment: for example, several Pthreads books and several other books on systems programming in my personal library. A few books mention it without going into great detail [1, 8]. One magazine article discusses the technique at length without providing code [10]. I found only one source with both a detailed discussion and an implementation (in Java) [2].

Why should programmers care about concurrency control in general and hand-over-hand locking in particular? In a word, performance. Even in the bygone age of uniprocessors, multithreaded code made servers more efficient and made interactive software more responsive by overlapping computation with I/O. Today, well-designed *concurrent* software enjoys genuine *parallel* execution on ubiquitous multicore and multiprocessor hardware. Embarrassing parallelism, in which different threads don't interact at all, remains "good work if you can find it"; most multithreaded software, however, isn't so lucky and must orchestrate orderly access to shared memory. Mutex-based concurrency control is the most conventional way to do so, and hand-over-hand locking is a primordial pattern that embodies timeless principles—and sometimes outperforms the alternatives.

So let's brush up on hand-over-hand locking. We'll start with the simplest dynamic data structure, the singly linked list, and review several ways to arrange safe access to linked lists in multithreaded programs. We'll consider hand-over-hand locking in detail, describing its advantages over the alternatives. We'll walk through a working C program whose threads employ the hand-over-hand protocol to access a linked list. Finally, we'll conclude with generalizations and extensions of the basic techniques that we've covered.

Concurrent Lists

A linked list is an easy way to implement the abstraction of an unordered, unindexed, dynamic collection of items. Lists support all of the operations that make sense for such collections: traversing the contents of a collection and inserting, reading, writing, and deleting items along the way. I'd use the word "set" rather than "collection" but in some contexts, e.g., the C++ Standard Template Library, `<set>` confusingly refers to an *ordered* container. Lists are useful in themselves and also as building blocks in more elaborate data structures, e.g., hash tables.

If multiple threads access a collection concurrently, they must avoid data races, which lead to undefined behavior according to the C and C++ language standards. There are several ways to implement a concurrent list safely.

Transactional Memory

Arguably the easiest concurrency control mechanism from the programmer's point of view is transactional memory (TM). TM allows a thread to execute a sequence of instructions atomically and in isolation, preventing other threads from observing intermediate states of the data that the instructions manipulate. A concurrent linked list based on TM avoids data races, and some TM research prototypes would allow genuine parallel access to a linked list, but mainstream TM implementations would effectively *serialize* access to the list. In other words, for the present purpose, off-the-shelf industrial-strength TM-based concurrency control would combine the safety and simplicity of single-threaded code with the performance of single-threaded code, defeating one of the main motives for multithreading.

Non-Blocking Approaches

At the opposite ends of the ergonomic and performance spectra lie non-blocking (lock-free, wait-free) techniques based on the careful use of atomic CPU instructions. The main attraction of non-blocking techniques is that the untimely suspension or death of one thread (due, for example, to a software bug or an unfortunate CPU scheduling decision) doesn't prevent other threads from doing useful work. That's a major advantage compared with mutex-based isolation, which offers no similar guarantee. The main downsides of non-blocking techniques are that they're rather esoteric, to put it mildly—every new contribution is a tour de force by experts—and sometimes they work best with automatic garbage collection. See Michael [6] for a good example of a non-blocking list and Herlihy and Shavit [2] for a broad discussion of non-blocking techniques.

Mutex-Based Isolation

Mutex-based isolation is well understood, and good implementations of POSIX-standardized mutexes have been available for decades. Protecting an entire linked list with a single mutex is easy, but such *coarse-grained* locking serializes access to the list and creates a potential performance bottleneck.

Fine-grained locking for a linked list means associating a mutex with each list node. Per-node locks allow multiple threads to access different parts of the list simultaneously, potentially improving performance. Fine-grained locking, however, isn't guaranteed to be faster, and indeed it can be slower than coarse-grained locking, depending on myriad details beyond the scope of this column. A more worrisome downside of fine-grained locking is that it's just plain trickier than coarse-grained locking; opportunities abound for errors that can cause data races or deadlocks.

It pays to study carefully the correct access discipline, hand-over-hand locking. We'll walk through an implementation, and then we'll reflect on the protocol's properties and benefits.

The Code

The C99/C11 program listed in this section is available at [7]. We'll pore over everything but boilerplate like `#includes`. The purpose of the example program is to emphasize the locking protocol, so it avoids frills for the sake of clarity.

The following `struct` is the building block of our linked list. Each node on the list contains the mutex that protects it, a simple `data` field, and a pointer to the next node on the list.

```
typedef struct node { pthread_mutex_t m;
                      int data;
                      struct node *next; } node_t;
```

For brevity and simplicity we'll just hard-wire a short list into the program. The list consists of a dummy `head` node followed by five "real" nodes, `A` through `E`, whose data fields are respectively initialized to values `1` through `5`:

```
#define           PMI PTHREAD_MUTEX_INITIALIZER
static node_t E = {PMI, 5, NULL},
              D = {PMI, 4, &E},
              C = {PMI, 3, &D},
              B = {PMI, 2, &C},
              A = {PMI, 1, &B},
           head = {PMI, 0, &A};  // dummy node
```

For diagnostic printouts, it's convenient to derive a human-readable name from a pointer to a node. Since our quick-and-dirty program uses a short hardwired list, we can get away with a static mapping of node pointers to name strings. Compared with the alternative of an `if`/`else` statement cascade, the ternary operator (`?:`) saves keystrokes and yields a pure expression:

```
#define NAME(p)  (   &head == (p) ? "head"                  \
                   : &A    == (p) ? "A"                     \
                   : &B    == (p) ? "B"                     \
                   : &C    == (p) ? "C"                     \
                   : &D    == (p) ? "D"                     \
                   : &E    == (p) ? "E"                     \
                   : NULL  == (p) ? "NULL" : (assert(0), "?") )
```

A simple program isn't well served by elaborate, verbose runtime checks, so we use a handful of succinct function-like macros to consolidate error checking. All of our function-like macros expand to *expressions* rather than statements because expressions may appear in a wider range of contexts; later we'll see one in the initialization part of a `for` loop.

If anything unexpected happens, the program falls on its sword via the `DIE()` macro below, which expands to a parenthesized expression that uses the comma operator to evaluate `perror()` and `assert()` for their side effects: `perror()` prints an interpretation of errno; assert() prints the filename and line number where things went wrong and dumps a core file that we may autopsy with a debugger. DIE() appears in contexts like func() && DIE("func"), where func() returns nonzero to indicate failure. The short-circuit property of the && operator ensures that DIE() is evaluated if and only if func() fails.

```
#define DIE(s)      ( perror(s), assert(0), 1 )
#define PT(f, ...)  ( ( errno = pthread_ ## f (__VA_ARGS__)) \
                          && DIE(#f)                          )
```

There's a lot to unpack in the PT() macro above, so we'll walk through it slowly to see how it leverages several C preprocessor features. The problem PT() solves is that several Pthreads functions we use don't *set* the standard errno variable but instead *return* an error number; they return zero to indicate success. PT() allows us to call any of these functions, arranging for errno to be set and DIE() to be called if the function returns nonzero. The easiest way to understand how PT() does its job is to expand a typical use with the compiler's preprocessor ("gcc -E"). For example, expanding PT(join, t[i], &tr); and formatting for clarity yields:

```
(
    (  errno = pthread_join (t[i], &tr)  )
    &&
    (  perror("join"), assert(0), 1      )
);
```

The token-pasting operator ## glues PT()'s first argument, join in the example above, to pthread_. All subsequent arguments to PT() correspond to the ellipsis parameter ("...") in the macro definition, so they get dropped in where __VA_ARGS__ appears in the macro replacement list. In the example above, the last two PT() arguments t[i] and &tr end up as arguments to pthread_join(). The return value of pthread_join() is assigned to the standard errno variable; POSIX defines errno to be a per-thread variable, so there's no data race if two threads call PT() concurrently. The && operator ensures that control reaches the expanded DIE() macro if and only if pthread_join() returns nonzero to indicate failure. Finally, notice in PT()'s definition that parameter f appears a second time in its replacement list, in "DIE(#f)". A single # is the "stringification" operator: in the example above, PT() argument join corresponds to PT() parameter f, so DIE(#f) in PT()'s replacement list expands to DIE("join"), whose expansion places the "join" in perror("join").

Given a pointer to a list node, the LOCK() and UNLK() macros below lock and unlock the mutex embedded in the node. UNLK() also sets the pointer to NULL, which helps to catch a common and insidious bug: dereferencing a pointer to a node after unlocking the node. That would have been a silent data race, but we've turned it into a loud SIGSEGV.

```
#define LOCK(p)        PT(mutex_lock,   (&((p)->m)))
#define UNLK(p)  ((void)PT(mutex_unlock, (&((p)->m))), (p)=NULL)
```

The next macro isn't strictly necessary, but it facilitates testing on my computer. The standard printf() function is thread-safe, but two race detectors that I use, Helgrind and DRD from the Valgrind family of tools, falsely attribute data races to printf(). Protecting printf() with a mutex squelches these false positives. The print mutex can't cause a deadlock because we never try to lock any other mutex while holding it.

```
static pthread_mutex_t   pm = PMI;  // print mutex
#define printf(...)                                  \
  do { PT(mutex_lock,   &pm); printf(__VA_ARGS__); \
       PT(mutex_unlock, &pm); } while (0)
```

Now we're ready for the interesting part: function hoh() below traverses our linked list, observing the hand-over-hand locking protocol. hoh() will be the start routine passed to pthread_create(). Its lone argument will be an identifier string that prefixes each thread's diagnostic printouts. These prefixes make it easy to separate out per-thread reports to see what each thread saw as it traversed our linked list.

```
static void * hoh(void * ID) {
  char *id =     (char *)ID;
  node_t *p, *n;  // "previous" follows "next" down the list
  printf("%s: begin\n", id);
  for (p = &head, LOCK(p); NULL != (n = p->next); p = n) {
    // A:  *p locked & might be dummy head
    //     *n not yet locked & can't be head
    LOCK(n);
    // B:  we may remove *n here
    UNLK(p);
    // C:  best place to inspect *n or insert node after *n
    printf("%s: node %s @ %p data %d\n",
           id, NAME(n), (void *)n, n->data);
    n->data++;
  }
  // D
  sleep(1) && DIE("sleep");  // stall for "convoy" interleaving
  UNLK(p);
  printf("%s: end\n", id);
  return id;
}
```

The for loop of hoh() walks two pointers down the linked list: n ("next") goes first, followed by p ("previous"). The loop initialization locks the dummy head node, and the loop body iterates once per non-dummy node. At comment A, node *p is a locked node whose successor node *n exists; *p might point to the dummy head node—it does on the first iteration—but n never points to the head.

Now comes the "hand-over-hand" aspect: We lock the next node *n while still holding a lock on its predecessor *p. At no point in the for loop is it safe to access *n's successor (*n->next), which is unlocked, but after locking *n we may access pointer n->next, for example, to see if we're at the end of the list by comparing it to NULL. Comment B, where both *p and *n are locked, is the right place to remove *n. We must lock two consecutive nodes to remove the one farther down the list, otherwise concurrent attempts by different threads to remove two adjacent nodes may interfere in such a way that only *one* node is removed [2].

After comment B we unlock *p. At comment C, node *n alone is locked; this is the best place for inspecting or modifying the contents of *n alone because other nodes may access *p simultaneously. We can insert a node after *n here too, but first we should ask why we care *where* to place a new node if the list represents an *unordered* collection—why not simply insert at the head of the list? At comment C we no longer hold a lock on *p so it's no longer safe to read or write *p; as noted above, the UNLK() macro sets *p to NULL to catch careless errors. Our example program prints the name and data field of node n and then increments the data field.

After the for loop terminates at the end of the list, at comment D we gratuitously sleep() while holding a lock on the last list node to produce an interesting "convoy" interleaving of threads.

Inserting a node into the list doesn't require anything like hoh(). Simply lock the head node and splice in the new node after it. If a list represents an *unordered* collection, there's seldom a good reason to insert anywhere else. It's possible to use a list to represent an *ordered* collection by inserting nodes into proper position according to some comparison criterion, but if the collection is large and we must frequently search it to find particular nodes, then a list will be inefficient compared with a search tree or skip list. If an ordered collection is *not* large it might be reasonable to store it as a list, but coarse-grained locking might outperform fine-grained locking.

The main() function below runs hoh() twice single-threaded then spawns several threads that concurrently traverse the list.

```
#define NTHREADS 4
int main(void) {
  pthread_t t[NTHREADS];  int i;  void *tr;
  char m1[] = "1st (serial) traversal",
       m2[] = "2nd (serial) traversal",
       id[NTHREADS][3] = {{"T0"},{"T1"},{"T2"},{"T3"}};
  hoh((void *)m1);
  hoh((void *)m2);
  printf("\nmain: going multi-threaded:\n\n");
  for (i = 0; i < NTHREADS; i++)
    PT(create, &t[i], NULL, hoh, (void *)id[i]);
  for (i = 0; i < NTHREADS; i++) {
    PT(join, t[i], &tr);
    printf("main: joined %s\n", (char *)tr);
  }
  printf("\nmain: all threads finished\n");
  return 0;
}
```

The example code tarball at [7] includes a README containing the commands that I use to compile and run the example program. When the program runs, the interleaving of individual thread outputs reflects the interleaving of the threads themselves as they walk down the list. In the typical output below, thread T1 zooms down the list, then stalls at the sleep(1) call while holding a lock on the last node, E. T2 then gets as far as D, T0 advances to C, and T3 makes it only to B. T1 wakes, releases its lock on E and exits, allowing T2, T0, and T3 to each take a step forward on the list in that order. When T2 exits, T0 and T3 each advance one hop forward. Thus the convoy of threads plods down the list in the manner of an inchworm.

```
T1: begin
T1: node A @ 0x557caa098120 data 3
T1: node B @ 0x557caa0980e0 data 4
T1: node C @ 0x557caa0980a0 data 5
T1: node D @ 0x557caa098060 data 6
T1: node E @ 0x557caa098020 data 7
T2: begin
T2: node A @ 0x557caa098120 data 4
T2: node B @ 0x557caa0980e0 data 5
T2: node C @ 0x557caa0980a0 data 6
T2: node D @ 0x557caa098060 data 7
T0: begin
T0: node A @ 0x557caa098120 data 5
T0: node B @ 0x557caa0980e0 data 6
T0: node C @ 0x557caa0980a0 data 7
T3: begin
T3: node A @ 0x557caa098120 data 6
T3: node B @ 0x557caa0980e0 data 7
T1: end
T2: node E @ 0x557caa098020 data 8
T0: node D @ 0x557caa098060 data 8
T3: node C @ 0x557caa0980a0 data 8
T2: end
T0: node E @ 0x557caa098020 data 9
T3: node D @ 0x557caa098060 data 9
T0: end
T3: node E @ 0x557caa098020 data 10
T3: end
```

Filtering the output (e.g., "./hoh | grep '^T0:'" for thread T0) makes it easier to see what individual threads encountered as they traversed the list:

```
T0: begin
T0: node A @ 0x557caa098120 data 5
T0: node B @ 0x557caa0980e0 data 6
T0: node C @ 0x557caa0980a0 data 7
T0: node D @ 0x557caa098060 data 8
T0: node E @ 0x557caa098020 data 9
T0: end

T1: begin
T1: node A @ 0x557caa098120 data 3
T1: node B @ 0x557caa0980e0 data 4
T1: node C @ 0x557caa0980a0 data 5
T1: node D @ 0x557caa098060 data 6
T1: node E @ 0x557caa098020 data 7
T1: end
```

```
T2: begin
T2: node A @ 0x557caa098120 data 4
T2: node B @ 0x557caa0980e0 data 5
T2: node C @ 0x557caa0980a0 data 6
T2: node D @ 0x557caa098060 data 7
T2: node E @ 0x557caa098020 data 8
T2: end

T3: begin
T3: node A @ 0x557caa098120 data 6
T3: node B @ 0x557caa0980e0 data 7
T3: node C @ 0x557caa0980a0 data 8
T3: node D @ 0x557caa098060 data 9
T3: node E @ 0x557caa098020 data 10
T3: end
```

Each thread saw the list as previous threads left it, *precisely as though the list were protected by a single mutex*.

Properties and Benefits

That's an important attraction of hand-over-hand locking: we get the *parallelism* of fine-grained locking with the simple, sane *semantics* of coarse-grained locking; the changes that one thread makes while traversing the list are, from the viewpoint of all other threads, *atomic*. As the list grows large, at some point fine-grained locking usually begins to improve performance compared with coarse locking, though exactly when depends on the details. Deadlock is impossible because all threads acquire locks in the same order, i.e., list order.

The major limitation of hand-over-hand locking is that threads must traverse the list in one direction only. One implication of this "don't look back" rule is that a thread can't atomically splice a node out of the middle of a long list and splice it back in at the head, which is a bummer, because move-to-front lists offer outstanding performance for some purposes [9]. More generally, hand-over-hand locking doesn't allow us to arbitrarily rearrange a linked list. If we want to rearrange a list with per-node mutexes we can simply lock the head node *and hold that lock* while locking hand-over-hand to the end of the list, thus ensuring that no other threads are accessing any node; then we may alter the list arbitrarily, because effectively we'll be holding a big lock on the entire list.

Generalizations and Extensions

Linked lists are a natural way to implement unordered, unindexed collections. Hash tables implement unordered but *indexed* collections, and search trees implement *ordered and indexed* collections. The techniques we've discussed generalize beyond linked lists to hash tables and search trees: hash tables can represent hash buckets as linked lists, each of which may employ fine-grained locking, and search trees can employ hand-over-hand locking directly.

Unfortunately, the fine-grained locking story for hash tables and search trees isn't as tidy and compelling as that for linked lists. Hash tables invite medium-grained locking—one mutex per hash bucket—which makes more sense than fine-grained locking in the typical case where each bucket contains only a handful of items. Implementing hand-over-hand locking for *balanced* search trees is quite tricky [10].

Persistence

Making a linked list *persistent* is conceptually straightforward: we lay out the list in a file-backed memory mapping with help from a few simple persistent memory programming techniques [3, 4]. Supporting high *concurrency* in a persistent linked list using the techniques discussed above requires "persistence-friendly" mutexes suitable for embedding in persistent data structures, which ordinary `pthread_mutex_t`s aren't. The design of persistence-friendly mutexes is beyond the scope of this column; the main difficulty involves mutex initialization when a program restarts.

If a persistent and highly concurrent linked list must tolerate crashes, for example, because we can't guarantee that the program accessing it will always enjoy an orderly shutdown, we'll need a suitable crash tolerance mechanism. On conventional hardware the right crash tolerance mechanism for persistent memory programming is remarkably easy to implement by leveraging features present in certain file systems [4]. Crash tolerance imposes further requirements on persistence-friendly mutexes: post-crash recovery must quickly and conveniently restore all embedded mutexes to an *unlocked* as well as initialized state. The most onerous requirement on any program that purports to tolerate crashes is that it survive strenuous, realistic tests [5]. Documenting the design, implementation, and testing of persistent, crashproof, and highly concurrent data structures is future work, perhaps for a future installment of this column.

Conclusion

Despite their well-known shortcomings, old-fashioned mutexes will be with us for a long time to come. Even today, conventional mutual exclusion sometimes outshines the alternatives, and fine-grained locking is sometimes the best foundation for high-performance concurrent data structures. Hand-over-hand locking is a conceptually simple protocol for safe multithreaded access to data structures protected by fine-grained locks. The simplest context where fine-grained locking and hand-over-hand traversal make sense is a linked list, and any serious student of concurrent programming should master this primordial pattern.

Readers who want to go further might conduct experiments to explore the tradeoffs in different designs. For a concurrent linked list, when is it faster to use a single mutex on the entire list versus per-node mutexes? Are spinlocks faster than `pthread_mutex_ts`? If a single lock protects the entire list, how much does the move-to-front heuristic [9] help for realistic access patterns? Does it ever pay to maintain list items in sorted order? How do hand-crafted concurrent lists compare to off-the-shelf library implementations of unordered unindexed collections? Please share your results with me!

References

[1] R. Arpaci-Dusseau and A. Arpaci-Dusseau, "Lock-Based Concurrent Data Structures," in *Operating Systems: Three Easy Pieces*, Chapter 29, p. 9: http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf.

[2] W. Herlihy and N. Shavit, *The Art of Multiprocessor Programming* (Morgan Kaufmann, 2008).

[3] T. Kelly, "Persistent Memory Programming on Conventional Hardware," *ACM Queue*, vol. 17, no. 4 (July/August 2019): https://queue.acm.org/detail.cfm?id=3358957.

[4] T. Kelly, "Good Old-Fashioned Persistent Memory," *;login:*, vol. 44, no. 4 (Winter 2019): https://www.usenix.org/publications/login/winter2019/kelly.

[5] T. Kelly, "Is Persistent Memory Persistent?" *ACM Queue*, vol. 18, no. 2 (March/April 2020): https://queue.acm.org/detail.cfm?id=3400902.

[6] M. M. Michael, "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets," in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*: https://docs.rs/crate/crossbeam/0.2.4/source/hash-and-skip.pdf. DOI: https://dl.acm.org/doi/10.1145/564870.564881

[7] T. Kelly, Example code to accompany this article: https://www.usenix.org/sites/default/files/kelly0920_code.tgz.

[8] M. Scott, *Programming Language Pragmatics*, Third Edition (Morgan Kaufmann, 2009). See Exercise 12.14 on p. 642.

[9] D. Sleator and R. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM*, vol. 28, no. 2 (February 1985): https://www.cs.cmu.edu/~sleator/papers/amortized-efficiency.pdf. DOI: https://dl.acm.org/doi/10.1145/564870.564881

[10] H. Sutter, "Choose Concurrency-Friendly Data Structures,"*Dr. Dobb's Journal*, June 27, 2008: http://www.drdobbs.com/parallel/choose-concurrency-friendly-data-structu/208801371.

SIGINFO

Everything Is a Punch Card

SIMSON L. GARFINKEL

Simson L. Garfinkel is a senior computer scientist at the US Census Bureau and a researcher in digital forensics and usability. He recently published *The Computer Book* (Sterling, 2019), a coffee table book about the history of computing. sigmail@simson.net

It's commonly said that in UNIX, "everything is a file."

The meaning of this catchy aphorism is that most UNIX resources can be accessed using names in the file system with a small, consistent set of function calls. So not only can we `open()`, `read()`, and eventually `close()` regular files like `/etc/motd` and `/bin/ls`, we can read the contents of the hard drive (if you have suitable permissions) by opening `/dev/disk0`, the first physical disk on a Macintosh, and even `/dev/mem`, the Linux "device" that lets user processes read system memory.

In this column I'll look at the origins of files and file systems, and contrast the UNIX approach with a subtly different approach that was developed for the Multics operating system, in which files are actually named segments in a two-dimensional memory address space. On Multics, saving a "file" was really creating a named memory segment and then persisting it to long-term storage. Finally, I'll look at how the idea of named persistent memory segments backed by non-volatile memory is making a comeback and will likely be an important part of the storage stack in the near future.

The Historical File

Back in the 1500s a *fyle* was a string or wire used to bind together paper documents, or so reports the *Oxford English Dictionary:* "Thapothecaries shall kepe the billis that they serue, vpon a fyle" (1525). Also spelled *file,* by 1600 the word was used variously to denote the documents in a legal proceeding; a catalog, list, or roll; or even the figurative thread of a person's life.

Put simply, English has had a difficult relationship with the word "file" since the beginning. Sometimes the word refers to the case or container for organizing physical embodiments of information, sometimes it refers to the objects put into that container, and sometimes it refers to the information itself.

Although these days most information that's stored in files is video, when I think of a "file" on my computer, I typically think of a text file. That is, I think of a collection of lines, each somewhere between 1 and 80 characters long, separated by some kind of "end-of-line" character. And for this I have to thank Herman Hollerith and the company he created, The Tabulating Machine Company.

Hollerith graduated from Columbia University in 1879 and took a job working for one of his professors, William P. Trowbridge, who had just taken a temporary assignment working on the 1880 Census in Washington, DC, where he was compiling statistics on power and machinery used in manufacturing. Looking back, this wasn't very surprising: by all accounts Hollerith was a hard-working, brilliant, and ambitious fellow who frequently attracted the mentorship of his older colleagues.

Once in Washington, DC, Hollerith met another future mentor, John Shaw Billings, a surgeon who had become the director of the Army Surgeon General's library after serving in the Civil War. Billings was also working on the Census, where he was in charge of tabulating vital statistics.

The 1880 Census, also known as the 10th Census, was a massive information operation. Census employees collected data from all over the country and brought it to Washington, DC, where it was manually processed according to many different criteria—a process called tabulating—and eventually published. You can think of this processing as a series of SQL SELECT statements with suitable GROUP BY and WHERE clauses. The 10th Census used computers as well, but they were all the human kind [1].

The Card File

Billings suggested that Hollerith create some kind of machine to mechanize the laborious tabulation process. Perhaps Hollerith could build a machine that counted notches on cards of paper, Billings suggested, with each card representing a single person's demographic characteristics, like their age or sex? Hollerith found this idea fascinating and eventually transferred to work under Billings in the vital statistics division just to spend a few months learning the job. When the work on the 10th Census started winding down in 1881, Hollerith moved to Boston, where he had been offered a teaching position at the Massachusetts Institute of Technology.

When he wasn't teaching, Hollerith experimented with ideas for the census machine. Inventing was far more interesting to Hollerith than teaching—he couldn't stand the thought of teaching the same course a second time—so he quit the Institute and took a job back in Washington, DC as a patent examiner. But once he learned the ins-and-outs of the US patent system, he quit *that* job and became a full-time inventor, supporting himself by doing patent work for others.

Hollerith's first census machine patent application described a machine with a long tape of paper and rows of holes representing each person, but Hollerith eventually returned to an idea suggested by Billings. He built prototype machines and, with Billings' help, used them in vital statistics projects in Baltimore and New York City.

The 11th Census had a competition for a machine to assist in the tabulations: Hollerith's machine was one of three tested. Hollerith won the contract, supplied the tabulating equipment for the 11th Census, and eventually incorporated The Tabulating Machine Company in 1896. The company merged with its competitors in 1911 to form the Computing-Tabulating-Recording Company, which was renamed International Business Machines in 1924.

The Hollerith cards used in the 1890 Census had 12 rows of 24 columns and were sized 6⅝″ by 3¾″ so that they fit perfectly inside boxes used to store paper money. When users needed more storage per card, the space between the rows was reduced, allowing the card to hold 45 columns. This still wasn't enough storage, so in 1928 IBM standardized on a card of 7⅜″ by 3¼″ with rectangular holes punched in 80 columns of 12 rows. That was the final standard, and it had lasting influence. The IBM 3270 display terminal introduced in 1971 had an 80-character wide screen, as did the IBM PC introduced in 1981. Indeed, *PEP 8—Style Guide for Python Code*, last revised in August 2013, recommends that source code not exceed 79 characters because some editors wrap when the user tries to edit the last character on an 80-character line.

Older readers may recall receiving punch card checks and utility bills imprinted with the words "do not fold, spindle or mutilate." A *spindle* is a nasty spike pointed straight up and mounted on a weight for holding papers. That is, a spindle is a fyle, and you should avoid using a spindle to file your card file, because the extra hole will be read as an error.

Larger punch cards were used for voter ballots in various parts of the United States until the election of November 2000, after which they were largely replaced due to concerns over their usability and accuracy.

The Circular File

Punch cards were all the rage in information processing for more than a half century. The US Social Security Administration had a master card file sorted by each person's nine-digit social security number. It had another set of punch cards sorted according to the phonetic code of each person's surname. Chrysler had punch cards for its inventory control system. Grades from standardized exams were punched onto cards [2], making it easier for researchers to compute statistics. Really, almost every bit of information that was needed for later processing was stored on punch cards. Even though early computers had magnetic tape, data on tapes was frequently loaded using high-speed punch card readers, and put back onto cards for long-term storage after processing.

In 1956, IBM announced the IBM 305 RAMAC, the Random Access Memory Accounting System. The system's breakthrough technology was the IBM 350, the world's first commercial hard drive. There were 50 metal disks, each with 100 concentric tracks, and a moving read/write-head assembly. The whole thing could store five million 6-bit characters, or 3.75MB. The base system rented for $3,200/month, of which $650 was for the disk storage unit. IBM sold more than a thousand of these vacuum tube-based computers until 1961, when the line was discontinued.

Programming the 305 was complicated: not only was there nothing resembling a modern file system, the program itself had to include pauses to allow the RAMAC's disks to rotate into the appropriate position and for the head to complete any required seek operations. When I downloaded and read the 1957 manual [3], I was most surprised by the matter-of-fact way that IBM described the 305.

It's not a general-purpose computer that has a first-in-the-world megabyte-sized random access memory: it's a system designed for the specific task of helping companies automate inventory, billing, and accounts receivable. That is, it's an electronic punch card file! The big paradigm shift that the manual tries to convey to the reader is that idea that "files are located *in the machine,"* —emphasis in the original—rather than in some external box.

Old paradigms die hard.

On UNIX, Everything Is a File

Modern UNIX and Linux owes much of its flexibility to the way that the operating system handles files and file systems. While other operating systems maintain a different namespace for every physical device, UNIX puts everything into a single hierarchy, a single unified naming system for all files currently accessible.

The second advantage of the "everything is a file" approach manifests when programs running on UNIX get a "file" to open and, lo, it's actually the name of a device. Most UNIX programs will still work, provided that the calling process has the correct authorization to open the file.

This ability to treat devices as files extends to pseudo-devices like `/dev/stdin`, `/dev/stdout`, and `/dev/tty`, which map to `stdin`, `stdout`, and the controlling terminal of the current process. For example, while some programs like `wc` will take their input from `stdin` if no input file is provided, other programs will only take their input from "files." You can give these programs `/dev/stdin` as their input file and then put them into a shell pipeline, like a properly written UNIX program.

Recently, I had a program that decided what file type it was reading by looking at the file's extension. I wanted the program to read its input from a pipe. My solution was to create symbolic in `/tmp` with the appropriate extension, point the link at `/dev/stdin`, and give the program the link for its input. Convoluted, perhaps, but the hack worked the first time.

Another thing that is obviously not a file is memory. Yes, Linux systems have devices like `/dev/mem` and `/dev/kmem` that let programs access memory through the file system, but memory is not file. And although UNIX and Linux have the `mmap()` family of system calls to map files into memory or write blocks of memory out to disk, use of these calls is quite limited. That's an unfortunate result of the UNIX "flat" memory model, in which the program's code, data stack, and any "extra" information can all be accessed using the same pointers.

Because UNIX processes only have access to that single flat address space, files mapped into memory might be mapped into a different location each time a program runs—and it certainly is mapped into different locations when run in different programs. This isn't a problem when code is mapped into memory, as is the case with shared libraries, because most shared code is compiled as position independent code (PIC).

Loading nondeterministically into different regions of memory is a big problem when loading data, however. After all, the whole reason to map a disk file into memory is speed. But if the program can't guarantee where the file is going to land, then the program will need to resort to using indirect memory accesses and various kinds of pointer arithmetic to find every data object. Such approaches are now so well-established that we accept them without much thought, but having to mediate practically every memory reference with pointer arithmetic can have a significant performance impact.

The flat memory space of modern operating systems also has security and reliability implications: many security problems of the last three decades ultimately result from the fact that a `(char *)` pointer in the C programming language can effectively reference any part of the executing program's data, stack, or code.

Multics Files Are Segments

Many of the ideas that make UNIX and Linux great were developed for Multics, the project started in 1965 by MIT's Project MAC, Bell Telephone Laboratories, and General Electric Company's Large Computer Products Division [4]. For example, the very idea of a single tree-structured, hierarchical file system holding all of the system's programs and user files was invented for Multics. Also invented for Multics is the idea that the command processor—the Multics creators called it a shell—would be a normal user program, and that commands would be implemented as programs sitting in the file system rather than making commands a privileged part of the operating system.

Files certainly existed at the time that the Multics project started, as did virtual memory, which was invented in 1962 for the Atlas computer at the University of Manchester. But Multics unified files and memory in a way that was not widely adopted.

On Multics, files are simply pieces of memory that are given names in the hierarchical file system. Multics uses the word *segments* to describe these pieces of memory.

A Multics process might have hundreds of segments mapped into memory at any given time. When segments are mapped into a process context—called loading—the segment's symbolic file system name is mapped to a segment number. Pointers are confined within a segment. Corbató and Vyssotsky's paper from the 1965 Fall Joint Computer Conference [5] describes this as "two-dimensional" addressing. Segments make it easy to provide for the secure sharing of code and data between users, because a single segment can be accessed concurrently by any number of processes, while the underlying hardware controls whether an individual process can read or write to each specific segment.

Multics ran on the GE-645, a computer created for the purpose of running Multics. The actual hardware is somewhat odd by current standards. The GE-645 had a 15-bit segment number and an 18-bit offset to a word within the segment; the underlying system used 36-bit words, divided into four 9-bit "bytes." This machine still runs today, albeit in emulation. You can log into a community Multics system and try it out at https://www.ban.ai/multics/.

Segments neatly circumvent the problems of shared, persistent memory: with two-dimensional pointers (segment and offset), the offset of an individual datum doesn't change when it is mapped out and mapped back in. This means that Multics didn't need to use position independent code, didn't need to relocate code when it was loaded into memory, and allowed code to be shared between executing programs, which meant that only a single copy of each library needed to be loaded into memory—something that wasn't widely available in the UNIX world until the 1990s.

Intel tried to implement segments on the iAPX 432 in 1975, but the project was overly ambitious and ran late. So instead, the company focused on the 8086, a 16-bit version of its successful 8080 microprocessor. Launched in 1978, the 8086 has just four "segment" pointers—the code segment, the data segment, the stack segment, and the "extra" segment—and a 20-bit address is computed by taking a 16-bit segment number, shifting it to the left 4 bits, and adding the offset. That is, segments were a tool for extending memory from 64 KiB to 256 KiB, but not for managing data, shared libraries, or implementing memory protection.

The modern x64 architecture still has these CS, DS, ES, and SS pointers, but they are all set to 0 (zero) to create a flat 64-bit memory space. Now 64 bits is a lot of addressable memory, and we could use some of them for some kind of virtual segment number, but on today's hardware only 48 bits of the address pointers are used: take away 16 bits for a segment number, and that leaves only 32 bits for an offset within a segment. So it might be possible to implement something like Multics segments on modern hardware, but it ultimately won't deliver the same security properties that Multics did because Multics segment/offset pointers simply could not overflow into the next segment. Still, a segmented memory model might be an improvement over what we have today—provided that the segments were large enough.

The Next File

The idea of saving memory in named segments may be coming back into vogue with the advent of so-called storage-class memory (SCM). This memory is a lot like the magnetic core memory of the 1950s and '60s in that it is directly addressable from the CPU and doesn't forget its contents when it is turned off. It's faster than disk and more expensive per byte than disk or flash, but slower and less expensive per byte than DRAM.

One such memory system currently on the market is Optane, manufactured by Micron for Intel. You can buy Optane packaged on a DIMM module or as a PCIe card that looks like a SSD. Plug it into a DIMM slot, and Optane looks like slow memory that doesn't get reset after restart—but be careful, because your system's power-on self-test (POST) might wipe it unless the POST is programmed not to do so. Plug Optane into a PCIe slot, and it looks like an incredibly fast, but small, SSD.

SCM memory is here today, and it might open up a lot of possibilities if people would simply use it. For example, you can buy *today* a server with 24 DIMM slots and give it 12 2-TiB Optane modules, for 24 TiB of non-volatile memory, and 12 128-GiB DDR4 modules, for 1.5 TiB of main memory. You could use such a system to build a massive database server: keep the index and transaction log in the 24 TiB Optane storage, and you won't need to flush the index to disk when the server shuts down and read it into memory when the server starts up. Bailey, Ceze, Gribble, and Levy explored other ideas for using SCM in their 2011 HotOS XIII paper, "Operating System Implications of Fast, Cheap, Non-Volatile Memory"[6]. Meanwhile, Yang, Kim, Hoseinzadeh, Izraelevitz, and Swanson explore the performance of Optane in their FAST '20 paper, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory" [7].

References

[1] For a brief description of the use of human computers in the 10th Census, see https://bit.ly/slg-100. For a more detailed study, see David Alan Grier's excellent book on the subject, *When Computers Were Human* (Princeton University Press, 2005).

[2] To learn more about the use of punch cards in the testing of Indian children, see K. E. Anderson, E. G. Collister, and C. E. Ladd, *The Educational Achievement of Indian Children: A Re-Examination of the Question: How Well Are Indian Children Educated?* (Bureau of Indian Affairs, 1953). https://bit.ly/slg-101.

[3] It's not hard to find IBM RAMAC 305 manuals online. My favorite is http://ed-thelen.org/comp-hist/22-6264-1-IBM-305-RAMAC-ManualOfOperation.pdf.

[4] For more information about Multics, see https://multicians.org/history.html.

[5] F. J. Corbató, and V. A. Vyssotsky, "Introduction and Overview of the Multics System," in *Proceedings of the November 30—December 1, 1965, Fall Joint Computer Conference, Part I*, pp. 185-196: https://www.multicians.org/fjcc1.html.

[6] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating System Implications of Fast, Cheap, Non-Volatile Memory," in *Proceedings of 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011: https://www.usenix.org/legacy/events/hotos11/tech/final_files/Bailey.pdf.

[7] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proceedings of 18th USENIX Conference on File and Storage Technologies (FAST '20)*: https://www.usenix.org/conference/fast20/presentation/yang.

For Good Measure

Security Measurement in the Present Tense

DAN GEER

Dan Geer is the CISO for In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc.
dan@geer.org

Writing a column sometimes requires an "at the time of writing" disclaimer if the situation being described is fluid, de novo, or both. So it is now, which is to say early June, 2020.

By a fluid and de novo situation, I mean the global pandemic known as COVID-19, which is a different beast depending on where you are and how you live. The view from my kitchen table includes a formerly tight lockdown looking soon to be relaxed, pervasive work-at-home for people in technology jobs, a burst of demand for supply chain data, debt burdens too substantial to handle gracefully now or later, and so forth and so on. What might we imagine and, in turn, want to measure under the general topic of "cybersecurity metrics" given the situation? In so many words, here, as in so very much of life, the hard part is getting the questions right—right in the sense of right-for-the-time and supportive of wise decisions. Good questions yield useful answers.

The attack surface comes to mind. I suspect that a material and quite measurable enlargement of the enterprise attack surface due to work at home is hardly a hypothesis. The two components of that expansion that come to my mind as most subtle are routing and sync. How might we measure that expansion, and are estimations on routing and sync the way to go (modeling complex pathways for attack in either case)? Is there a constant of proportionality here and, if so, what might it be? Can its nature be determined by measurement or can measurement merely confirm the assertion of attack surface expansion? Does the fraction of the enterprise's staff working at home reveal a kind of dose-response relationship (a curve of proportionality that demonstrates causality)?

Secondarily, should we expect the changes in the attack surface to show hysteresis? In hysteresis, the output of a system depends not only on its input, but also on its history of past inputs. Put differently, when the force of deformation is relaxed, does the surface spring back to its original position or is the deformation inelastic? Twitter's "work at home forever" comes to mind, but I am thinking more of software installs and changes to standard operating procedure, such as for meetings—installs and changes that won't be de-installed when a COVID-19 vaccine appears. This hysteresis would seem particularly acute in the cybersecurity arena since, as has long been observable, when security products are eclipsed, whether by new organization charts or by new products, existing security products are never de-installed.

The probability of small supplier business failure is surely up. This would imply that the fraction of unmaintained software has risen or will soon rise. Is that measurable? Does the idea of receivership for abandoned software products need measurement (and what kind), or is this just a matter of governmental will? Would measurement help buck up such government will [1]? We'll have this running-but-unsupported situation soon enough once self-modifying code gains autonomy.

What is an "essential" activity and what is not essential is proving to be variously contentious. There's an interesting measure in that, for sure—what fraction of the economy is essential? Rank ordering countries by what fraction of their economy is essential is, likewise, interesting, as would rank ordering cybersecurity functions by whether their operators are deemed essential accordingly. In 2008, we learned a lot about essentialness in and around finance, resulting in an entirely new set of (US) rules for entities that are "too big to fail," or, to be more precise, entities that are SIFIs—systemically important financial institutions. Legalistically, a financial institution is a SIFI if it would pose a serious risk to the economy as a whole were it to collapse. Don't we need that concept in cybersecurity by analogy? Don't we need formal stress testing for computing entities that are not too big to fail but too interconnected to fail [2]? Doesn't cybersecurity eventually, if not now, require such formality? Might we not start now thinking this through?

To the extent that organized opposition (attackers) have an interest in stockpiling 0wned machines, can we measure any uptick in stockpiling in a way that demonstrates causality related to the lockdown crisis? This might be closely related to the routing aspect of attack surface expansion, for example. Or is there a measure that says unequivocally that 0wning a leaf node is nowadays so easy that stockpiling is fiscally irresponsible from the point of view of organized attackers operating as a straight-up business? Or is ransomware now the mechanism of 0wnership? Reported trends in ransomware beg for data on whether the opposition is getting better or the playing field easier to manipulate.

If permanent contraction of enterprises, whether profit or nonprofit, is inevitable, should the cybersecurity workforce enjoy some degree of protection from that contraction? Is there a measure, such as percentage of workforce or percentage of budget, that should be held constant as enterprises contract? (Or, if not that, held above some floor?) Is the skill set among cybersecurity workers a national resource, and do we have numbers to prove it? Is it far-fetched to compare pen-testers adrift in a cybersecurity job collapse to nuclear scientists adrift in the collapse of the USSR? Whatever we've been measuring needs to be re-measured so that trendlines can be established [3].

What about those individuals who were about to enter the cybersecurity workforce, such as recent graduates or those about to be discharged from relevant military service? Do we ensure they find work, or do we have a measure that proves they are unneeded? Are we understaffed with respect to the cybersecurity challenge, overstaffed with respect to the economically provable benefit of cybersecurity practitioners, both, or neither? In many industries, re-opening seems likely to involve replacing humans with algorithms, an ongoing process surely accelerated by the pandemic. Should that be the case in cybersecurity and, regardless of your answer, what might we be measuring here?

In public health, one of the great measurement innovations was the introduction of "quality-adjusted life years" (QALY) as an outcome measure for public health interventions. Do we have some sort of parallel to the QALY measure in cybersecurity? What would it take to have such a measure be defensible as a policy driver? Who should get to set the "adjustment" schedule itself? Also in public health, analyses are often calibrated not just by quality-adjusted life years but also by disability-adjusted life years (DALY, as in disability averted). Is something like DALY more like what we should be measuring in cybersecurity? Or is measurement of either the QALY and DALY sorts built on assumptions that don't actually obtain in cybersecurity? For that matter, where are the tails of distributions getting heavier—the prodromes of black swan events?

In military affairs and emergency management alike, it is all but mandatory that for any given operation or event there be a thorough and dispassionate "after action report" (AAR). Where these are done under a unified command structure such as the Federal Emergency Management Agency [4] or the Department of Homeland Security, their form and scope is itself set by policy. The spirit of the AAR exercise is that of learning lessons from what might realistically be called natural experiments, and formal, fixed output can help make up for the undesigned-ness of any natural experiment. All of which leads us to the question of what should we do in cybersecurity for measuring (and documenting) our version of natural experiments? I would argue that down this path is where we find such things as responsible disclosure mechanisms, bug bounty programs, purposefully opaque software updates, the intermittent appearance of truly novel attacks, and various research results on malware dwell times. Yet to the point here, with lots of cybersecurity AARs to be written in and for the age of pandemics, should we not be measuring and, if so, measuring what, exactly?

One can straightforwardly analogize the "lockdown" strategy as that of decreasing the societal and/or viral attack surface by fiat. I cannot recall as vigorous a purposive reduction in attack surface as the one we saw with COVID-19 (and may, of course, see again should recurrence pick up). On the biologic side, the lockdown was supported by rather an explosion of creative modeling. Take just the one example of wearing a face mask; it protects others from your spew more than it protects you from others'. The benefit of wearing a mask is not transitive, but the risk of not wearing one is (transitive). That's a bit like not allowing your computers to be part of a botnet; it doesn't protect you from others but rather it protects others from you. We need a measure for how much your computing is a danger to others [5], though, of course, such a measure (and the policy it would support) is likely to be met with the same mix of hostile compliance that mandatory face masks exhibits. What should we measure? What should we model? How might we think quantitatively on what sort of cyber pandemic would require turning off electronic commerce until a suite of not yet designed patches (vaccines) could be rolled out to machines young and old alike? Are those countries experimenting with disconnecting from the public Internet [6] on to something measurable?

Health policy and management is perfectly happy (and for good reason) with herd immunity; should we be [7]? What if the exposed fraction of Internet users is largely concentrated in one jurisdiction or among one class of users? Or, as described in the prior reference, how we measure would be correlated with what we conclude is our societal mandate—would we prefer to minimize harm (like reserving scarce vaccine for the young and the old) or would we prefer to minimize transmission (like reserving scarce vaccine for health care workers and undertakers)? Don't answer "both."

In Summary

What I am trying to get at is that what actions we take, at least what considered actions we take, is as influenced by what we measure as it is by what those measurements show. Thinking it out in advance sure beats decision-making under the influence of adrenaline.

I close with a quote from John Foster Dulles, Secretary of State under President Eisenhower:

> The measure of success is not whether you have a tough problem to deal with, but whether it is the same problem you had last year.

References

[1] D. Geer, "On Abandonment": geer.tinho.net/ieee/ieee.sp.geer.1307.pdf.

[2] D. Geer, "For Good Measure: Stress Analysis," *;login:*, vol. 39, no. 6 (December 2014): https://www.usenix.org/system/files/login/articles/login_dec14_13_geer.pdf.

[3] D. Geer, "For Good Measure: Cyberjobsecurity," *;login:*, vol. 45, no. 1 (Spring 2020): https://www.usenix.org/system/files/login/articles/spring20_14_geer.pdf.

[4] For an example after action report, see emilms.fema.gov/IS130a/groups/57.html.

[5] D. Geer, "CyberGreen Metrics," October 2016: www.cybergreen.net/img/medialibrary/CyberGreen%20Metrics%20v.2.pdf.

[6] Russia, December 2019, for example.

[7] D. Larremore, D. Geer, "Progress Is Infectious," *IEEE Security & Privacy*, vol. 10, no. 6 (November 2012): geer.tinho.net/fgm/fgm.geer.1211.pdf.

/dev/random
WFH

ROBERT G. FERRELL

Robert G. Ferrell, author of *The Tol Chronicles*, spends most of his time writing humor, fantasy, and science fiction. rgferrell@gmail.com

I spent some time in medical school in the late 1980s. (They eventually caught me skulking around in the hall and threw me out.) I don't remember coronaviruses—*orthocoronavirinae*, to virologists—being therein addressed as anything serious in terms of human pathology, other than maybe as one of the causes of the common cold. They were mostly associated with birds and bats. That abruptly changed in 2002 with the emergence of the SARS outbreak. Since then, it's just been one bout of coronaviolence after another, culminating in the present day with the imaginatively named Covid-19. (Imagine if they'd named measles "Morbillivid-54.") One consequence of this has been a dramatic increase in non-traditional work environments, especially working from home.

My own first foray into working remotely came in 1998. I did UNIX systems administration and information security, such as it existed then in the federal government, which was in name only. Early on I drove to a local office of another agency in the same department each day, but gradually slid into working from home full-time, doing mostly external penetration testing and incident response statistics for headquarters.

I'd like to emphasize that there is a considerable difference between working remotely and working from home (WFH). While WFH can be truly rewarding, working remotely is a fool's paradise. It has all the drawbacks of going into the office with none of the advantages. You still have to dress appropriately, sit in a sterile office environment, and stock an additional refrigerator and coffee machine. You have to fight traffic while discovering just why it's called a "remote site," worry about other occupants who can't be bothered to stay home when contagiously ill, and put up with bosses who, because they can't observe you physically, drag you into endless teleconferences. They do this when you WFH too, admittedly, but being able to attend while your lower half is wearing only underwear or comfortable track suit pants attenuates the sting.

In case you were trying to sneak in a little actual remote work during your day between inconvenience and onerous oversight, the intrusive, arbitrary policies of the agency or company that manages the building will dash that hope against the rocks of reality. Remodeling, fire drills, inspections, noisy tenants, parking lot repaving, and the nearby cacophony of highway traffic (remote work facilities are almost always in some low-rent industrial park next to the freeway) will guarantee only limited concentration is possible. Ear buds can be a welcome panacea if you're one of the happy few who can work with music going on, but alas I do not count myself among your number. My own brain demands silence in exchange for creativity.

In the late '90s and early aughts, the Office of Personnel Management, that all-purpose HR department for the civilian aspects of the US federal government, began to encourage/cajole federal agencies to allow their employees to work remotely. Toward that end, they either established, or issued guidelines for establishing, remote federal work sites. These were supposed to reduce fuel consumption, consolidate power/office supply costs, and ease the burden of being a faceless bureaucratic cog in a ponderous gargantuan machine, if only by a smidgeon. Mostly, I think, remote work was intended to make it appear as though the federal government was moving toward being a bit more environmentally friendly. Green baby steps, as it were.

I appreciated this initiative at the time because without it, I would not have been able to perform my Reston, VA, duties from all the way down in San Antonio. Having said that, I still feel that the remote work concept was, among other things, an attempt to give employees the illusion of more flexibility while maintaining management's feeling of control. As with the majority of compromises, neither side was really satisfied.

I made the transition to WFH rather seamlessly in the chaotic days following 9/11, because I spent a couple of weeks hanging around in various forums looking for suspicious chatter on behalf of a three-letter agency, and they did not want me coming in over a government network. When that temporary assignment was over, I just never went back to the "remote office," saving myself a 90-mile daily round trip. By that point, my duties had been transferred from Reston to Denver. I'm not certain how long it took before my boss figured out I wasn't reporting to the remote work facility any longer, but I suspect it was quite a while. By that time, I'd settled into my rather productive routines, and she probably figured as long as I was doing the job, the "where" didn't much matter.

The thesis here, in case my long and winding rhetorical road has left you confused, is that working remotely is just another version of going into the office, unlike WFH. Although I can see that certain jobs do not lend themselves well to the WFH paradigm—volcanologist, airline pilot, firefighter, thoracic surgeon, construction crane operator, and so on—WFH is a natural fit for those employed in the purely digital realm. I offer my sympathy and gratitude to people whose commitment to the hands-on life allows the rest of us to sit comfortably in our recliners with a spreadsheet open on one screen and cat videos on the other.

These days I'm a freelance author, admittedly, so for me WFH is more or less a given. I suppose I should call it "WFW," or Working from Wherever (I happen to be). One problem with WFW is that it affords a great many options for entertaining myself in a manner not conducive to, you know, actual *writing*. Sure, back when I worked in an office I could create paperclip sculptures or take-out menu origami, but those pursuits require at least minimal physical participation on my part. Procrastination these days is just a mouse/PS4 controller click away. Write a paragraph, watch an episode or two. Rinse and repeat. I used to write at least 2,000 words a day. Now I'm lucky if I write that much in a week. I can, however, rattle off the filmographies of a couple dozen actors on command. That's got to count for something, right?

Book Reviews

MARK LAMOURINE AND RIK FARROW

Bad Choices: How Algorithms Can Help You Think Smarter and Live Happier

Ali Almossawi
Penguin Random House LLC, 2017, 146 pages
ISBN 978-0-7352-2212-0

Reviewed by Mark Lamourine

In the Summer 2020 issue I reviewed Ali Almossawi's first book, in its online and print editions, *Bad Arguments*. That was a book about logic and logical fallacies, a subject that is always timely. It led me to his second book, *Bad Choices*, about algorithms and how we can use them in daily life. This serves two purposes. The superficial goal is to show how algorithmic thinking can make ordinary tasks more efficient and effective. But the real goal, more subtle and subversive, is to show that the concepts of programming and computation aren't as abstract and alien as they seem when presented in the classroom. We use algorithms every day, and, with just a little attention paid, we can see how pretty much all computation matches problem-solving activities from everyday life.

Almossawi starts each chapter with a little anecdote about a character whose name is a bad pun and who has some task to accomplish. The tasks range from sorting socks to finding all the items on a grocery list while visiting the minimum number of different aisles. He presents each vignette with a statement of the objective and two or three methods of trying to accomplish the task, and then the fun begins.

The veneer of a picture book or a children's story slips away pretty quickly. In the first chapter, he introduces the idea of algorithmic complexity based on the relationship between the size of the job and the time needed to complete the task. He doesn't go deep into the math but presents the growth curves and the concepts of polynomial and logarithmic growth. These are sprinkled through the remaining chapters for comparison. Later chapters cover the ideas behind arrays, associative arrays, hash functions, quick sort, and binary trees, among others.

The first seven chapters of *Bad Choices* are online at https://bookofbadchoices.com/. The first page is even read by Almossawi's son, I think. The presentation is very faithful to the book, down to the graphical page-turn transitions. If you're trying to decide if you want to buy a hard copy for someone, it's an excellent facsimile.

The bad choices of the title are the obvious ways that we do small tasks. For the typical size of daily chores, things like bubble sort or exhaustive search of a clothes rack for the right shirt size pose no real problem. Almossawi uses them to present alternatives and introduce in an informal way the most common algorithms used in computer science. It won't make anyone a programmer, and it won't teach a software developer anything they don't already know. But it might help demystify the idea of algorithms for someone who wants to get more comfortable with computation, and it might even help them sort socks or craft a clever tweet.

The Skeptics' Guide to the Universe: How to Know What's Really Real in a World Increasingly Full of Fake

Steven Novella, with Bob Novella, Cara Santa Maria, Jay Novella, and Evan Bernstein
Grand Central Publishing, 2019, 528 pages
ISBN 978-1-5387-6052-9

Reviewed by Mark Lamourine

I think the authors of *The Skeptics' Guide* must see the irony in the fact that their book is a self-help guide, though I don't expect you'll find it in that section of a book store. In a time of industrial-scale misinformation, the skills needed to evaluate what you see and hear and read must be actively taught and learned. Even more important may be the knowledge of how we as humans can be deceived or deliberately deceive ourselves. You have to want to learn and be willing to let go of what you want to believe if you want to grow.

The Skeptics' Guide to the Universe is the collected wisdom of the hosts of a podcast of the same name. They have been working together since 2005. They created and run the Northeast Conference on Science and Skepticism (NECSS). I have been listening to the podcast for several years, and I admit I am a fan.

The core of the book is the idea of *scientific skepticism*, which is not to be confused or dismissed as philosophical skepticism. The latter is the idea that nothing can be known or trusted. Scientific skepticism is an approach to understanding in which one accepts that learning is possible but that it is a matter of refinement. It is the idea that while one can never achieve certainty, it is possible to approach it in a way that allows one to act in the face of incomplete understanding. The ability to give up a cherished idea in the face of evidence is the most important tenet.

Each of the chapters in the book is fairly short, from two to ten pages at most. They are meant as an introduction to a topic and an invitation to learn more. Each chapter is backed by references that the reader can use to go into a topic in more depth.

Curiously, there is almost nothing in the book telling the reader what *to* believe. In the main section of the book, Novella talks about the ways in which we as humans can mistake the world. First he discusses the realm of illusion and the failure of human intuition. Recent research into the malleability of memory and recall, the mind's ability to see patterns where they don't exist, the meanings attributed to dream and near dream experiences all can inform our response to seeing something strange or apparently inexplicable. It can lead us to question our certainty in our memories and experiences, at least enough to withhold judgment without confirmation.

Novella proceeds to talk about how to think about what we perceive and how we interpret it. This process is known as *metacognition*. It's easy to dismiss the idea of metacognition as "navel gazing," but that's actually the point. Those who would dismiss it would cite what they call common sense. This section is a list of the ways in which "common sense" isn't.

This isn't a way in which "people are stupid." The first chapter in this section is on the Dunning-Kruger effect, but if there's any takeaway it's that this applies to everyone, depending on the topic. A well-educated, intelligent person needs to always be on guard because it is easy for anyone to assume that, since they are expert in one field, they are qualified to evaluate and speak about another. The overriding message of this section is that one needs to be constantly aware of the possibility of being mistaken, especially when you are confident that you are not.

Additional chapters in this section cover motivated reasoning, formal logical fallacies, and some of the more common informal fallacies such as appeal to nature, misinterpreting statistics, or believing coincidence is more than coincidental. In each case, the purpose is to help the reader understand the human tendency toward misperception and how to recognize and correct for it.

In the remainder of this section, Novella covers recognizing the characteristics of pseudo-science and understanding a set of lessons from history. Both groupings contain examples of deliberate hoaxes, honest mistakes, and systemic failures.

The final two sections finally begin to talk about what a reader can do to address misinformation in the media and in life. Today we have access to far more information than we can possibly digest individually. We have to learn to evaluate the sources and our own responses to determine how to use what we get to form a view of the world. The point is never to arrive at certainty, but to create a level of understanding and confidence that allows us to act reasonably. This is always a provisional understanding, and it is assumed that we will continue to learn and refine this view, sometimes even rejecting previously held ideas if new data changes our understanding. Those familiar with Bayesian statistics will be familiar with this idea. When our worldview changes, we can change our behavior to match.

Taken as a whole, *The Skeptics' Guide* is a collection of things to note and to keep in mind when taking in the news of the world and trying to make sense of it for everyday life. It holds a number of cautionary tales, but the message is always one of optimism. It *is* possible to learn and to act reasonably in our society, but it takes some care and self-discipline. It's also possible to recognize *don't care* conditions, where you can let your guard down and relax. Not every topic needs skeptical scrutiny.

This book is not going to convert anyone from a closely held ideology. The nature of human identity means that we don't change who we are easily or quickly. For someone who is confused by the current torrent of input and wants some ideas about how to try to process it without becoming cynical or nihilistic, *The Skeptics' Guide to the Universe* is a great start.

Re-Engineering Legacy Software

Chris Birchall
Manning Publications Co., 2016, 214 pages
ISBN 978-1-61729-250-7

Reviewed by Mark Lamourine

In my experience, software developers are prone to producing crap. It's not all our fault. It's a factor of the limitations on money, time, and sometimes attention and patience. Tasks like writing code and running tests repeatedly aren't the most exciting aspects of coding. Regardless of the reason, there's a lot of code out there that meets Birchall's criteria for "legacy software."

The title of the book belies the scope of Birchall's ambitions. His focus is on refactoring the entire process of software development. I've read and reviewed a number of books that go into depth on the facets of modern software engineering processes. There are books about revision control, automated testing and build processes, and agile planning methods. *The Practice of System and Network Administration* [1] and *Refactoring* [2] are classics, but the first is a general-purpose tome defining the ideals of the industry, and the second is a tightly focused exposition of a neglected facet of the software development process. Each has a place, and both can be daunting to someone looking for an overview that touches on all the needed topics but leaves the details and depth for another time. *Re-Engineering Legacy Software* tries to fill that gap.

In the first half of the book, Birchall does concentrate on the code base and he does start with basic refactoring, but he doesn't stop there. In the next two chapters he expands to reworking software architecture and then again to the considerations of a complete rewrite. He doesn't advocate for either method as a means to reach a more maintainable design. His approach is to look at the factors that would influence the decision to implement either an incremental or bulk replacement of an existing code base. He leaves it to readers to evaluate their own situations.

It is in the second half that he moves on to the design of, not the software, but the software development environment and processes. These are aspects that I think are often either neglected or that slavishly adhere to some ideology that may not take into account the specific needs of the team, the customer, or the project. Birchall discusses the common modern techniques of automated testing, continuous development, and delivery, but with a view to adding them to a project where they are not currently in use. He clearly is an advocate of modern practices, and he brings an agile view to implementing them in existing environments. The focus is again on incremental improvement, not on wholesale replacement, though he does discuss times where that might be the best course.

The final section covers project management and software development culture. Here he echoes in brief the messages of *The DevOps Handbook* [3] and *The Phoenix Project* [4]. These are the classic works on modern software development process and culture. Birchall glosses over the types of cultural and personal pressures that can lead to wasting time on precious features, or alternatively, the mistaken avoidance of writing throwaway code to allow for incremental improvement.

Re-Engineering Legacy Software won't replace any of the old favorites on my book shelf. On the other hand, I would recommend it to someone entering software project management cold or approaching a legacy project for the first time. Birchall makes a subject that can be the focus of ideological wars and pet software tools accessible without a lot of the hype and heat that have been present over the last decade or so. The flip side of "fail fast" is "one bite at a time," and Birchall's book is bite-sized.

References
[1] T. Limoncelli, C. J. Hogan, and S. R. Chalup, *The Practice of System and Network Administration,* 3rd edition (Addison-Wesley Professional, 2016).

[2] M. Fowler, *Refactoring,* 2nd edition (Addison-Wesley Professional, 2018).

[3] G. Kim, J. Humble, P. Dubois, J. Willis, and J. Allspaw, *The DevOps Handbook* (IT Revolution Press, 2016).

[4] K. Behr, G. Spafford, and G. Kim, *The Phoenix Project,* 5th Anniversary edition (IT Revolution Press, 2018).

IT Architect Series: Stories from the Field
Matthew Wood, John Yani Arrasjid, and Mark Gabryjelski
IT Architect Resource, LLC, 2020, 270 pages
ISBN: 978-0-9990929-1-0

Reviewed by Rik Farrow

In the preface of this ebook, John Arrasjid writes that the stories are supposed to be both informative and entertaining. I can agree with John's statement, as I learned things from reading about the misfortunes of others, but also found myself often entertained at the same time.

Stories from the Field begins with a long preface, including a classification scheme for categorizing the stories, using terms like Analysis, Communication, Politics, Database, and Risk. As I just read straight through, the categories really didn't make any difference to me, but at least hinted at what I'd soon be reading.

The stories themselves are written by 35 contributors, presumably all IT architects. I wasn't familiar with "IT architect" as a job description, but learned as I read that this person works with a team to design large scale distributed systems for some business purpose. Most of the team works for a company that does installations, often called a partner but what in the past might have been called a VAR. The team includes people who handle the business side of the project, but also technologists like programmers and network engineers working beside the IT architect.

The stories roughly follow a pattern where the project is described, and this is where I learned about the protocols for designing these projects as well as the systems and software used in current IT departments. There are lots of references to VMware products, and the acronyms used for different types of offerings, like virtual desktop infrastructure (VDI), cloud computing hypervisor (vSphere), and the VMware Enterprise hypervisor (ESXi) took some getting used to. Note that this is not a technical book and is not specific to VMware, but VMware products are often involved in the stories.

Each story ends with lessons learned, and after a while I became familiar with the patterns of failure. Most common were failures in communication that led to misunderstandings, but almost as common were mission creep, although sometimes the *creep* was more like a *leap*, as customers would suddenly decide on installing the just-released version of a major release or have purchased different, and usually cheaper, equipment. There are things that an SRE would find more familiar, such as failure to determine all dependencies until a server fails, and turns out to be the keystone in an entire system.

Overall, I enjoyed reading *Stories*, as the stories themselves are short, informative, and generally fun to read. And, honestly, I felt glad that it was somebody else who had to live through the misadventures.

ENIGMA®

A USENIX CONFERENCE

SECURITY AND PRIVACY IDEAS THAT MATTER

Enigma centers on a single track of engaging talks covering a wide range of topics in security and privacy. Our goal is to clearly explain emerging threats and defenses in the growing intersection of society and technology, and to foster an intelligent and informed conversation within the community and the world. We view diversity as a key enabler for this goal and actively work to ensure that the Enigma community encourages and welcomes participation from all employment sectors, racial and ethnic backgrounds, nationalities, and genders.

Enigma is committed to fostering an open, collaborative, and respectful environment. Enigma and USENIX are also dedicated to open science and open conversations, and all talk media is available to the public after the conference.

PROGRAM CO-CHAIRS

Lea Kissner
Apple

Daniela Oliveira
University of Florida

The full program and registration will be available in November.

enigma.usenix.org

FEB 1–3, 2021

OAKLAND, CA, USA

usenix® THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Writing for *;login:*

We are looking for people with personal experience and expertise who want to share their knowledge by writing. USENIX supports many conferences and workshops, and articles about topics related to any of these subject areas (system administration, programming, SRE, file systems, storage, networking, distributed systems, operating systems, and security) are welcome. We will also publish opinion articles that are relevant to the computer sciences research community, as well as the system adminstrator and SRE communities.

Writing is not easy for most of us. Having your writing rejected, for any reason, is no fun at all. The way to get your articles published in *;login:*, with the least effort on your part and on the part of the staff of *;login:*, is to submit a proposal to login@usenix.org.

PROPOSALS

In the world of publishing, writing a proposal is nothing new. If you plan on writing a book, you need to write one chapter, a proposed table of contents, and the proposal itself and send the package to a book publisher. Writing the entire book first is asking for rejection, unless you are a well-known, popular writer.

;login: proposals are not like paper submission abstracts. We are not asking you to write a draft of the article as the proposal, but instead to describe the article you wish to write. There are some elements that you will want to include in any proposal:

- What's the topic of the article?
- What type of article is it (case study, tutorial, editorial, article based on published paper, etc.)?
- Who is the intended audience (syadmins, programmers, security wonks, network admins, etc.)?
- Why does this article need to be read?
- What, if any, non-text elements (illustrations, code, diagrams, etc.) will be included?
- What is the approximate length of the article?

Start out by answering each of those six questions. In answering the question about length, the limit for articles is about 3,000 words, and we avoid publishing articles longer than six pages. We suggest that you try to keep your article between two and five pages, as this matches the attention span of many people.

The answer to the question about why the article needs to be read is the place to wax enthusiastic. We do not want marketing, but your most eloquent explanation of why this article is important to the readership of *;login:*, which is also the membership of USENIX.

UNACCEPTABLE ARTICLES

;login: will not publish certain articles. These include but are not limited to:

- Previously published articles. A piece that has appeared on your own Web server but has not been posted to USENET or slashdot is not considered to have been published.
- Marketing pieces of any type. We don't accept articles about products. "Marketing" does not include being enthusiastic about a new tool or software that you can download for free, and you are encouraged to write case studies of hardware or software that you helped install and configure, as long as you are not affiliated with or paid by the company you are writing about.
- Personal attacks

FORMAT

The initial reading of your article will be done by people using UNIX systems. Later phases involve Macs, but please send us text/plain formatted documents for the proposal. Send proposals to login@usenix.org.

The final version can be text/plain, text/html, text/markdown, LaTeX, or Microsoft Word/Libre Office. Illustrations should be PDF or EPS if possible. Raster formats (TIFF, PNG, or JPG) are also acceptable, and should be a minimum of 1,200 pixels wide.

DEADLINES

For our publishing deadlines, including the time you can expect to be asked to read proofs of your article, see the online schedule at www.usenix.org/publications/login/publication_schedule.

COPYRIGHT

You own the copyright to your work and grant USENIX first publication rights. USENIX owns the copyright on the collection that is each issue of *;login:*. You have control over who may reprint your text; financial negotiations are a private matter between you and any reprinter.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

OSDI | 20

14th USENIX Symposium on Operating Systems Design and Implementation

NOVEMBER 4–6, 2020 • VIRTUAL EVENT

OSDI brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software. The symposium emphasizes innovative research as well as quantified or insightful experiences in systems design and implementation.

PROGRAM CO-CHAIRS

Jon Howell
VMware Research

Shan Lu
University of Chicago

View the program and register today!

www.usenix.org/osdi20