FreeBSD® JOURNAL

March/April 2021

FreeBSD 13

- Looking to the Future
- Tool Chain
- Boot Loader
- TCP Cubic
- Zstd in ZFS

Also:

Vendor Summit Report

Practical Ports

LETTER from the Foundation

Welcome to the 13.0 Release issue!

Like you, we are excited to see the hard work from the last two years come together in a single release. Results from many Foundation-funded projects are evident in the latest release. They include help with the transition from SVN to Git, contracting with Moritz Systems to update LLDB in FreeBSD to fully replace GDB as a modern debugger, and funding improvements to Linuxulator including stabilizing the code base and making it easier to get started using it. The Foundation also funded work on ZFS compression and other OpenZFS improvements as well as wifi and graphics improvements.

Thank you to those whose donations permitted the Foundation to continue to support FreeBSD and especially to everyone whose tireless efforts helped make the 13.0 release such a resounding success.

We hope you'll enjoy this special issue of the FreeBSD Journal and that you will share it with your friends and colleagues!

On behalf of the FreeBSD Foundation,
Deb Goodkin
FreeBSD Foundation Executive Director

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Design & Production • Reuter & Associates

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
3980 Broadway St. STE #103-107, Boulder, CO 80304
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

FreeBSD Vendor Summit 2020

BY ANNE DICKISON

Developer and vendor summits are at the heart of the FreeBSD community. They provide a much-needed avenue for face-to-face discussions, group decision making and all-important bug fixing. The FreeBSD Summit organizers, including members of the FreeBSD Foundation, quickly realized that a 2020 in-person event was not going to happen. With that in mind, the organizers began working on the first-ever, online, 2020 FreeBSD Vendor Summit. Taking place November 11–13, 2020, the Summit consisted of 3 half-day sessions with both vendor talks and discussion topics. FreeBSD Journal Editorial Board President, John Baldwin, emceed the event. In addition to being recorded, the sessions were live streamed on YouTube. Wednesday, November 11, began with a welcome from John, followed by an update from FreeBSD Foundation Executive Director, Deb Goodkin. The rest of the program included vendor talks from Heiko Wilke of Beckhoff, Muhammad Ahmad of Seagate and Andrew Wafaa of ARM. Next up, Ed Maste led the first discussion session of the day on the ARM64 roadmap including moving the platform to Tier 1. Day one concluded with a discussion on bhyve led by Peter Grehan and John Baldwin.

Day 2 began with John Baldwin welcoming everyone back and introducing the program. First to speak was Axel Kloth of Axiado, followed by Jonathan Eastgate of SimPRO, and Allan Jude and Sabina Anaja of Klara Systems. The day concluded with the FreeBSD Foundation's Director of Technology and Core Team Member, Ed Maste joining fellow Core Team Member, Warner Losh to head up a lively discussion about the Project's transition to Git and what that would look like for Vendors. Folks were able to participate in the discussion sessions and Q&A sessions of the vendor talks via the event channel, Summit slack channel, or within the YouTube live stream.

Day 3 began much like Day 2 with a brief welcome and introduction of the program. First up was Alexander Sideropoulos of NetApp. His talk was followed by the always productive 13.0 Planning Session led by Ed Maste, John Baldwin and FreeBSD Foundation Board and Core Team Member Geroge Neville-Neil. After a short break, another fruitful discussion session was centered around planning for the 14.0 release. Luca Pizzamiglio wrapped up the technical part of the summit with a talk on "A container-based service mesh on FreeBSD." Following the last talk of the day, attendees were invited to join a separate social hour providing a greater opportunity to network with other attendees.

Individual videos, notes, and recordings of the entire live stream are available on the November 2020 FreeBSD Vendor Summit wiki page. The next online FreeBSD Developer Summit is scheduled to take place on June 9–11, 2021, registration is now open.

ANNE DICKISON is the Marketing Director, FreeBSD Foundation.

WeGet letters

by Michael W Lucas

> Oh! Incorruptible Single Source of Truth,
>
> FreeBSD 13 tears me between lust and loathing. I need the improvements, but I have this superstitious fear that number 13 is unlucky. I know it's irrational, but that superstition couldn't have endured for centuries without there being some truth to it, could it?
>
> Tell me everything will be okay.
>
> —Hesitant Upgrader Geek

HUG,

What? Sorry, I was thinking of something else.

Don't feel insulted. You're not special. It's what I do. Think of other things, that is. Not apologize. Saying "sorry" isn't an apology, it's a statement that I am vaguely aware that you might get all emotional at me and the annoying screech of your tantrum would interfere with my digestion. An apology includes a recapitulation of your actions, an acknowledgement those actions harmed others, a statement of regret, and a query as to how you can compensate others for the damage you've inflicted. All four long-time readers of this column immediately comprehend that's all far too much work for me.

If you want an apology, you'll need to make it yourself.

And isn't that why we work with machines? The machine has no feelings and doesn't care about yours. Those 32-bit timers roll over and crash the system without regard for your spouse giving birth or the new Star Trek's release date or you giving birth. The machine treats us with undifferentiated indifference. So many sysadmins want to treat other humans with that same indifference, but all too often devolve into thoughtless, reflexive contempt.

You aren't equipped for indifference. The chunk of electrified fat occupying your skull called "you" (whatever that means) evolved for caring. Indifference got eaten off the evolutionary tree.

The machine is glorious in its indifference.

Best of all, the machine is ultimately logical.

The CPU is nothing but a collection of logic gates. The video card is so stuffed with logic gates that we don't even call it a video card anymore, it's a Graphics Processing Unit and it's most valuable as a ScamCoin Environmental Destruction Node. All those chips and circuits and ports on the motherboard are nothing but carefully intertwined wires charged with carefully regulated electricity.

Okay, the electricity has some quantum in it. Electrons can't help the quantum—our universe defines them that way. Don't blame anyone for how they're made, blame them for their choices and actions. (Like writing letters to advice columnists. That was certainly a choice.)

Quantum aside, at the macro-but-still-microscopic level? The whole machine is ultimately knowable.

If only there wasn't so much of it.

Think about what happens when you try to watch a conference video. You move the mouse over the play button. That mechanical mouse motion is transformed into electrical signals, which get dumped into some sort of operating-system-level interpreter, deciphered, and transformed into pointer motion on the screen. This is all operating system level work, originally developed by people from a previous generation. We consider these functions well-tested, even if the overwhelming majority of computer "experts" have no idea how it really works. Our predecessors wrote this code, and it basically works, so other than a few hard-core operating system developers we trust that the pointer will move.

The click goes through the same process, for the same reason.

The video is where things get really hinky.

Writing individualized instructions for each of the billions of transistors inside the hardware is so much work that, like apologizing, we refuse to do it. Propose writing a video player in pure assembler and any programmer capable of the task would either deride your parentage or charge enough up front to live comfortably in a non-extradition country. Even if you blackmailed a competent programmer into accomplishing such a task, they wouldn't really be addressing individual transistors. Primordial assembly, the sort that Kernighan and Ritchie wrote C to escape from, doesn't represent modern hardware. Assembly is closer to the logic gates than any other language, and it runs on top of processor microcode.

So, you add an abstraction, like C.

C lets us craft miraculous programs, like device drivers and text editors and segmentation faults. Some programmers can deftly hand-twiddle a "stack" that's a representation of computer memory in the 1970s. Forget mastering C; achieving journeyman C programmer status requires a certain species of electrified skull-fat, ample time, and either dedication or stubbornness.

Of those qualities, I possess only stubbornness. I do have laziness, which leads directly to Perl. Perl is written in C.

Let's say your video player is written in Perl. (You laugh, but I learned decades ago to never underestimate Perl programmers. A Perl programmer can achieve anything in the name of avoiding work.) Your code is an abstraction, running on an abstraction, running on an abstraction, running on a representation of hardware that was obsolete before Richard Nixon resigned. Every one of these abstractions has bugs.

By any reasonable logic, computers should not work. At all.

And yet, we've managed to make them work.

Realistically, your video player isn't written in Perl. It's in a web browser. The web browser is written using some sort of programming language or application toolset like Javascript or Go or Fortran or Haskell. Whatever. I don't know the real details and neither, unless you are extremely unfortunate, do you.

That's only the main engine of your web browser. It probably has add-on components written in Forth or Pascal or, Beastie help you, C++.

So, we don't have abstractions on abstractions. We have multiple piles of interlinked abstractions, all simultaneously affecting and rewriting one another as they co-operatively re-architect the contents of the machine's processor and memory. Yes, we've added "protections" to a bunch of these, but they're afterthoughts. Afterthought Security is not a thing.

Oh, I remember what I was thinking about!

Humanity's greatest invention? No, not the wheel. Or fire. Or even gelato.

It's bureaucracy.

A society is a machine made out of meat. We all have places in it. We're all continuously re-architecting its contents. Each of us can see only a tiny part of the machine. No one person can see the entirety of the machine; we can only truly see our little bit of it. We have opinions on the part of the meat machine that's most frustrating at the moment, because we're sure we wholly understand the issue even though others have spent years or decades maintaining it.

We sysadmins, we think we understand the machine when in reality we understand only a tiny slice of one of the many abstractions. A person who writes scripting languages thinks they have a good handle on memory management when what they really understand is the abstraction that the layer beneath provides to them. Repeat this for every single abstraction.

A modern computer is a giant bureaucracy. You understand, at most, an office. You could devote your life to comprehending the logic of one of these systems—but understanding the whole is nearly impossible. Evolving languages, evolving standards, evolving hardware mean that even if you achieve Buddha-level enlightenment, the machine will leave you behind.

Declaring a language "safe"? Read that as "We've done our best to isolate our mistakes from the other departments." I appreciate the effort even as I know failure is both inevitable and inexorable.

We march on a bridge of shifting sand across a bottomless chasm with no far end.

"Tell you everything will be fine?" No. Nothing will be fine. FreeBSD 13 is no different from any other operating system in that respect. It's merely the most honestly numbered release in history.

And it will never, ever apologize for it.

Have a question for Michael?
Send it to letters@freebsdjournal.org

MICHAEL W LUCAS The latest books to emerge from society's Michael W Lucas Abstraction Layer are TLS Mastery, Only Footnotes, and the imminent $git sync murder. Thirty years in systems administration have purged him of bitterness, cynicism, and sarcasm. Learn more at https://mwl.io.

Looking to the Future

BY JOHN BALDWIN

FreeBSD's 13.0 release delivers new features to users and refines the workflow for new contributions. FreeBSD contributors have been busy fixing bugs and adding new features since 12.0's release in December of 2018. In addition, FreeBSD developers have refined their vision to focus on FreeBSD's future users. An abbreviated list of some of the changes in 13.0 is given below. A more detailed list can be found in the release notes.

Shifting Tools

Not all of the changes in the FreeBSD Project over the last two years have taken the form of patches. Some of the largest changes have been made in the tools used to contribute to FreeBSD. The first major change is that FreeBSD has switched from Subversion to Git for storing source code, documentation, and ports. Git is widely used in the software industry and is more familiar to new contributors than Subversion. Git's distributed nature also more easily facilitates contributions from individuals who are not committers. FreeBSD had been providing Git mirrors of the Subversion repositories for several years, and many developers had used Git to manage in-progress patches. The Git mirrors have now become the official repositories and changes are now pushed directly to Git instead of Subversion. FreeBSD 13.0 is the first release whose sources are only available via Git rather than Subversion. The first phase of this process focused on adapting the Project's existing workflows and tools (such as Phabricator and Bugzilla) to work with the new Git repositories. The next phase will allow us to explore additional tools such as pre-commit testing and continuous integration.

> Not all of the changes in the FreeBSD Project over the last two years have taken the form of patches.

A second major change is the adoption of AsciiDoc for the source format of FreeBSD's documentation and website. FreeBSD's documentation consists of three broad groups: manual pages, books and articles (such as the FreeBSD Handbook), and the project website. Books and articles were previously written in an SGML markup language called DocBook, and the website was written directly in HTML. While DocBook is very expressive and provides support for many features such as callouts, footnotes, and indices, it is a verbose format. Since the original design of DocBook, lighter-weight markup languages such as MarkDown have become prevalent. AsciiDoc is a lighter-weight markup language similar to MarkDown that retains the expressiveness of DocBook. The FreeBSD documentation team recently converted all of the books, articles, and website to AsciiDoc. This provides a simpler and easier to read format that will make it easier for new folks to contribute documentation.

Manual pages continue to be written in a dialect of troff known as mdoc.

Planning for Future Systems

One of the changes in FreeBSD's focus over the past few years has been to emphasize support for systems that users will be using in the future over support for older systems used by a decreasing number of users. This does not mean abandoning support for all systems which are not brand new. However, as some older systems recede further into history, the benefit of maintaining support for those systems in the tree no longer justifies the cost. FreeBSD 13.0 removes support for older 32-bit ARM systems as well as the UltraSparc platform. Device drivers for some older devices that are no longer commonly used have also been removed. In addition, in recognition of the dominance of 64-bit x86 systems, the 32-bit x86 architecture has been demoted to a Tier 2 architecture.

Streamlining our focus has allowed the Project to devote more resources to other architectures and drivers whose use will grow in the future. ARM, PowerPC, and RISC-V have all received substantial changes including support for new drivers and improved performance. The 64-bit x86 architecture now supports Hygon Dhyana processors as well as support for 57-bit user virtual addresses on newer Intel processors. Finally, all of the architectures in 13.0 are supported by the in-tree LLVM toolchain including the clang compiler and lld linker. By no longer maintaining compatibility with legacy GPLv2 toolchains, FreeBSD can now adopt modern language and toolchain features. (For more on this, see Ed Maste's article, FBSD 13 Tool Chain also in this issue.) Along with changes to replace or retire other GPL-licensed components in the base system, this also means that FreeBSD 13.0 ships with only two GPL utilities and one LGPL library in the base system.

A second major change is the adoption of AsciiDoc for the source format of FreeBSD's documentation and website.

OpenZFS

FreeBSD has included ZFS in the base system for over a decade. FreeBSD's ZFS support was originally ported from OpenSolaris and for a long time tracked the ZFS support in the public OpenSolaris (later illumos) repository. Over the past few years, active development of ZFS has moved out of the illumos repository into the cross-platform OpenZFS project. FreeBSD 13.0 replaces the illumos-derived ZFS support with code from OpenZFS. This brings in several new features including encrypted datasets and ZSTD compression. (See also Allan Jude's article, Zstandard Compression in OpenZFS also in this issue)

Networking

13.0 includes several networking changes. Kernel TLS offload enables a single web server to transmit hundreds of gigabits of HTTPS traffic (see John Baldwin's article, TLS Offload in the Kernel). The NFS client and server now support NFSv4.2. This includes a new system call to permit optimized server-side file copies. The NFS client and server also support NFS over TLS via kernel TLS offload.

Security

FreeBSD 13.0's kernel contains several improvements to the kernel cryptography framework used for geli(8), ZFS, IPsec, and kernel TLS. 64-bit ARM systems will now make use of accelerated software cryptography for the AES-GCM and AES-XTS ciphers out of the box via the armv8crypto(4) driver. Both 32-bit and 64-bit x86 systems also include support for accelerated software cryptography in the default kernel via the aesni(4) driver.

Boot Loader

The per-kernel boot loader includes several changes. First, when booting from UEFI, the default install now installs the full boot loader to the EFI system partition. Previously, a small boot loader in the EFI system partition was used to locate and boot the full boot loader. This two-stage process proved unwieldy and the firmware now loads the full boot loader directly. Secondly, on x86 systems the boot loader now uses a graphical display on the video console both when booting via UEFI and when booting via BIOS. This graphical console is then handed off to the kernel for use as a framebuffer by the vt(4) driver.

Virtualization

FreeBSD 13.0 includes several virtualization improvements both as a guest and a host. The VirtIO suite of device drivers in the kernel now support version 1 of the VirtIO specification. This improves compatibility with hypervisors, emulators, and simulation models which provide VirtIO devices. The bhyve(8) hypervisor includes several changes including improved VNC support (including compatibility with the built-in "Screen Sharing" VNC client in macOS), VirtIO 9p filesystem sharing, and initial support for virtual machine snapshots.

We hope you enjoy
FreeBSD 13.0.
Join us for the next adventure
developing FreeBSD 14!

Conclusion

FreeBSD 13.0 is the product of contributions from the Project's community over the past two years. Thank you to everyone who has contributed to this release by testing snapshots, reporting bugs, submitting patches, working with users on social media, and countless other tasks. We hope you enjoy FreeBSD 13.0. Join us for the next adventure developing FreeBSD 14!

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for 20 years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger and LLVM. John lives in Concord, California, with his wife, Kimberly, and three children: Janelle, Evan, and Bella.

FreeBSD 13.0 Tool Chain

BY ED MASTE

FreeBSD 13.0 marks a significant milestone in the evolution of the FreeBSD tool chain: the completion of a decade-long migration to using a modern, permissively licensed compiler, linker, debugger, and miscellaneous binary utilities for all of FreeBSD's supported architectures.

From the beginning, FreeBSD relied on the GNU tool chain, including the GCC compiler, Binutils linker and binary utilities, and the GDB debugger. These tools were kept current with regular updates by a group of developers, both volunteer and paid. This continued until 2007 when the GNU project changed the license on these tools to version 3 of the GNU Public License (GPLv3), and they were not updated again.

A few years later, about twenty, developers met for a developer summit session at BSDCan 2010 to plan the future of the tool chain, with a goal of moving to a modern, permissively licensed tool chain. License considerations were only one part of the rationale. New technology, the ability to implement new and bespoke functionality, and promoting competition in OSS tools also factored into the team's goals.

At the same time, a number of tool-chain projects were firming up, both external to and within the FreeBSD project. Summit attendees noted that LLVM and Clang were maturing rapidly and gaining interest in the compiler research community. A BSD-licensed ELF tool-chain project was in progress, as well as a number of debugger projects.

The team set a goal of collaborating with these tool-chain projects and migrating to the new components. Individual components were added to FreeBSD as they became usable, often installed alongside the existing tool, but disabled by default. After testing and validation, they became default, sometimes on an architecture-by-architecture basis. Eventually, the new tool was enabled across all supported architectures and the old tool was removed. Aside from a small caveat with the debugger, this is now complete for all tool-chain components and all supported architectures. Read on for the full details.

From the beginning, FreeBSD relied on the GNU tool chain, including the GCC compiler, Binutils linker and binary utilities, and the GDB debugger.

Compiler

The compiler is one of the most important components in the tool chain and was the first major component that we migrated. The Clang compiler had its initial release in 2007 and was maturing quickly. Roman Divacky imported a version into FreeBSD's contrib software tree in

June 2010, and we provided it (as /usr/bin/clang) in FreeBSD 9.0 for only the x86 and powerpc architectures.

With substantial effort on the part of the Clang developers and integration effort from Roman, Ed Schouten, Dimitry Andric, and others, we enabled Clang as the default compiler (/usr/bin/cc) for i386 and amd64 in 2012. This shipped with FreeBSD 10.0.

In 2014, Clang was ready to be the default compiler for little-endian Arm. It was also provided on PowerPC, but not as the default compiler. This is the configuration shipped in FreeBSD 11.0. FreeBSD 11.0 also introduced 64-bit arm (AArch64) support, with Clang as the only available compiler.

Compiler updates continued in the lead up to FreeBSD 12.0, but architecture support remained the same. 2019 and 2020 saw significant upstream Clang/LLVM architecture improvements, and remaining architectures including RISC-V and MIPS switched to Clang by default. Sparc64 was the sole remaining architecture relying on the obsolete in-tree GCC version. When FreeBSD/sparc64 was retired, there was no need to keep GCC in the tree, and it was removed in early 2020.

Tool chains typically include an assembler. As of FreeBSD 13.0 we use Clang's integrated assembler (IAS) to assemble parts of the FreeBSD base system and no longer provide a standalone /usr/bin/as.

Using Clang as FreeBSD's standard compiler has made it a compelling system for full-system research. The University of Cambridge's Temporally Enhanced Security Logic Assertions (TESLA) and Capability Hardware Enhanced RISC Instructions (CHERI) research projects build on Clang/LLVM and benefit from the availability of a full, Clang-built operating system kernel and userland.

Compiler updates continued in the lead up to FreeBSD 12.0, but architecture support remained the same.

Linker

A linker combines object files and libraries into an executable or shared library. When the tool chain project started, no alternative linker with a clear path to viability existed. The MCLinker project demonstrated some early momentum, but ultimately did not support many features required by the FreeBSD build.

By 2015, LLVM's lld was making good progress, and we imported a snapshot in 2016. We started with it built by default but installed with a non-default filename. It was available via a compiler switch, -fuse-ld=lld.

By FreeBSD 12.1 lld was built for all architectures except sparc64 and riscv64 and was the default linker for 32- and 64-bit arm and x86. With the development of riscv64 support upstream in lld and sparc64's retirement, lld became the sole linker for all architectures in FreeBSD 13.0.

Binary Utilities

A tool chain includes a number of small tools for examining or processing object, library and executable files—ELF objects in the case of FreeBSD. This includes tools like size, strings, and readelf for examining files, objcopy for converting or processing objects, and ar for creating and extracting library archives. This category also includes libraries used by other tools to parse objects—for example, libelf and libdwarf.

In older FreeBSD versions, most of these tools were obtained from GNU binutils with a few exceptions--for example, the archive manager was a bespoke tool based on libarchive. FreeBSD also implemented versions of libelf and libdwarf starting in 2006.

The ELF Tool Chain project began as a standalone effort in 2008, led by Joseph Koshy. The project imported some existing FreeBSD tools and libraries, and additional contributors joined the project. Kai Wang implemented many of the missing tools, including elfcopy/objcopy and readelf.

By 2015, the ELF Tool Chain versions of addr2line, elfcopy/objcopy, nm, size and strings were sufficiently functional that we switched to them by default and shipped them in FreeBSD 11.0.

This work did not address the assembler or linker; work began on both tools within ELF Tool Chain, but neither is functional at present. We addressed the assembler by relying on Clang's integrated assembler instead and switched to LLVM's lld for the linker for almost all architectures. Sparc64 was the sole architecture relying on the in-tree GNU ld. With the retirement of sparc64 support we were able to remove binutils from the FreeBSD tree prior to FreeBSD 13.0.

The migration from binutils also left FreeBSD without objdump in the base system. Much of the information provided by objdump is also available (in a slightly different format) from readelf, and LLVM provides an llvm-objdump that is largely compatible. It is not yet installed by default in the base system but will likely be enabled in a future version. Of course, GNU objdump is also available by installing the binutils port or package.

Source Level Debugger

The remaining tool chain component is the debugger, and LLVM provided a path forward here as well. LLDB is LLVM's debugger, described in detail in previous FreeBSD Journal articles. It builds on LLVM components for disassembly, uses Clang as the expression parser, and is scriptable via Python and Lua.

We added LLDB to the build as an experimental feature in 2013, prior to FreeBSD 10.0, and enabled it by default for amd64 and arm64 in 2015, for FreeBSD 11.0. In 2017, Karnajit Wangkhem submitted a patch adding support for i386 to the JIT expression engine, and we enabled it by default in FreeBSD 12.0.

LLDB includes FreeBSD target support for 32-bit Arm, as well as PowerPC and MIPS, although it is not well tested. Builds may be enabled by default after sufficient testing and integration. LLDB is now a functional FreeBSD userland debugger, but currently lacks kernel debugging support.

LLDB includes FreeBSD target support for 32-bit Arm, as well as PowerPC and MIPS.

With FreeBSD Foundation sponsorship, Moritz Systems recently fixed many outstanding bugs, improved arm64 target support, added an initial follow-fork mode, and is improving userland core file debugging. Projects to implement live kernel debugging and post-mortem kernel coredump support are under discussion. We also need to implement RISC-V target support.

CTF Tools

FreeBSD's DTrace support makes use of the Compact C Type Format (CTF), which provides the minimum required debug information to make C language types available to DTrace scripts. This currently uses three tools released under the Common Development and Distribution License (CDDL). They implement CTF version 2 and are largely unchanged since the import of DTrace from OpenSolaris in 2008.

There are a number of alternative CTF tool implementations available. Contemporary binutils includes libctf, a library to parse and edit CTF data. The CTF format has also been extended to version 3 and more recently to version 4. Also, OpenBSD's Martin Pieuchot has implemented a minimal set of permissively licensed CTF tools. Future FreeBSD tool chain work will be needed to determine where we take the CTF tooling.

Diagnostic Tools

Valgrind is part of a set of tools for memory access and leak debugging and checking other aspects of program behavior. FreeBSD support for Valgrind has been maintained outside of the main Valgrind tree for almost two decades, and a patched Valgrind is available from the ports tree and packages collection. Many different developers have maintained and updated the FreeBSD Valgrind port over the years; most recently Paul Floyd has updated it to the latest released version 3.17.0 and is working to commit the FreeBSD changes upstream.

FreeBSD support for Valgrind has been maintained outside of the main Valgrind tree for almost two decades

Clang includes built-in support for various sanitizers that provide debugging and diagnostic information. AddressSanitizer detects memory errors such as out-of-bounds accesses or use-after-free and is available via the -fsanitize=address command line flag (using Clang in the base system). A related tool, MemorySanitizer, detects reads of uninitialized variables. ThreadSanitizer detects data races, and UndefinedBehaviourSanitizer catches undefined C behavior (such as signed integer overflow) at runtime. Additional sanitizers exist and will need more work to enable them on FreeBSD.

Clang also includes a static analyzer invoked via the scan-build front end. It is available with Clang installed from an LLVM package, not the base-system Clang. The static analyzer serves a purpose broadly similar to compiler warnings, but at a much higher level.

For code coverage analysis, LLVM provides llvm-cov. It is also installed as gcov and operates in a GNU, gcov-compatible mode when invoked via the alias. When a program is compiled with –coverage, it emits a .gcov file when it exits, and gcov uses that to display execution counts for each line (or basic block) of the source. Performance profiling is available via bespoke BSD licensed code: hwpmc kernel support and the pmcstat userland tool.

Future Work

With the transition to Clang/LLVM complete, the tool chain team is investigating enhancements we can build on this foundation. One of these is Link Time Optimization (LTO), intermodular optimization performed during the link stage. Essentially LTO uses object files and libraries containing LLVM intermediate representation (IR) instead of target binary ELF objects. This allows these to be combined at link time and have the LLVM optimization passes operate on the entire binary as a whole and not just individual compilation units. One caveat with LTO is that the nm and ar tools need to be able to parse LLVM IR symbol tables, and the ELF Tool Chain versions used by the base system do not have this capability. We will need to either extend ELF Tool Chain nm and ar or switch to the LLVM versions of these tools.

Another tool chain feature provided by Clang is Control Flow Integrity (CFI). CFI broadly refers to techniques used by the compiler to avoid run-time attempts to subvert a program's intended operation (control flow) by some malware that manages to gain execution. Clang's CFI support requires LTO and includes a number of individual checks to detect various cases of bad casts and invalid indirect calls.

The CHERI project presents another opportunity for future work in the FreeBSD tool chain. CheriBSD is a derivative of FreeBSD that implements CHERI memory protection and software compartmentalization and is maintained in an external repository. CHERI also includes an LLVM-based tool chain with support for a number of CHERI-enhanced ISAs.

ED MASTE manages project development for the FreeBSD Foundation. He is also a member of the elected FreeBSD Core Team. Aside from FreeBSD, he has contributed to a number of other open-source projects, including LLVM, ELF Tool Chain, QEMU, and Open vSwitch. He lives in Kitchener-Waterloo, Canada, with his wife, Anna, and children, Pieter and Daniel.

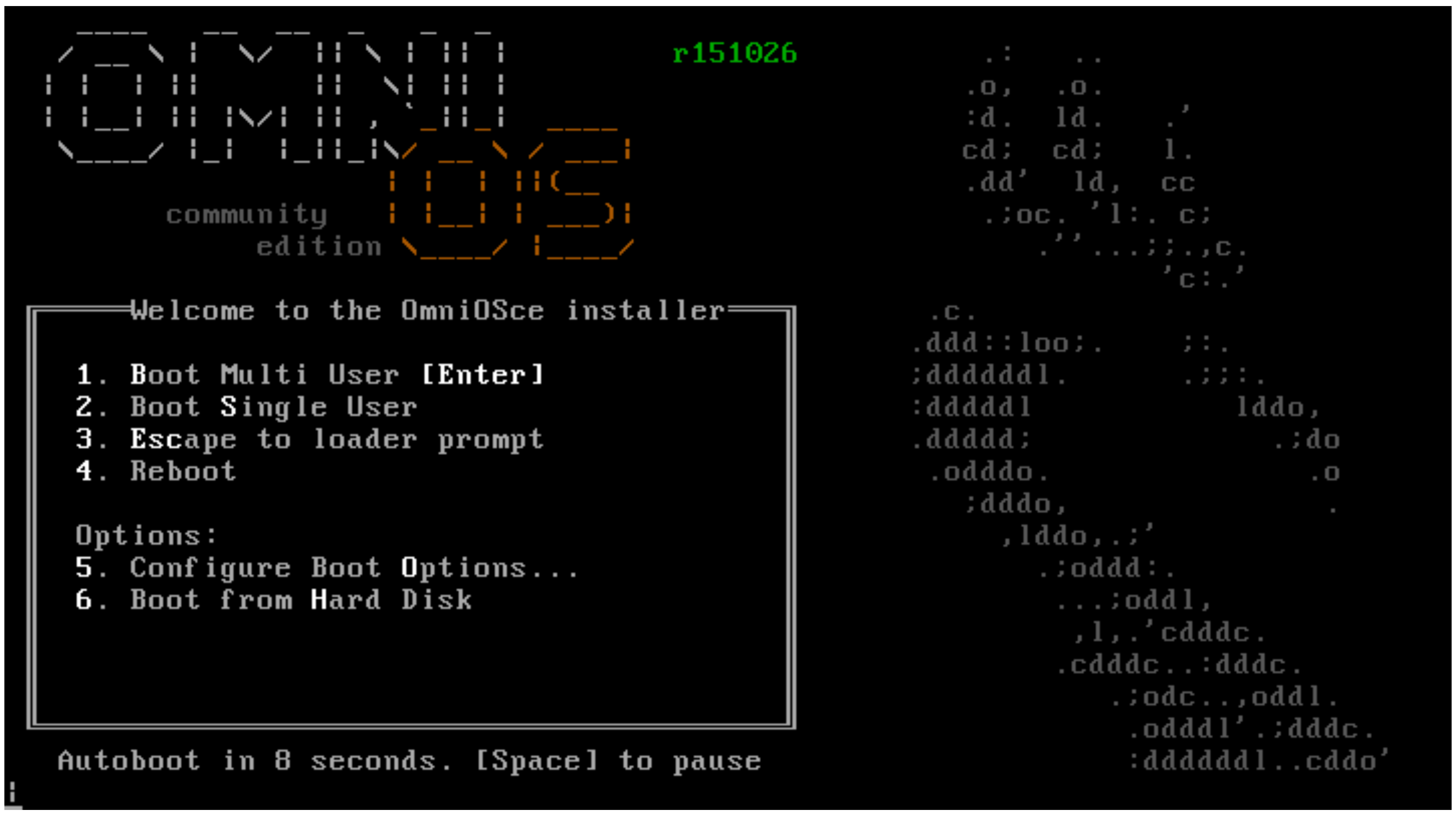
Is There a New Loader in FreeBSD 13.0?

BY TOOMAS SOOME

The short answer is “no, it is still the same good old loader.” But we are trying to make it more friendly and support more features.

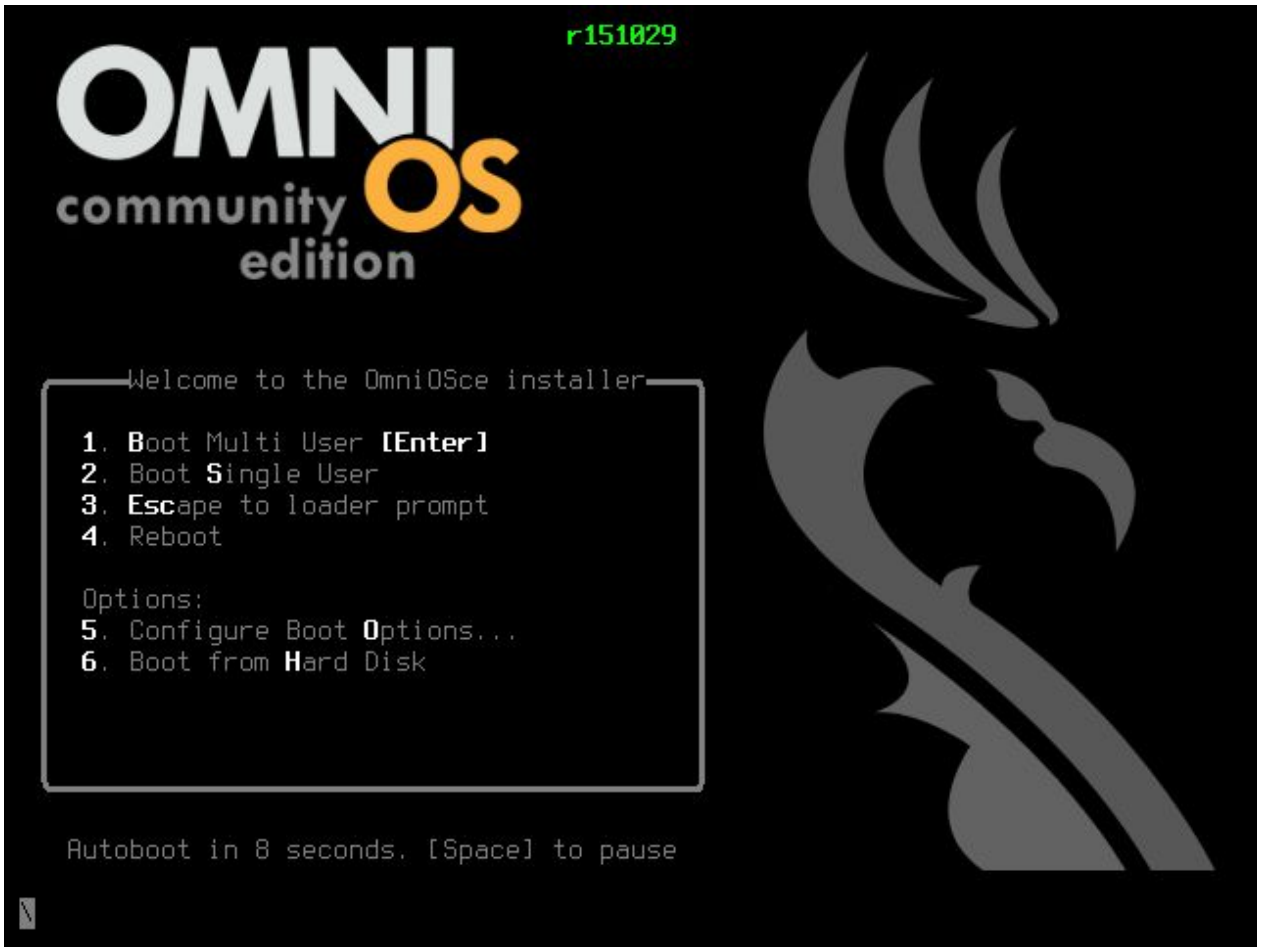
I started working with boot loaders that were not on FreeBSD but on illumos. At that time, illumos was using old grub 0.96 and supported only the BIOS boot. UEFI systems were already around at that time and illumos really needed to support UEFI systems. My initial work was to investigate newer grub. It was feature-rich, widely used but hard to manage, and its licensing is not friendly when you need to support features specific to file systems like zfs or to add operating-system specific features. This led me to the FreeBSD boot loader and contributing to its development.

When I started porting the FreeBSD boot loader to illumos, illumos supported a serial console and VGA text mode console only. To support UEFI, I had to implement support for the UEFI framebuffer-based console for the illumos kernel. Once implemented, a logical step was to add the same feature for the loader, and when there is an option to draw the console on the UEFI framebuffer, then re-using the same code to draw on the Vesa BIOS Extensions (VBE) linear framebuffer is just another logical step in development. Once done, we got public postings like this https://omnios.org/setup/fb:



```
                                   r151026

          community
             edition

 Welcome to the OmniOSce installer

  1. Boot Multi User [Enter]
  2. Boot Single User
  3. Escape to loader prompt
  4. Reboot

  Options:
  5. Configure Boot Options...
  6. Boot from Hard Disk

 Autoboot in 8 seconds. [Space] to pause
```

Figure 1 Loader with ascii art.



```
Welcome to the OmniOSce installer

 1. Boot Multi User [Enter]
 2. Boot Single User
 3. Escape to loader prompt
 4. Reboot

 Options:
 5. Configure Boot Options...
 6. Boot from Hard Disk


  Autoboot in 8 seconds. [Space] to pause
```

Figure 2 Loader with images.

Back to the FreeBSD Boot Loader

But enough about illumos, let's get back to FreeBSD and see what we have managed to do there. Please note, most of the work I have done has involved flowing from Freebsd to illumos or vice versa.

OpenZFS

As FreeBSD 13.0 is now using OpenZFS, we support most OpenZFS features for booting. The encrypted datasets and draid are still in the todo list, however.

Console

Literally, the most visible change is the graphical console for the loader. While the current implementation is not perfect and can be improved, I hope most users will enjoy the updated look.

The loader console terminal emulator is teken from the kernel tree. From there, we have the first two tunables we can set:

```
teken.fg_color
teken.bg_color
```

The acceptable values are ansi color names or numeric values 0 – 7.

The UEFI loader uses the framebuffer console by default unless the serial console is configured.

The BIOS loader defaults to use text mode at this time. For the BIOS loader, the console can be controlled by setting:

```
screen.textmode="0"
```

This will cause the loader to set up VBE framebuffer mode and to use a display-preferred resolution when EDID information is available. The default fall back resolution is 800x600.

Both UEFI and BIOS loaders allow setting the screen resolution via the following tunables:

```
efi_max_resolution
vbe_max_resolution
```

Having vbe_max_resolution set will also cause the loader to switch to use VBE framebuffer mode.

The framebuffer mode can also be set, queried and support modes listed by platform specific commands.

Command `gop` allows a user to get, set and list resolutions in the UEFI loader. Command `gop off` will switch the loader from drawing the console to the UEFI built-in terminal output method—Simple Text Output Protocol.

Command `vbe` allows the user to get, set and list resolutions in the BIOS loader. Command `vbe on` will switch the loader to use VBE framebuffer and `vbe off` will switch the loader to use VGA text mode.

If a user has switched the BIOS loader to use the VBE framebuffer but is booting an older kernel that does not provide a VT vbefb driver, then the loader will switch the console to VGA text mode just before starting the loaded kernel.

Fonts

At this time, we are providing terminus family console fonts, installed into the /boot/fonts directory. The loader has a built-in 8x16 font, but to save space, the built-in font only provides an ASCII set.

After the loader starts and initializes and it has obtained access to disks and determined the boot device and a boot file system, the loader will search for /boot/fonts directory and INDEX.fonts file. If present, the loader will get the list of available fonts and will build an internal, indexed list of available fonts. The INDEX.fonts file is used because we need to support tftp file transport, but tftp protocol does not implement reading directory listings.

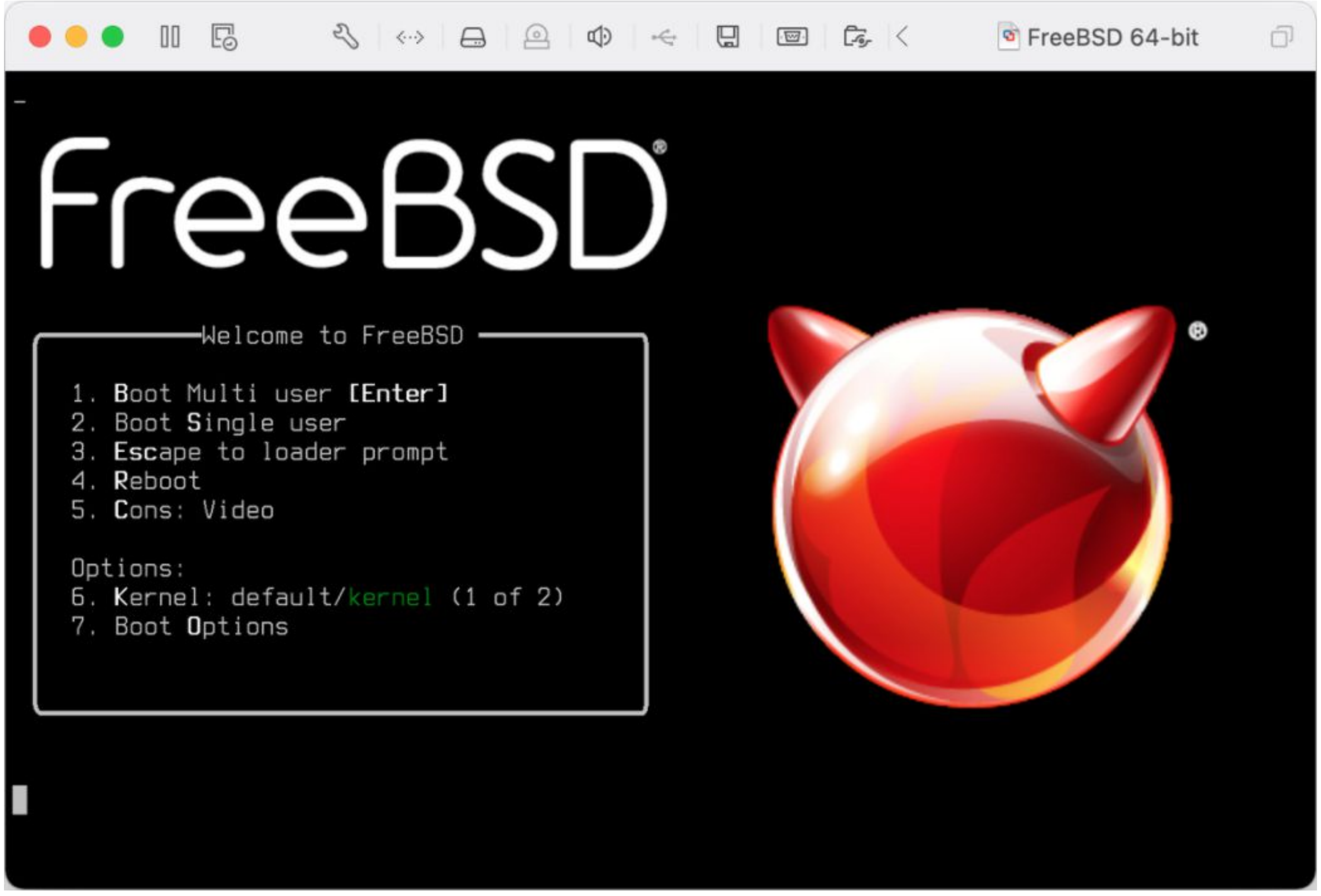
Once we have a list of available fonts, we load a preferred font based on the resolution used on the console display. If the default selection is not working well for a user, there are two methods for changing defaults:

First, the environment variable screen.font appears and permits changing a used font. An attempt to use a bad value or to unset the value will cause the list of currently available fonts to be printed on the console.

Second, the command `loadfont` allows a user to load a custom font file such as `/boot/fonts/gallant.fnt`. The font needs to be prepared with the `vtfontcvt(8)` tool.

The font indexing in the loader assumes unique font sizes. When there is an already-registered font file for font size 8x16, an attempt to load a different font file providing the same font size will cause the previously loaded file to be replaced by a new file.

The font currently used by the loader will be passed to the loaded kernel. By doing so, we preserve the look and feel and achieve a consistent transition from the boot loader to the running operating system.



```
Welcome to FreeBSD

 1. Boot Multi user [Enter]
 2. Boot Single user
 3. Escape to loader prompt
 4. Reboot
 5. Cons: Video

 Options:
 6. Kernel: default/kernel (1 of 2)
 7. Boot Options
```

Figure 3 FreeBSD 13.0 Boot loader

Images

To build better looking screens, the loader supports display of PNG-formatted image files. At this time, we require TrueColor with an alpha channel, and we support image scaling. Examples of how to use images in a logo or brand components of the bootloader menu can be found from drawer.lua and gfx-orb.lua files.

Drawbacks

Some systems experience a slow console when the framebuffer console is used. With the VBE framebuffer, one possible workaround is to use smaller color depth—default is 32-bit colors. With both VBE and UEFI, it may be possible to use smaller resolution and to configure better resolution once the KMS driver is running. And, of course, in some cases the only reasonable option may be to use a text console.

Summary

So again, no, there is not a new boot loader in FreeBSD, but we are trying to make sure it does what it should do—which is support loading and booting the FreeBSD operating system, provide features people might need for the task, and look reasonably good.

TOOMAS SOOME Born in Estonia. Toomas has been a UNIX admin since 1993, working mainly with Solaris. He is also an infrastructure architect, illumos developer and a FreeBSD src committer. Toomas is not afraid of boot loaders and can read and write Forth.

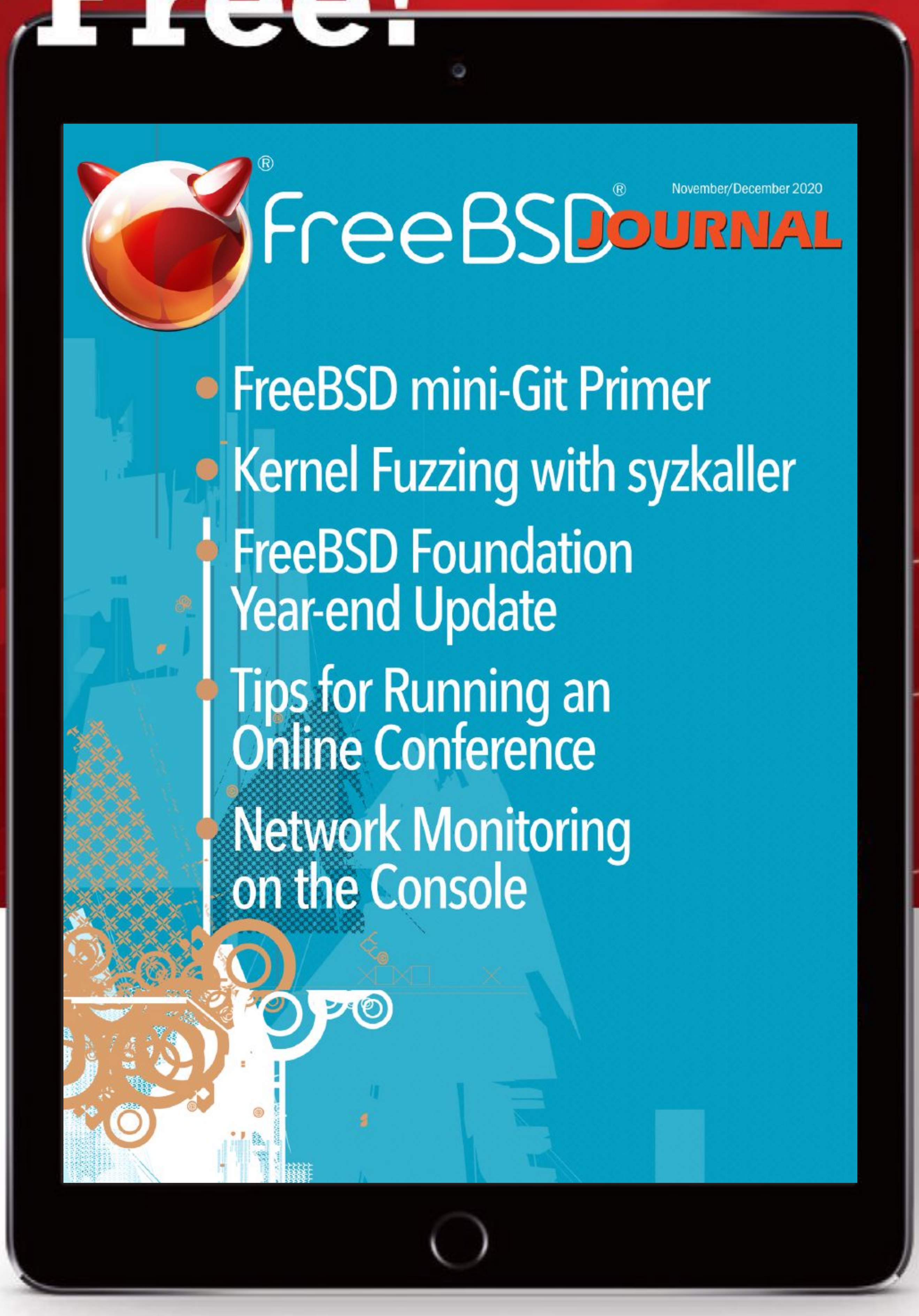
FreeBSD® JOURNAL

The FreeBSD Journal is Now Free!

Yep, that's right Free.

The voice of the FreeBSD Community and the BEST way to keep up with the latest releases and new developments in FreeBSD is now openly available to everyone.

DON'T MISS A SINGLE ISSUE!



FreeBSD® JOURNAL

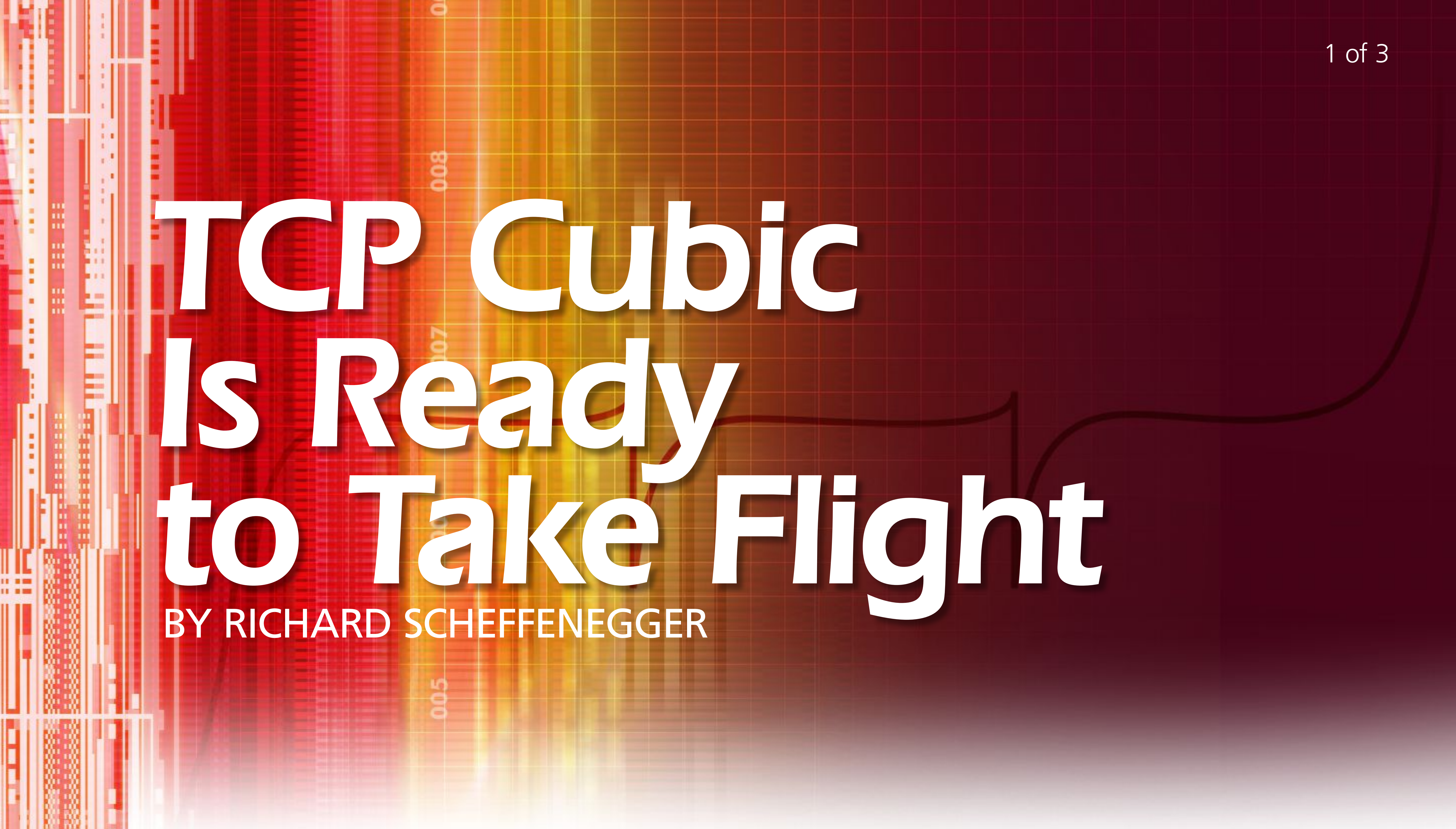
November/December 2020

- FreeBSD mini-Git Primer
- Kernel Fuzzing with syzkaller
- FreeBSD Foundation Year-end Update
- Tips for Running an Online Conference
- Network Monitoring on the Console

2021 Editorial Calendar

- Case Studies (January-February)
- FreeBSD 13 (March-April)
- Security (May-June)
- Desktop/Wireless/Graphics (July-August)
- Cloud (September-October)
- Embedded (November December)

Find out more at: freebsdfoundation/journal



TCP Cubic Is Ready to Take Flight

BY RICHARD SCHEFFENEGGER

With FreeBSD 13.0, numerous improvements were made to the cc_cubic loadable congestion control module. TCP Cubic was originally implemented by Lawrence Stewart during his time at Swinburne University of Technology, Center for Advanced Internet Architectures based on an early draft of what eventually became RFC8312. TCP Cubic has become the de-facto standard congestion control mechanism in use today.

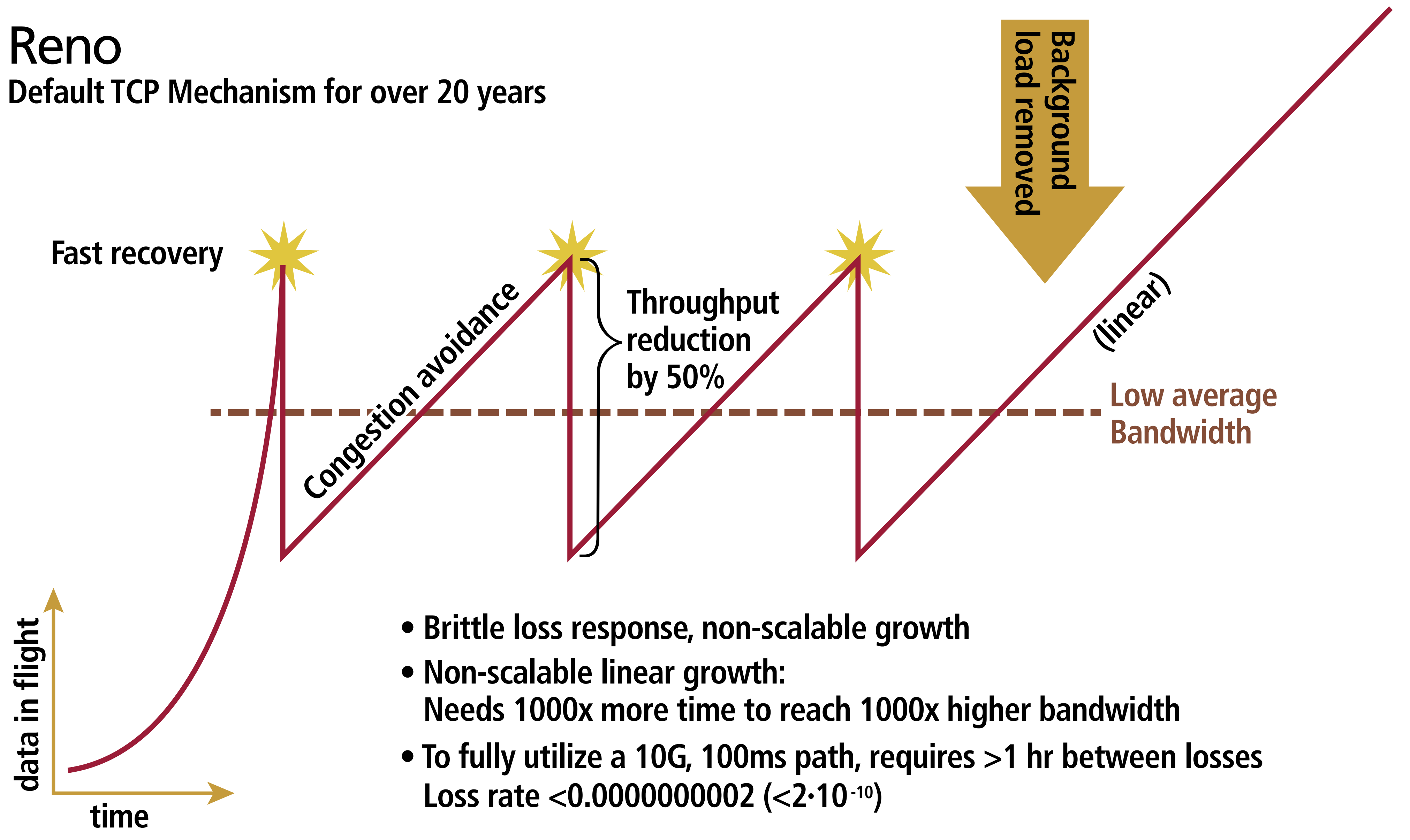
TCP Cubic

The default TCP congestion control in use by FreeBSD for the longest time is name NewReno—a variant of the Reno congestion control mechanism with improved loss recovery. The job of a congestion control algorithm is to detect and prevent an overload situation of the network where more data is injected than can be transported or delivered. NewReno used to be the gold standard in this space but does suffer a few restrictions.

While Van Jacobson has shown that any AIMD (additive increase, multiplicative decrease) scheme exhibits a stable operation for controlling the traffic, with modern high-speed links, the time it takes NewReno to ramp up the effective transmission speed is lackluster. If an overload situation is detected--typically using an explicit signal like a packet loss or specific bits in the TCP/IP headers—NewReno will reduce the effective transmission speed—and I use this term loosely, to not get bogged down on details like available data to transmit, congestion window and burst behavior, and timing when the application is ready to send more data—to 50% of the speed at the time the overload occurred. With a sufficient amount of data to transmit, provided by the local application at a sufficiently high speed, NewReno will then ramp up the transmission speed by roughly 1 full-sized packet every round-trip time (RTT).

> The job of a congestion control algorithm is to detect and prevent an overload situation of the network.

But running these numbers using modern networking technology, e.g. 10G links across the country with a latency of 100ms, it may take a singular NewReno session up to (5 Gbps / (1500 * 8)) * 0.1 sec RTT ~= 10 hours to ramp back up to utilize all the available bandwidth—provided no other packet drops (as indication of congestion) happen.



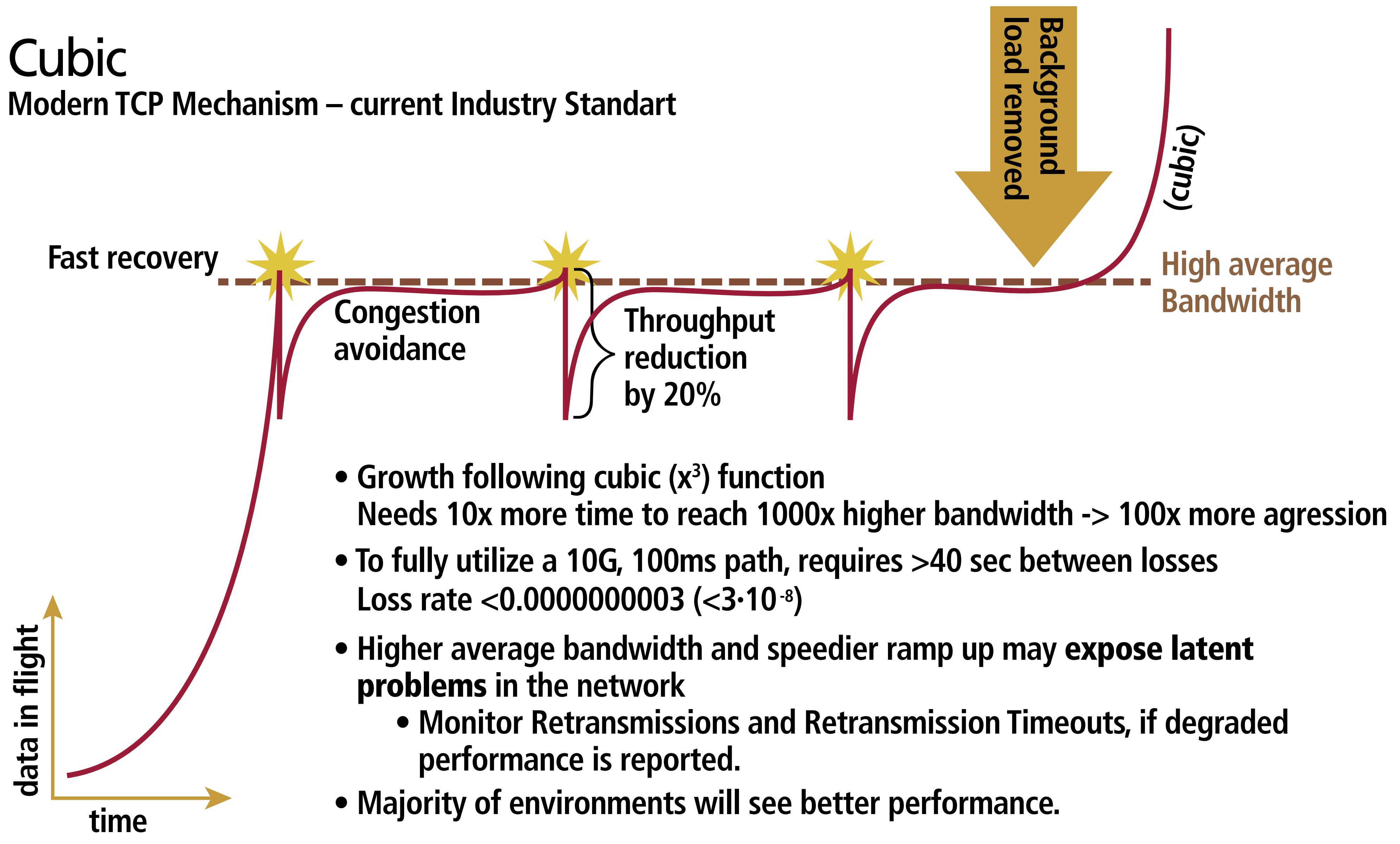
Reno

Default TCP Mechanism for over 20 years

- Brittle loss response, non-scalable growth
- Non-scalable linear growth:
  Needs 1000x more time to reach 1000x higher bandwidth
- To fully utilize a 10G, 100ms path, requires >1 hr between losses
  Loss rate <0.0000000002 (<$2\cdot10^{-10}$)

While TCP—when sending unlimited amounts of data—is designed to probe and eventually exceed the maximum bandwidth of the network, slow ramp-up is detrimental to this goal.

TCP Cubic addresses these limitations with two major changes. The first one is to reduce the speed only to 70% (80% in early drafts) of the transmission speed at time of overload. The second is to ramp up afterwards using a cube function which is scaled in such a way as to linger around the previous limit for a good time, but ramping up to that limit quickly--and if the available bandwidth of the network is no longer as restricted, to ramp up faster and faster, effectively matching the exponential bandwidth growth during TCP slow start.



Cubic

Modern TCP Mechanism – current Industry Standart

- Growth following cubic (x^3) function
  Needs 10x more time to reach 1000x higher bandwidth -> 100x more agression
- To fully utilize a 10G, 100ms path, requires >40 sec between losses
  Loss rate <0.0000000003 (<$3\cdot10^{-8}$)
- Higher average bandwidth and speedier ramp up may **expose latent problems** in the network
  - Monitor Retransmissions and Retransmission Timeouts, if degraded performance is reported.
- Majority of environments will see better performance.

While all these foundations were implemented—including a fast integer approximation for calculating the cube-root--some of the parameters did change between the cubic draft of 2007 and ultimate RFC8312. Thus, some work was necessary to being the existing code in-line with the RFC.

In the meantime, most other major OSs adopted Cubic as their default congestion mechanism, as in a direct competition between NewReno and Cubic, a flow using NewReno will get less share of the bandwidth available. Fortunately, Cubic was designed in a way to not fully starve out other congestion control mechanisms.

The existing code also assumed some implicit limits in the cubic code, which do not always hold with general purpose traffic patterns. A number of edge cases were not fully addressed. For example, nowadays, application-limited sessions are the norm. This is when TCP basically runs out of data to send, and all the state engines driven by processing more data have a discontinuation in time. As Cubic uses wall clock time rather than the passing of data over the session…<== rather than D23655 (cubic and slot start interaction) <== slow start…this has created some undesirable effects. (Author—is this change correct or did we misunderstand?)

While starting to run Cubic as a general-purpose congestion control on FreeBSD, the following issues were addressed without any claim of this being a complete list. Some general issues with the TCP base stack also showed up and were fixed while working on Cubic.

D26181 (editorial nit)
D26060 (adjust cwnd continuously, not only once per window – leading to massive traffic bursts)
D25976 (treat ECN like packet loss for Cubic)
D25746 (properly time the start of a cubic epoch with slowstart)
D25133 (cubic and RTO interaction)
D25065 (cubic and application limited)
D24657 (editorial)
D23655 (cubic and slot start interaction)
D23353 (cubic and ECN)
D19118 (deal with overflows during cubic math)
D18982 (prepare for good cubic math)
D18954 (cubic and After-Idle)

Overall, the foundation of cubic that has been available since FreeBSD 8.0 has been a solid foundation of the basic functionality and algorithms. A lack of production deployment left a number of corner cases and boundary conditions—e.g. for very long running TCP sessions--unchecked.

With the above improvements done, exercising the TCP Cubic variant in FreeBSD 13.0 should allow for slightly better throughput, especially across the public internet with high latency sessions. Nevertheless, additional exposure to peculiar traffic patterns may still show some shortcomings, even though the code is now in a more robust state to deal with most scenarios.

Not only was Lawrence Stewart very helpful in this improvement effort, but much of the heavy lifting was performed by Cheng Cui, especially doing regression and unit testing as well as finding all these edge cases and providing code improvements. There have also been many productive discussions on the bi-weekly FreeBSD Transport group calls.

RICHARD SCHEFFENEGGER is Consulting Solution Architect at NetApp.

Zstandard Compression in OpenZFS

BY ALLAN JUDE

ZFS is a highly advanced filesystem with integrated volume manager that was added to FreeBSD in 2007 and has since become a major part of the operating system. ZFS includes a transparent and adjustable compression feature that can seamlessly compress data before storing it and decompress it before returning it for the application's use. Because the compression is managed by ZFS, applications need not be aware of it. Filesystem compression not only saves space, but in many circumstances, can even lower read and write latency by reducing the total volume of data that needs to be stored or retrieved.

Originally ZFS supported a small number of compression algorithms: LZJB (an improved Lempel–Ziv variant created by Jeff Bonwick, one of the co-creators of ZFS, it is moderately fast but only offers low compression ratios), ZLE (Zero Length Encoding, which only compresses runs of zeros), and the nine levels of gzip (the familiar slow, but moderately high-compression algorithm). Users could thus choose between no compression, fast but modest compression, or slow but higher compression. Unsurprisingly, these same users often went to great lengths to separate out data that should be compressed from data that was already compressed in order to avoid ZFS trying to recompress it and wasting time to no benefit. For various historical reasons, compression still defaults to "off" in newly created ZFS storage pools.

In 2015, LZ4 replaced LZJB as the default when users enable compression without specifying an algorithm.

In 2013, ZFS added a new compression algorithm, LZ4, which offered both higher speed and better compression ratios than LZJB. In 2015, it replaced LZJB as the default when users enable compression without specifying an algorithm. With this new high-speed compressor, combined with an existing feature called "early abort," it became feasible to simply turn on compression globally, since incompressible data would be detected and skipped quickly enough to avoid impacting performance. The early abort feature works by limiting the size of the output buffer given to the compression algorithm to one-eighth smaller than the input buffer size. If the compression algorithm cannot fit the output into that smaller buffer, it fails and returns an error. As a result, the algorithm can preemptively disengage if it is not going to provide sufficient gains, in which case the data is stored uncompressed to avoid the overhead of decompressing a block that was barely compressed. In fact, enabling LZ4 compression on everything is so low impact that this set-and-forget configuration is very common and has even been the default in FreeNAS for many years.

The Project Begins

The project started in the fall of 2016 after the author had to miss the OpenZFS Developer Summit due to a scheduling conflict with EuroBSDCon. The goal was to integrate a recently announced new compression algorithm into OpenZFS. Zstandard (Zstd for short) was created by Yann Collet, the original author of LZ4. The purpose of the new algorithm was to provide compression ratios similar to gzip (with even greater flexibility, offering more than twenty levels to gzip's nine!) but with speeds comparable to those seen with LZ4.

As the project began, we immediately ran into issues with stack size since Zstd was written as a userspace program and had a penchant for large stack variables. This was a problem for integrating Zstd into the kernel, where the stack was limited to 16 KB, and had to support all of the other layers of the operating system before and after the compression integrated into the filesystem. We temporarily sidestepped this problem by just increasing the stack size in our development kernel and got the first version of ZFS with Zstd compression working after a few weeks of work. Then we set about modifying Zstd to instead use heap memory returned by the kernel malloc framework to reduce stack usage. This was difficult as there were often multiple exit paths from functions where the allocated memory needed to be freed. After only limited success, the project was set aside for a while, knowing when we came back to it, it was likely to be even worse, as all of the local patches would need to be rebased forward to a newer version of Zstd.

Luckily, when it came time to return to the project, Zstd version 1.3 had been released, with greatly reduced stack usage, while also allowing the caller to manage their own memory allocation. With these welcome improvements, Zstd would no longer require extensive modifications for kernel integration. In the end, only superficial changes were required, and Zstd could be used largely unmodified.

By the fall of 2017 and the next OpenZFS Developers Summit, we had a working prototype to demo at the conference. The summit provided an invaluable opportunity to talk to experts and much more experienced developers about remaining challenges. One of these was how to have the user provide the compression type (Zstd) and the level (1-19) in a way that would not result in fatal confusion should a user later change the compression type to gzip, where a level like "19" might be invalid. This issue was mentioned during the talk, and afterwards Robert Mustacchi came up and suggested a remarkably elegant solution: only expose the compression type to the user offering the different levels of Zstd but store them internally in ZFS as separate values. While that whole conversation took less than two minutes of his time, it saved many weeks of work. During the breaks, we also talked to a few people about any ideas they might have to solve other issues, and what uses they might have for Zstd.

We presented our progress at BSDCan 2018 and there was a good deal of interest. Though there was still much to be done before it could be committed, the prototype showed how much benefit Zstd could provide to ZFS and FreeBSD.

Beyond the Prototype

After getting the initial functionality working, there were larger integration issues to address. How will this all integrate into ZFS? In the ZFS on-disk format, the compression type is stored in an 8-bit field in each block pointer. The top bit had already been borrowed to represent embedded block pointers, for the case where a block compresses so well (112 bytes), that it can be stored directly in the block pointer in place of the disk addresses and checksum, and therefore does not require its own allocation on disk. This means that no more than 127 compression algorithms are possible, and another bit may need to be borrowed in the same way in the future. A number of slots are already used: The value 0 does not actually mean no compression, it indicates that compression is inherited from the parent object. With levels for on, off, lzjb, empty (a whole block consisting entirely of zeros), gzip 1 through 9, ZLE, and LZ4, the first 15 values are already used. In the end, this Zstd patch introduced 41 additional compression levels (1-19, "fast" 1-9, "fast" 10-100 in increments of 10, "fast-500" and "fast-1000"), which could lead to very few possibilities left in the compression field in the on-disk format. After examining how the compression field in the block pointer is used, it became clear that the on-disk format only needs to map the compression setting to the correct decompression function, which is the same for all Zstd levels. At the time, it did not seem like it would be necessary to store the specific level of Zstd a block was compressed with.

After further work, it was discovered that sometimes we actually do need to know what level a block was compressed with. Namely, in the (presumably infrequent) case where the compressed ARC feature is disabled, the L2ARC would consistently fail with checksum errors. The L2ARC is a second-level cache that copies data at risk of being evicted from the primary ARC. By design, the L2ARC avoids the overhead of keeping its own copy of the checksum of each block, and instead refers to the checksum in the original block pointer. This means each block must be recompressed with the exact same settings before being written into the L2ARC. When reading back from the L2ARC, the block is checksummed and compared to the on-disk block and the original checksum. With the previous compression algorithms, there were no additional parameters to consider, but with Zstd, recompression at the default level would most likely generate a different output, and therefore mismatched checksums.

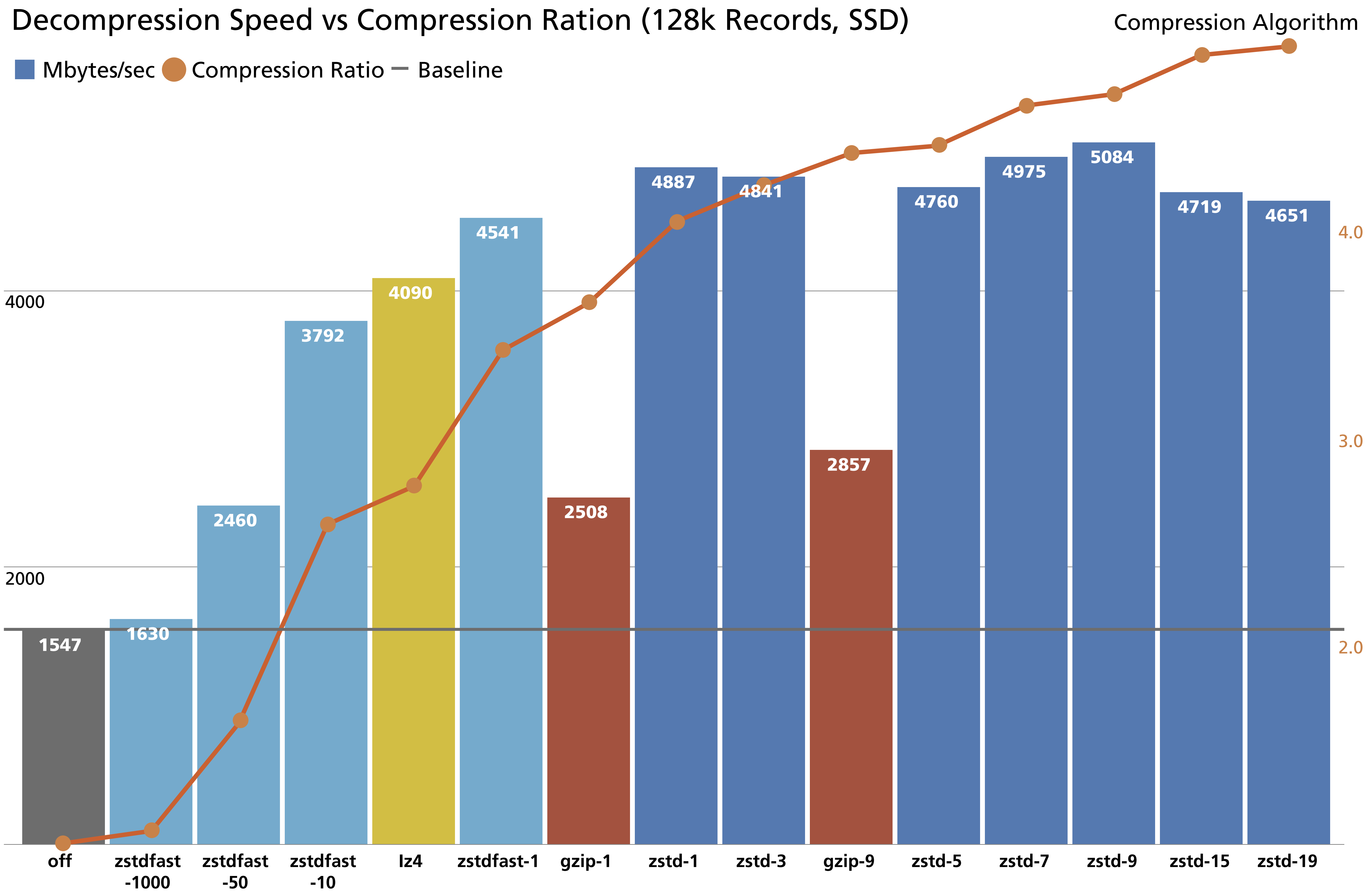
After further work, it was discovered that sometimes we actually do need to know what level a block was compressed with.

To solve this, we extended an existing concept used in LZ4, where the first 4 bytes of a compressed block on disk are used to store the compressed length of the block. Since allocations on disk will always be whole sectors, this allows LZ4 to avoid reading and attempting to decompress the random data in the slack space between the end of the compressed data and the end of the sector. Zstd compressed blocks use a larger header and store the version of Zstd and the level of compression in addition to the size. We decided to store the version of Zstd used to make it easier to upgrade the version of Zstd in the future, giving us the possibility to include multiple versions of the Zstd compression functions, so that a block could always be recreated if required. This is most likely to come in handy for the "NOP-write" feature: when a block is to be overwritten, ZFS can compare the checksum of the new block, and if it is the same as the old block, it does not need to rewrite the data. This type of operation is very common with Oracle databases and may also happen with certain types of backup software. If the original block is compressed with an older version of Zstd but now recompressed with a newer version, this could lead to a loss of this optimization. If ZFS is able to detect this situation, and attempt compression with the older version of Zstd, it can avoid the unexpected growth of snapshots of an Oracle database.

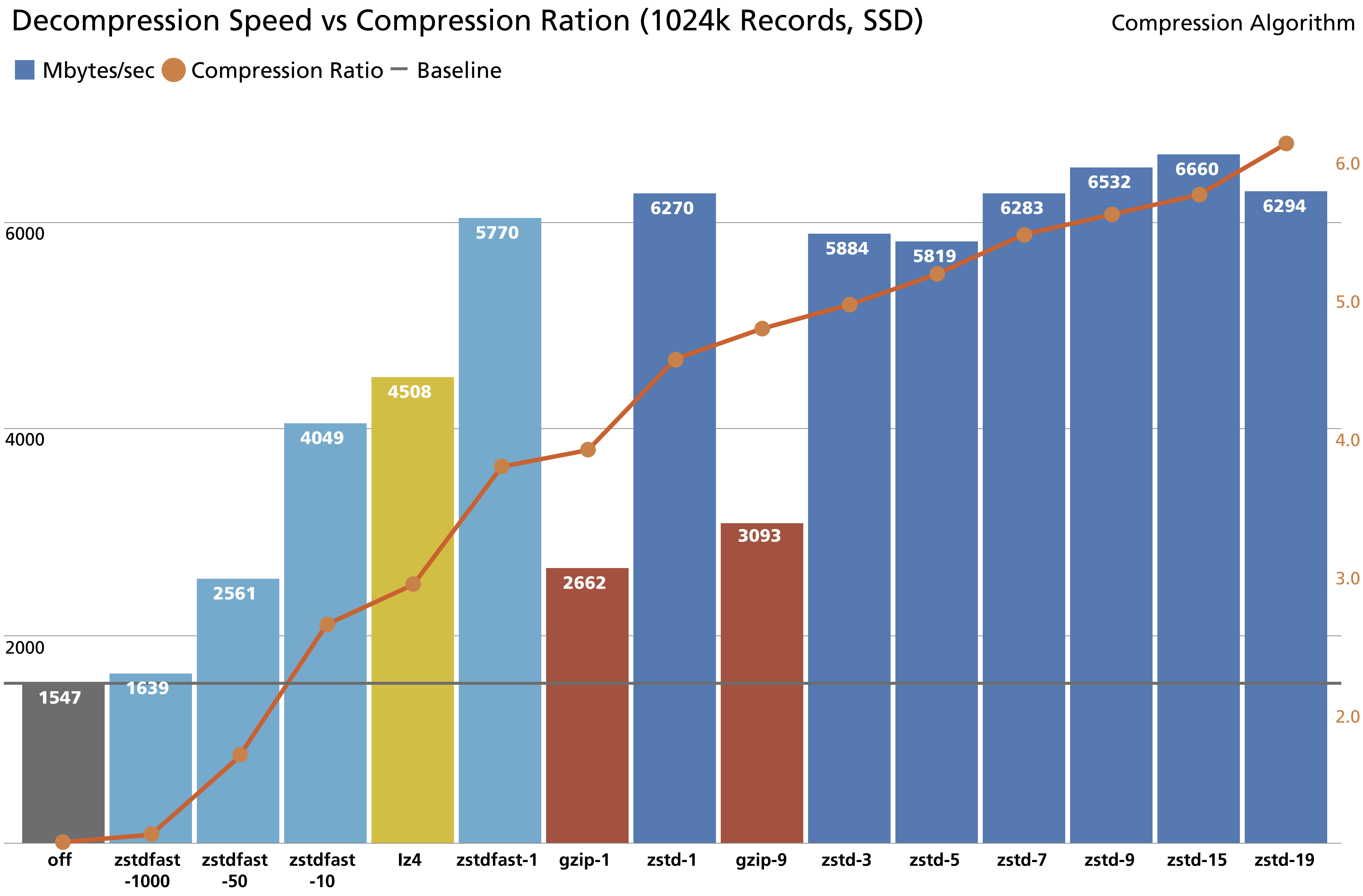
Where Zstd Shines

Zstandard provides a large selection of compression levels, allowing the storage administrator relatively fine-grained control over balancing performance and compression ratio. One of the main advantages of Zstd is that the decompression speed is independent of the compression level. For data that is written once but read many times, Zstd allows the use of the highest compression levels without a performance penalty. When writing large amounts of data, ZFS compresses each record individually, so it is able to take advantage of the many processor cores available on modern systems. Even when data is updated frequently, there are often performance gains that come from enabling compression. One of the biggest advantages comes from the compressed ARC feature--itself a recent improvement in ZFS. ZFS's Adaptive Replacement Cache (ARC) now caches the compressed version of the data in RAM and decompresses it each time it is requested. This allows the same amount of cache to store more (often much more) logical data and metadata, increasing the cache hit ratio, and improving performance for the most frequently and most recently accessed data. If upgrading from LZ4 to Zstd increases the on-disk compression ratio, those gains directly multiply the efficacy of every byte in the compressed ARC.

In the chart below, we compare storing a large uncompressed tarball of FreeBSD source code on ZFS using a variety of compression algorithms and levels. The test system used four striped SATA SSDs, the read speed without compression was limited by the available throughput of the underlying storage devices to around 1.5 GB/s, however, as the compression ratio of the data goes up, the read speed generally increases as well, since the limiting factor is still how fast the compressed data can be brought in from the underlying storage. Compared to gzip, Zstd decompresses much faster, and wastes few of these gains as it does not generally require more CPU time in decompression.


Decompression Speed vs Compression Ration (128k Records, SSD)

Interestingly, using a larger ZFS "record size" allows even greater ratios. The reason for this is ZFS compresses each record independently, so record size has a large impact on the possible compression gains; the larger the record, the more optimal the compression dictionary. gzip-9 sees the compression ratio increase from 4.3x to 4.7x, it only gains a modest 8% additional throughput, while Zstd-9 boots its ratio from 4.9x to 5.5x and gains 28% more performance, reaching more than four times the throughput the hardware is capable of.


Decompression Speed vs Compression Ration (1024k Records, SSD)

One thing to be aware of is that ZFS will not store a block compressed if the savings from compression do not result in the savings of at least one disk sector. For example, on a typical database filesystem, with a recordsize of 16 KB, if the compression ratio is 1.32x, resulting in the final block being 12.1 KB, it will still require the same four 4 KB sectors to be stored, so it will be less work to just store the data uncompressed. However, if the compression ratio is 1.34x, requiring 11.9 KB of storage space, this can be achieved with just three 4 KB sectors, so ZFS will use the compressed version. The compressionratio property of a dataset returns the average of all the records.

What's Next?

The integration of Zstd into ZFS has just begun and the future undoubtedly holds many improvements. Already, we have thoughts along these lines. For example, we expect using the advanced Zstd API to provide more hints about the maximum size of the input data could reduce memory usage and improve Zstd's ability to take advantage of "early abort," which we spoke of early in the article. There are likely a number of opportunities to optimize the way ZFS sets up and tears down Zstd compression contexts and to increase the reuse of these contexts with the Zstd reset API, which one would expect to significantly improve compression performance with small blocks.

Aside from continuing to optimize Zstd for ZFS, the next obvious evolution is to remove the need for the user to decide what Zstd level is best (there are 40 options to choose from after all). Instead, we envision a user simply setting compress=zstd-auto and ZFS dynamically adapts in some sensible way. When using Zstd from the command line, to compress a stream being sent over the network, the user can specify—adapt=min=3,max=10 and Zstd will vary the compression level based on how quickly the network buffer is emptied. This ensures that the compression is not a bottleneck by lowering the compression level if the network has available bandwidth, or conversely, by increasing the time spent on compression if the network is not able to keep up with the current compression level.

In ZFS, this would likely be modelled on the amount of "dirty" data (data waiting to be compressed and written to disk). When new data is written to ZFS, it will be compressed with the maximum compression level. If the rate of incoming writes is too high for ZFS to keep up with the requested level of compression, which results in the amount of dirty data steadily increasing, the compression level would lower incrementally, ideally settling on the maximum level that does not limit throughput. As always, the ZFS philosophy is to make sensible use of system resources while minimizing the need for adjustment and tweaking by the user.

When new data is written to ZFS, it will be compressed with the maximum compression level.

Conclusion

Zstd support shipped as part of the recently released OpenZFS 2.0, which is available as replacement for the base ZFS in FreeBSD 12.2 via the sysutils/openzfs package and is integrated into the FreeBSD 13.0 development branch.

I want to give a special thanks to everyone at the FreeBSD Foundation for the grant that made it possible to get this long-running project finished and merged in time for OpenZFS 2.0. Thanks also to Sebastian Gottschall, Kjeld Schouten-Lebbing, and Michael Niewöhner who did the Linux port, including the additional kmem compatibility code, and creating most of the tests included in the final patch. I also want to thank the team that worked to integrate FreeBSD support into the upstream OpenZFS repo, and everyone at the OpenZFS project. Lastly, my thanks also go out to everyone who tested and reviewed the many versions of the patches over the years until it was finally committed.

ALLAN JUDE is VP of Engineering at Klara Inc., a global FreeBSD Professional Services and Support company. He also hosts the premier weekly BSD podcast, BSDNow.tv and served on the FreeBSD Core team from 2016 to 2020. He is the co-author of *"FreeBSD Mastery: ZFS"* and *"FreeBSD Mastery: Advanced ZFS"* with Michael W. Lucas.

PRACTICAL PORTS

Can't Git Enough?

BY BENEDICT REUSCHLING

This column covers ports and packages for FreeBSD that are useful in some way, peculiar, or otherwise good to know about. Ports extend the base OS functionality and make sure you get something done or, simply, put a smile on your face. Come along for the ride, maybe you'll find something new.

You may be aware that FreeBSD 13.0 is the first major release cut from Git instead of Subversion. This has long been in the making with a lot of careful handling of the source that makes FreeBSD so valuable. I clearly remember a certain FreeBSD devsummit in Maarsen, Netherlands many years ago, where the FreeBSD project decided to finally jump on the Subversion bandwagon. Coming from CVS (and probably rcs before that, ask the people who've been around longer than I...), switching version control systems is certainly not easy. Especially when you have a history going back to the very first days at UC Berkeley, every single change is precious. You never know when you'll need to dig up some obscure historical fact because a device driver misbehaves, or a developer needs to know why an interface was implemented in a certain way. But switching version control systems is not just a technical task, it's a social one, too. It comes with requirements for convincing and bringing onboard the people who will use it after the switch has been done. There is plenty of controversy about Git and how it behaves. Relearning some of the concepts of version control systems and how Git does it is probably the best way to deal with it, along with an open-minded approach, of course.

> There is plenty of controversy about Git and how it behaves.

Certainly, Git was around the ports collection before the switch happened and many a developer has used it for years for their own personal projects or at work (sometimes without having a choice). People missing the "central-source-of-truth" approach from Subversion et al. can set up a Gitlab system using www/gitlab-ce. There are plenty of extra ports to keep you busy setting it up on a rainy, lockdown day. The graphical user interface hides many of the perceived warts of Git behind a user-friendly interface. From replacing files by uploading a changed version that turns it into a commit and push to reading the version history, all is done without needing to know a single Git command. This and other similar UIs like www/gitea, devel/cgit, devel/git-cola, make it easy for non-developers to keep track of documents in an office setting, no matter if it is IT-related or not. The little "time machine for files" is useful to pretty much anyone needing an old version of a file the way it was before your cat managed to not only walk across the keyboard, but also save the document in the process. Extra points for the rodent hunter when you were in vi at the time.

Software development never seems to be an easy task, so all the help one can get is welcome. I've always wondered how developers start: do they think of a name for their software first or start to hack on it right away? If you can't think of a good name and asdf, qwerty, and similar are already taken (we won't discuss them here, I promise), how about doing the old reverse-y thing? This must have inspired the author of devel/tig who wrote an ncurses-based terminal interface for Git. Because why not? Browsing your version-controlled tree is quick this way and staging your next change, looking at the history, writing that bad, one-word commit message is all possible (the latter is discouraged though--give the historians a bit more information about why you made that change at 4:20 in the morning).

Software development never seems to be an easy task, so all the help one can get is welcome.

More than once in my Unix class, I have told students that the Unix developers were lazy, but the good kind of laziness. What I mean is that they sat down, figured out how their colleague's computer could do the work much better and put a lot of effort into it. Once it was done, they could be lazy because the silicon was doing all the hard work. It is not lazy for the sake of laziness, which is probably the highest form of procrastination. But whatever it may be, if you are in the same camp, take a look at devel/lazygit. A terminal UI in Go which starts with a nice rant on its project page:

"Rant time: You've heard it before, Git is powerful, but what good is that power when everything is so damn hard to do? Interactive rebasing requires you to edit a damn TODO file in your editor? Are you kidding me? To stage part of a file, you need to use a command line program to step through each hunk, and if a hunk can't be split down any further and contains code you don't want to stage, you have to edit an arcane patch file by hand? Are you KIDDING me?! Sometimes you get asked to stash your changes when switching branches only to realize that after you switch and un-stash, there weren't even any conflicts, and it would have been fine to just check out the branch directly? YOU HAVE GOT TO BE KIDDING ME!"

But after the ranting, the developer sat down and made life better for everyone. See, everyone can do it. The UI is certainly nice enough to give it a try. Another such tool for command line aficionados is devel/glab. Written for Gitlab, it lets you leave the familiar territories of the web UI and stay in the terminal where the real fun is happening.

More often than not, source code involves collaboration with others. My code certainly got better when another set of eyes took a look at it. After rolling them back from the top of their heads, people were not shy to point out where I could do better, sharing their wisdom along the way. The occasional praise was also there, so I did not start my new career as a salami smuggler. Reviewing on Github (where not only the cool kids hang out these days, but pretty much anyone needing a repo for their files) is often done using the Gerrit code review tool. If you spend a lot of time there, consider installing devel/git-review to support your coding workflow and review with others.

Beginners can take a look at devel/easygit to make the experience a little less painful or overwhelming. For those who have used it for a long time and want to brag about how many lines of code were changed just because they commit early and often, devel/gitinspector might help. Just decorate your own office—and not the cafeteria—with the resulting stats, or people will quickly remind you that it is quality not the quantity that counts.

Git is not the only version control game in town, there have been and always will be others. Its popularity certainly can't be overlooked and it must have come from a feature or two that people were missing. What's the easiest thing I can think of if you want version control but don't want Git? Keep multiple drafts in your email, attach the files you were working with or paste the content in the message body. It is even distributed if you pick up your work from another computer by going into your webmail. Surely that has its downsides, as it will be a security nightmare if you give people access to your "repo." But maybe that experience will haunt you enough to give Git a first, second, or third chance. Pick up a book, do one of the many online tutorials. The old "learning by doing" is also very effective. There can never be enough people contributing to open-source projects after all. Commit yourself to it and don't forget to pick the cherries along the way.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. He serves on the board of directors of the FreeBSD Foundation as vice president. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Together with Allan Jude, he is host of the weekly bsdnow.tv podcast.

Events Calendar

BSD Events taking place through September 2021

BY ANNE DICKISON

Please send details of any FreeBSD related events or events that are of interest for FreeBSD users which are not listed here to freebsd-doc@FreeBSD.org.

Users with organizational software that uses the iCalendar format can subscribe to the FreeBSD events calendar which contains all of the events listed here.

June 2021 FreeBSD Developer Summit

June 9–11, 2021
VIRTUAL

Join us for the 2021 FreeBSD Developer Summit. The online event will consist of half-day sessions, taking place June 9–11, 2021. It's free to attend, but we ask that you register with the eventbrite system to gain access to the meeting room. In addition to vendor talks, we will also have discussion sessions. More information can be found on the wiki.

USENIX LISA21

June 1–3, 2021
VIRTUAL

LISA is the premier conference for operations professionals, where we share real-world knowledge about designing, building, securing, and maintaining the critical systems of our interconnected world. The Foundation is pleased to again be an Industry Partner for this event.

EuroBSDcon 2021

September 16–19, 2021
Vienna, Austria

EuroBSDcon is the European annual technical conference gathering users and developers working on and with 4.4BSD (Berkeley Software Distribution) based operating systems family and related projects. The CFP is now open. Submit your FreeBSD talks by May 26, 2021.

FreeBSD Fridays

https://freebsdfoundation.org/freebsd-fridays/
FreeBSD Fridays will begin again in May.
Past FreeBSD Fridays sessions are available at: https://freebsdfoundation.org/freebsd-fridays/

FreeBSD Office Hours

https://wiki.freebsd.org/OfficeHours
Join members of the FreeBSD community for FreeBSD Office Hours. From general Q&A to topic-based demos and tutorials, Office Hours is a great way to get answers to your FreeBSD-related questions.

Past episodes can be found at the FreeBSD YouTube Channel.