FreeBSD on Firecracker

BY COLIN PERCIVAL

A large amount of fantastic open source software originates from "scratching an itch". Such was the case with Firecracker: In 2014, Amazon launched AWS Lambda as a "serverless" compute platform: Users can provide a *function* — say, ten lines of Python code — and Lambda provides all of the infrastructure between an HTTP request arriving and the function being invoked to process the request and generate the response.

To provide this service efficiently and securely, Amazon needed to be able to launch virtual machines with minimal overhead. Thus was born Firecracker: A Virtual Machine Monitor which works with Linux KVM to create and manage "microVMs" with minimal overhead.

Why FreeBSD on Firecracker?

In June 2022, I started work on porting FreeBSD to run on Firecracker. My interest was driven by a few factors.

First, I had been doing a lot of work on speeding up the FreeBSD boot process and wanted to know the limits that could be reached with a minimal hypervisor.

Second, porting FreeBSD to new platforms always helps to reveal bugs — both in FreeBSD and on those platforms.

Third, AWS Lambda only supports Linux at present; I'm always eager to make FreeBSD more available in AWS (although adoption in Lambda is out of my control, Firecracker support would be a necessary precondition).

The largest reason, however, was simply *because it's there*. Firecracker is an interesting platform, and I wanted to see if I could make it work.

Porting FreeBSD to new platforms always helps to reveal bugs — both in FreeBSD and on those platforms.

Launching the FreeBSD Kernel

While Firecracker was designed for Lambda's needs — launching Linux kernels — there were patches available from 2020 that added support for the PVH boot mode in addition to "linuxboot". FreeBSD had support for PVH booting under Xen, so I decided to see if that would work.

Here I ran into the first problem: Firecracker could load the FreeBSD kernel into memory but couldn't find the address at which to start running the kernel (the "kernel entry point"). According to the PVH boot protocol, this value is specified in an ELF Note — a piece of special metadata stored in ELF (Executable and Linker Format) files. It turned out that there are two types of ELF Notes: PT_NOTEs and SHT_NOTEs, and FreeBSD wasn't providing the one Firecracker was looking for. A small change to the FreeBSD kernel linker script fixed this, and now Firecracker was able to start executing the FreeBSD kernel.

That lasted for about 1 microsecond.

Early Debugging

FreeBSD has wonderful debugging functionality, but if your kernel crashes before the debugger is initialized or the serial console is set up, you're not going to get much help. In this case, the Firecracker process exited, telling me that the FreeBSD guest hit a triple-fault — but that's all I knew.

It turned out, however, that this was enough information to get me started, given a bit of creativity. If the FreeBSD kernel execution reached a `hlt` instruction, the Firecracker process would keep running but use 0% of the host's CPU time (since it was virtualizing a halted CPU). As such, I could distinguish between "FreeBSD is crashing before this point" and "FreeBSD is crashing after this point" by inserting a `hlt` instruction — if Firecracker exited, I knew that it was crashing before reaching that instruction. Thus started a process I referred to as "kernel bisection" — rather than bisecting a list of commits to find one which introduced a bug (as in `git bisect`) I would do a binary search through the kernel startup code to find the line of code that was making FreeBSD crash.

Xen Hypercalls

The first thing I discovered in this process was Xen hypercalls. The PVH boot mode originated as the *Xen*/PVH boot mode, and FreeBSD's PVH entry point was, in fact, an entry point specifically for booting under Xen — and the code made the quite reasonable assumption that it was, indeed, running inside Xen and could thus make Xen hypercalls. KVM (which provides the kernel functionality used by Firecracker) is not Xen, of course, so it doesn't provide those hypercalls; attempting to use any of them resulted in the virtual machine crashing. As an initial workaround, I simply commented out all the Xen hypercalls; later, I added code to check `CPUID` for a Xen signature before making calls e.g., to write debugging output to the Xen debug console.

FreeBSD has wonderful debugging functionality, but if your kernel crashes before the debugger is initialized or the serial console is set up, you're not going to get much help.

There was one Xen hypercall that provided essential functionality, however: Retrieving the physical memory map. (Of course, inside a hypervisor, the "physical" memory is only virtually physical. It's turtles all the way down.) Here, we're saved by the fact that Xen/PVH was retroactively declared to be *version 0* of PVH boot mode: From *version 1* onwards, a pointer to the memory map is passed via the PVH `start_info` page (a pointer to which is provided in a register when the virtual CPU starts executing). I had to write code to make use of the PVH *version 1* memory map instead of relying on a Xen hypercall to get the same information, but that was easy enough.

Another related issue arose from how Xen and Firecracker arrange structures in memory: Whereas Xen loads the kernel first and then places the `start_info` page at the end, Firecracker placed the `start_info` page at a fixed low address and then loaded the kernel afterwards. This would have been fine but that FreeBSD's PVH code – having been written with Xen in mind — assumed that the memory immediately after the `start_info` page would be free for use as scratch space. Under Firecracker, that very quickly meant overwriting the initial kernel stack — not an optimal outcome! A change to FreeBSD's PVH code to assign scratch space after *all* the memory regions initialized by the hypervisor fixed this problem.

ACPI — or Lack Thereof!

On x86 platforms, FreeBSD normally makes use of ACPI to learn about (and in some cases control) the hardware on which it is running. In addition to discovering things via ACPI which we might commonly think of as "devices" — disks, network adapters, etc. — FreeBSD also learns about fundamental things like CPUs and interrupt controllers via ACPI.

Firecracker, being deliberately minimalist, does not implement ACPI, and FreeBSD gets upset when it can't figure out how many CPUs it has or where to find their interrupt controllers.

Fortunately, FreeBSD has support for the historic Intel MultiProcessor Specification, which provides this critical information via an "MPTable" structure; it's not part of the GENERIC kernel configuration, but for running in Firecracker, we want to use a stripped-down kernel configuration anyway, so it was easy to add `device mptable` to make use of what Firecracker provides.

Except… it didn't work. FreeBSD still couldn't find the information it needed! It turned out that Linux has bugs in how it finds and parses the MPTable structure — and Firecracker, being designed to boot Linux, provided the MPTable in a way that Linux supported but was not in fact compliant with the standard. FreeBSD, having an implementation independently written to follow the standard, failed both to find the (incorrectly located) MPTable and to parse the (invalid) MPTable once it was found.

So now FreeBSD has a new kernel option: You can add `options MPTABLE_LINUX_BUG_COMPAT` to your kernel configuration if you need bug-for-bug compatibility with Linux's MPTable handling — and with that, FreeBSD managed to boot a bit further in Firecracker.

You can add options MPTABLE_LINUX_BUG_COMPAT to your kernel configuration if you need bug-for-bug compatibility with Linux's MPTable handling.

Serial Console

One of the few emulated devices — as opposed to *virtualized* devices like the Virtio block and network devices — provided by Firecracker is the serial port. In fact, in a common configuration, when you launch Firecracker, the standard input and output of the Firecracker process become the serial port input and output of the VM, making it seem like the guest OS is just another process running inside your shell (which, in a certain sense, it is). At least, that's how it's supposed to work.

By this point in the process of bringing up FreeBSD inside Firecracker I was able to boot a FreeBSD kernel with a root disk compiled into the kernel image — I didn't have the virtualized disk driver working yet — and read all the console output from the kernel. After all the kernel console output, however, FreeBSD entered the userland portion of the boot process, and I got 16 characters of console output — and then it stopped.

Funnily enough, I'd seen that exact symptom over ten years earlier, when I was first getting FreeBSD working on EC2 instances. A bug in QEMU resulted in the UART not sending an interrupt when the transmit FIFO emptied; FreeBSD wrote 16 bytes to the UART and then wouldn't write anymore because it was waiting for an interrupt which never arrived. Modern EC2 instances run on Amazon's "Nitro" platform, but in the early days they used Xen and devices were emulated using code from QEMU. Somehow, a decade after this bug was fixed in QEMU, exactly the same bug was implemented in Firecracker; but luckily for me, the workaround I put into the FreeBSD kernel — `hw.broken_txfifo="1"` — was still available, and adding that loader tunable (which, since Firecracker loads the kernel directly without going through the boot loader, meant compiling the value into the kernel as an environment variable) fixed the console output.

I then found that the console *input* was also broken: FreeBSD didn't respond to anything I typed into the console. In fact, tracing the Firecracker process, I found that Firecracker wasn't even reading from the console — because Firecracker thought that the receive FIFO on the emulated UART was full. This turned out to be another bug in Firecracker: While initializing the UART, FreeBSD fills the receive FIFO with garbage to measure its size and then flushes the FIFO by writing to the FIFO Control Register. Firecracker didn't implement the FIFO Control Register, so it was left with a full FIFO and quite sensibly didn't try to read any more characters to put into it. Here, I added another workaround to FreeBSD: If `LSR_RXRDY` is still asserted after we try to flush the FIFO via the FIFO Control Register — which is to say, if the FIFO didn't empty as requested — we now proceed to read and discard characters one by one until the FIFO empties. With this, Firecracker now recognized that FreeBSD was ready to read more input from the serial port, and I had a working bidirectional serial console.

Virtio Devices

While a system without disks or network could be useful for some purposes, before we can do very much with FreeBSD, we're going to want those devices. Firecracker supports Virtio block and network devices and exposes them to virtual machines as `mmio` (memory-mapped I/O) devices. First step to getting these working in FreeBSD: Add `device virtio_mmio` to the Firecracker kernel configuration.

Next up, we need to tell FreeBSD how to find the virtualized devices. FreeBSD expected `mmio` devices to be discovered via FDT (Flattened Device Tree), which is a mechanism commonly used on embedded systems; but Firecracker passes device parameters via the kernel command line with directives such as `virtio_mmio.device=4K@0x1001e000:5`. Second step to getting these working in FreeBSD: Write code for parsing such directives and creating `virtio_mmio` device nodes. (Once we create the device node, FreeBSD's regular process for device probing kicks in and the kernel will automatically determine the type of Virtio device and hook up the appropriate driver.)

If we have multiple devices however — say, a disk device and a network device — another problem arises: Firecracker passes directives the way Linux expects — as a series of key=value pairs on the kernel command line — while FreeBSD parses the kernel command line as environment variables… meaning that if there were two `virtio_mmio.device=` directives passed on the command line, only one was retained. To fix this, I rewrote the early kernel environment parsing code to handle duplicate variables by appending a numbered suffix: We end up with `virtio_mmio.device=` for one device and `virtio_mmio.device_1=` for the second device.

With this, I finally had FreeBSD booting and discovering all of its devices — but one more problem arose with disk devices: If I shut down the virtual machine uncleanly, on the next boot the system would run `fsck` on the filesystem, and the kernel would panic. It turned out that `fsck` is one of very few things in FreeBSD that will cause non-page-aligned disk I/Os, and FreeBSD's Virtio block driver was causing a kernel panic when trying to pass unaligned I/Os to Firecracker.

When an I/O crosses a page boundary — as will happen with page-sized I/Os which aren't aligned to page boundaries — the physical I/O segments will typically not be contiguous; most devices can handle I/O requests which specify a series of segments of memory to be accessed. Firecracker, being extremely minimalist, does not do this: Instead, it accepts only a single data buffer — meaning that a buffer that crosses a page boundary can't simply be split into pieces the way it would with other Virtio implementations. Fortunately, FreeBSD has a system in place specifically for taking care of device complications like this: `busdma`.

Once I had FreeBSD up and running in Firecracker, it rapidly became clear that there were some improvements to be made.

This was probably the hardest part of getting FreeBSD running in Firecracker, but after several attempts, I think I finally got it right: I modified FreeBSD's Virtio block driver to use `busdma`, and now unaligned requests are "bounced" (aka. copied via a temporary buffer) in order to comply with the limitations of the Firecracker Virtio implementation.

Revealed Optimizations

Once I had FreeBSD up and running in Firecracker, it rapidly became clear that there were some improvements to be made. One of the first things I noticed was that, despite having 128 MB of RAM in the virtual machine I was testing, the system was barely usable, with processes being frequently killed due to the system running out of memory. The `top(1)` utility showed that almost half of system memory was in the "wired" state, which seemed odd to me; so I investigated further, and found that `busdma` had reserved 32 MB of memory for bounce pages. This was clearly far more than needed — given Firecracker's limitations and the fact that bounce pages are generally not allocated contiguously, each disk I/O should use at most a single 4 kB bounce page — and I was able to reduce this memory consumption to 512 kB with a patch to `busdma` which limited its bounce page reservations for devices which supported only a small number of I/O segments.

Once the system was more stable, I started paying attention to the boot process. If you're watching a system boot and there's suddenly a pause in the messages scrolling past, there's probably something happening at that point which is slowing down the boot process. Simple eyeballing of the boot process — and also the shutdown process — revealed several improvements:

- FreeBSD's kernel random number generator usually obtains entropy from hardware devices, but in virtual machines this may not be an effective source. As a backup source of entropy, on x86 systems we make use of the `RDRAND` instruction to obtain random values from the CPU; but we were only obtaining a very small amount of entropy on each request and were only requesting entropy once every 100 ms. Changing the entropy gathering system to request enough entropy to fully seed the Fortuna random number generator shaved 2.3 seconds off the boot time.
- When FreeBSD first boots, it records a Host ID for the system. This is typically obtained from hardware via the **`smbios.system.uuid`** environment variable, which the boot loader sets based on information from BIOS or UEFI. Under Firecracker, however, there is no boot loader — and thus no ID being provided. We had a fallback system in place that would generate a random ID in software on systems that didn't have a valid hardware ID; but we also printed a warning and waited 2 seconds to allow the user to read it. I changed this code to print the warning and wait 2 seconds if the hardware provided an *invalid* ID, but proceed silently and quickly if the hardware simply didn't provide an ID.
- IPv6 mandates that systems wait for "Duplicate Address Detection" before using an IPv6 address. In **`rc.d/netif`**, after bringing up interfaces, we were waiting for IPv6 DAD if any of our network interfaces had IPv6 enabled. There's just one problem with that: We *always* have IPv6 enabled on the loopback interface! I changed the logic to only waiting for DAD if we had IPv6 enabled on an interface *other than the loopback* interface and sped up the boot process by 2 seconds — if another system is trying to use the same IPv6 address as us on our `lo0`, we have bigger problems than an address collision!

IPv6 mandates that systems wait for "Duplicate Address Detection" before using an IPv6 address.

- When rebooting, FreeBSD printed a message ("**`Rebooting...`**") and then waited 1 second "for printf's to complete and be read". This seemed minimally useful, since people can usually tell that the system is rebooting — there is now a **`kern.reboot_wait_time sysctl`** which defaults to zero.
- When shutting down or rebooting, the FreeBSD BSP (CPU #0) waits for the other CPUs to signal that they have stopped… and then waited an extra 1 second to make sure that they had a chance to stop. I removed the extra second of wait time.

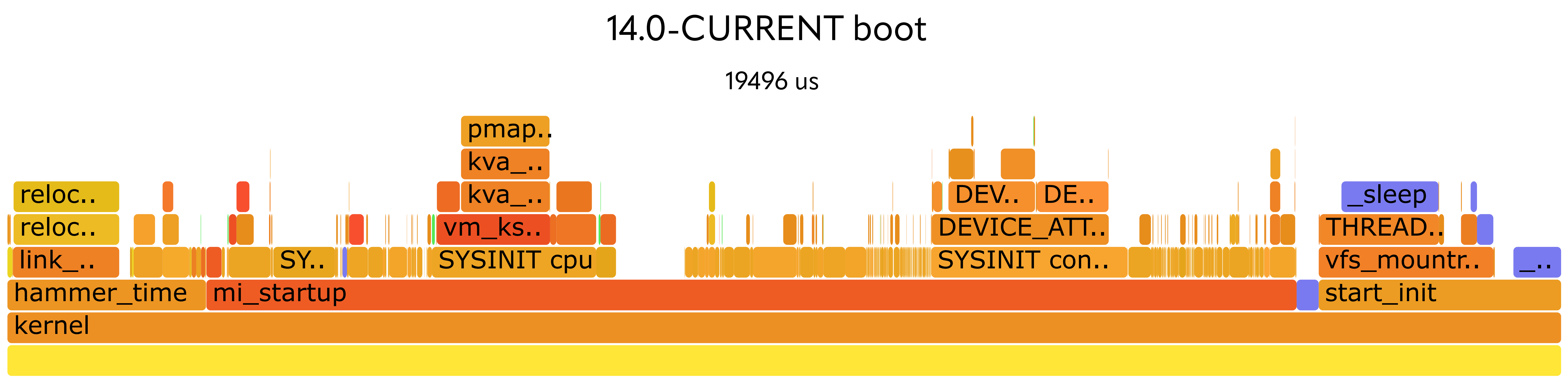
Once the low-hanging fruit was out of the way, I pulled out TSLOG and started looking at flamecharts of the boot process. Firecracker is a great environment for doing this, for two reasons: First, the minimalist environment eliminates a lot of noise, making it much easier to see what's left behind; and second, having Firecracker launch virtual machines extremely quickly made it possible to test changes to the FreeBSD kernel very rapidly — often well under 30 seconds to build a new kernel, launch it, and generate a new flamechart.

Investigation with TSLOG revealed a number of available optimizations:

- `lapic_init` had a 100000-iteration loop to calibrate how long `lapic_read_icr_lo` took to execute; cutting this down to 1000 iterations shaved off 10 ms.
- `ns8250_drain` called DELAY after each character was read; changing this to check for `LSR_RXRDY` and only DELAYing if nothing was already available to be read shaved off 27 ms.
- FreeBSD makes use of a CPUID leaf which most hypervisors use to advertise the TSC and local APIC clock frequencies; Firecracker, unlike VMWare, QEMU, and EC2, did not implement this. Adding support for this CPUID leaf to Firecracker shaved 20 ms off the FreeBSD boot time.
- FreeBSD was setting `kern.nswbuf` (which controls the number of buffers allocated for a variety of temporary purposes) to 256 regardless of the size of the system; changing this to `32 * mp_ncpus` shaved 5 ms off the boot time on a small (1 CPU) virtual machine.
- FreeBSD's `mi_startup` function, which kicks off machine-independent system initialization routines, was using a bubblesort to order the functions it called; while this was reasonable in the 90s given the small number of routines needing to be ordered at that point, there are now over 1000 such routines and the bubblesort was getting slow. Replacing it with a quicksort will save 2 ms. (Not yet committed at press time.)
- FreeBSD's `vm_mem` initialization routine was initializing `vm_page` structures for all available physical memory. Even on a relatively small VM with 128 MB of RAM, this meant initializing 32768 such structures — and took a few ms. Changing this code to initialize `vm_page` structures "lazily" as the memory is allocated for use will save 2 ms. (Not yet committed at press time.)
- Firecracker was allocating VM guest memory via an anonymous mmap, but Linux was not setting up the paging structures for the entire VM guest address space. As a result, the first time any page was read, a fault would occur taking roughly 20,000 CPU cycles to be resolved while Linux mapped in a page of memory. Adding the `MAP_POPULATE` flag to Firecracker's `mmap` call will save 2 ms. (Not yet committed at press time.)

Current Status

FreeBSD boots under Firecracker — and does so extremely quickly. Including uncommitted patches (to FreeBSD and also to Firecracker), on a virtual machine with 1 CPU and 128 MB of RAM, the FreeBSD kernel can boot in under 20 ms; a flame chart of the boot process appears below.



There is still work to be done: In addition to committing the patches mentioned above and getting PVH boot mode support merged to "mainline" Firecracker, there's a significant amount of "cleanup" work to be done. Due to the history of PVH boot mode originating from Xen, the code used for PVH booting is still mixed up with Xen support; separating those will simplify things significantly. Similarly, it's currently impossible to build a FreeBSD arm64 kernel without PCI or ACPI support; finding the bogus dependencies and removing them will allow for a smaller FreeBSD/Firecracker kernel (and also shave off a few more microseconds from the boot time – we spend 25 us checking to see if we need to reserve memory for Intel GPUs).

More aspirationally, it would be great to see if Firecracker could be ported to run on FreeBSD — at a certain point, a virtual machine is a virtual machine, and while Firecracker was written to use Linux KVM, there's no fundamental reason why it shouldn't be possible to make it use the kernel portion of FreeBSD's bhyve hypervisor instead.

Anyone wanting to experiment with FreeBSD in Firecracker can build a FreeBSD 14.0 kernel with the amd64 `FIRECRACKER` kernel configuration, and check out the feature/pvh branch from the Firecracker project; or if that branch no longer exists it means the code has been merged into the mainline Firecracker tree.

If you try out FreeBSD on Firecracker — especially if you end up using it in production — please let me know! I started this project mainly out of interest, but I'd love to hear if it ends up being useful.

COLIN PERCIVAL has been a FreeBSD developer since 2004 and was the project's Security Officer from 2005 to 2012. In 2006, he founded the Tarsnap online backup service, which he continues to run. In 2019, in recognition of his work bringing FreeBSD to EC2, he was named an Amazon Web Services Hero.