Optimizing QEMU boot time

Richard W.M. Jones Red Hat Inc. rjones@redhat.com

Abstract

Everyone knows that containers are really fast and lightweight, and full virtualization is slow and heavyweight ... Or that's what we thought, until Intel demonstrated full Linux virtual machines booting as fast as containers and using as little memory. Intel's work used kvmtool and a customized, cut down guest kernel. Can we do the same using libvirt, QEMU, SeaBIOS, and an off the shelf Linux distro kernel? The short answer is *no*, but we can get pretty close, and it was an exciting journey learning about unexpected performance roadblocks, developing tools to measure the boot process, and shaving off milliseconds all over the place. The work has practical significance because it will allow us to deploy secure containers, protected by hardware virtualization. Even if you never plan to use containers, you're still benefiting from a faster QEMU experience.

1 Intel Clear Linux

Intel's Clear Linux means a lot of different things to different people. I'm only going to talk about a narrow aspect of it, usually known as "Clear Containers", but if other people talk about Intel Clear Linux they might be talking about a Linux distribution, OpenStack or graphics technologies.

LWN has a useful and relatively recent introduction to Clear Containers `https://lwn.net/Articles/644675/`.

Until recently Intel hosted a Clear Containers demo. If you downloaded it and ran `bash ./boot.sh` then it booted into a full Linux VM in about 150ms, and using 20 MB of RAM.

Intel are using this technology along with a customized Docker driver to run Docker containers safely inside a VM. The overhead (150ms / 20 MB) is very attractive since it doesn't impact on the density that containers give you. It's also aligned with Intel's interests, since they are selling chips with VT, VT-d, EPT, VPID and so on and they need people to use those features.

The Clear Containers demo uses `kvmtool` with several non-upstream patches such as for DAX and 64 bit guests. Since first demonstrating Clear Containers, Intel has worked on getting vNVDIMM (needed for DAX) into QEMU.

The Clear Containers demo from last year uses a patched Linux kernel. There are many non-upstream patches. More importantly they use a custom, cut down configuration where many subsystems not used by VMs are cut out entirely.

2 Real Linux distros use QEMU

Can we do the same sort of thing in our Linux distros? Let's talk about some things that constrain us in Fedora.

We'd prefer to use QEMU over kvmtool. QEMU isn't really "bloated". It's featureful, but (generally) if you're not using those features they don't slow things down.

We *can't* use the heavily patched and customized kernel. Fedora is strictly "upstream first". Fedora also ships a single kernel image for baremetal, virtual machines and all other uses, since building and maintaining multiple kernels is a huge pain.

3 Stating the problem

What we want to do is to boot up and shut down a modern Linux kernel in a KVM virtual machine on a modern Linux host. Inside the virtual machine we will eventually want to run our Docker container. However I am just concentrating on the overhead of the boot and shutdown.

Conveniently – and also the reason I'm interested in this problem – libguestfs does almost the same thing. It starts up and shuts down a small Linux-based appliance. If you have `guestfish` installed, then you can try running the command below (several times so you have a warm cache). Add `-v -x` to the command line to see what's really going on.

```
$ guestfish -a /dev/null run
```

4 Measurements

The first step to improving the situation is to build tools that can accurately measure the time taken for each step in the boot process.

Booting a Linux kernel under QEMU using the `-kernel` option looks like table 1.

Table 1: Steps run when you use QEMU `-kernel`

query QEMU's capabilities
run QEMU
run SeaBIOS
run the kernel
run the initramfs
load kernel modules
mount and pivot to the root filesystem
run `/init`, `udevd` etc
perform the desired task
shutdown
exit QEMU

How do you know when SeaBIOS starts or various kernel events happen?

I started out looking at various QEMU tracing options, but ended up using a very simple technique: Attach a serial console to QEMU, timestamp the messages as they arrive, and use regular expression string matches to find significant events.

The three programs I wrote (two in C and one in Perl) use libguestfs as a convenient framework, since libguestfs has the machinery already for creating VMs, initramfses, capturing serial console output etc. They are:

- `boot-benchmark`

  `boot-benchmark` runs the boot up sequence repeatedly, throwing away the first few runs (to warm the cache) and collecting the mean test time and standard deviation.

  ```
  $ ./boot-benchmark
  Warming up the libguestfs cache ...
  Running the tests ...

  test version: libguestfs 1.33.29
  ```

```
  test passes: 10
host version: Linux moo.home.annexia.org 4.4.4-301.fc23.x86_64 #1 SMP
    host CPU: Intel(R) Core(TM) i7-5600U CPU @ 2.60GHz
     backend: direct               [to change set $LIBGUESTFS_BACKEND]
        qemu: /home/rjones/d/qemu/x86_64-softmmu/qemu-system-x86_64
qemu version: QEMU emulator version 2.6.50, Copyright (c) 2003-2008
         smp: 1                    [to change use --smp option]
     memsize: 500                  [to change use --memsize option]
      append:                      [to change use --append option]

Result: 568.2ms ±8.7ms
```

- `boot-benchmark-range.pl`

  `boot-benchmark-range.pl` is a wrapper script around `boot-benchmark` which lets you benchmark across a range of commits from some other project (eg. QEMU or the kernel). You can easily see which commits are causing or solving performance problems as in the example below:

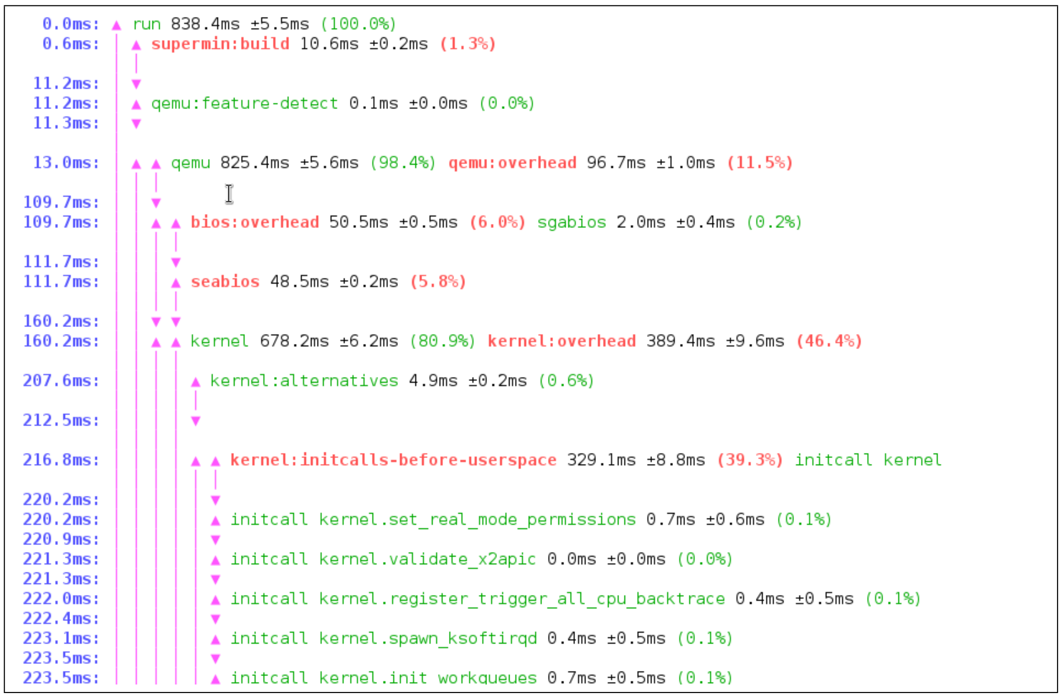
  ```
  $ ./boot-benchmark-range.pl ~/d/qemu 3123bd8^..8e86aa8
  da34fed hw/ppc/spapr: Fix crash when specifying bad[...]
  1666.8ms ±2.5ms

  3123bd8 Merge remote-tracking branch 'remotes/dgibson/[...]
  1658.8ms ±4.2ms

  f419a62 (origin/master, origin/HEAD, master) usb/uhci: move[...]
  1671.3ms ±17.0ms

  8e86aa8 Add optionrom compatible with fw_cfg DMA version
  1013.7ms ±3.0ms ↑ improves performance by 64.9%
  ```

- `boot-analysis`

  `boot-analysis` performs multiple runs of the boot sequence. It enables the QEMU serial console (and other events from libguestfs), timestamps the events, and then presents the sequence graphically as shown in figure 1. Also shown are mean times and standard deviations and percentage of the total run time.

  This test also prints which activities took the longest time, see figure 2.

The source for these tools is here: `https://github.com/libguestfs/libguestfs/tree/master/utils`.



```
  0.0ms: ▲ run 838.4ms ±5.5ms (100.0%)
  0.6ms: │ ▲ supermin:build 10.6ms ±0.2ms (1.3%)
         │ │
 11.2ms: │ ▼
 11.2ms: │ ▲ qemu:feature-detect 0.1ms ±0.0ms (0.0%)
 11.3ms: │ ▼

 13.0ms: │ ▲ ▲ qemu 825.4ms ±5.6ms (98.4%) qemu:overhead 96.7ms ±1.0ms (11.5%)
         │ │ │
109.7ms: │ │ ▼
109.7ms: │ │ ▲ ▲ bios:overhead 50.5ms ±0.5ms (6.0%) sgabios 2.0ms ±0.4ms (0.2%)
         │ │ │ │
111.7ms: │ │ │ ▼
111.7ms: │ │ │ ▲ seabios 48.5ms ±0.2ms (5.8%)
         │ │ │ │
160.2ms: │ │ ▼ ▼
160.2ms: │ │ ▲ ▲ kernel 678.2ms ±6.2ms (80.9%) kernel:overhead 389.4ms ±9.6ms (46.4%)

207.6ms: │ │ │ │ ▲ kernel:alternatives 4.9ms ±0.2ms (0.6%)
         │ │ │ │ │
212.5ms: │ │ │ │ ▼

216.8ms: │ │ │ │ ▲ ▲ kernel:initcalls-before-userspace 329.1ms ±8.8ms (39.3%) initcall kernel
         │ │ │ │ │ │
220.2ms: │ │ │ │ │ ▼
220.2ms: │ │ │ │ │ ▲ initcall kernel.set_real_mode_permissions 0.7ms ±0.6ms (0.1%)
220.9ms: │ │ │ │ │ ▼
221.3ms: │ │ │ │ │ ▲ initcall kernel.validate_x2apic 0.0ms ±0.0ms (0.0%)
221.3ms: │ │ │ │ │ ▼
222.0ms: │ │ │ │ │ ▲ initcall kernel.register_trigger_all_cpu_backtrace 0.4ms ±0.5ms (0.1%)
222.4ms: │ │ │ │ │ ▼
223.1ms: │ │ │ │ │ ▲ initcall kernel.spawn_ksoftirqd 0.4ms ±0.5ms (0.1%)
223.5ms: │ │ │ │ │ ▼
223.5ms: │ │ │ │ │ ▲ initcall kernel.init_workqueues 0.7ms ±0.5ms (0.1%)
```

Figure 1: boot-analysis timeline

Only now that we have the right tools to hand can we work out what activities take time.

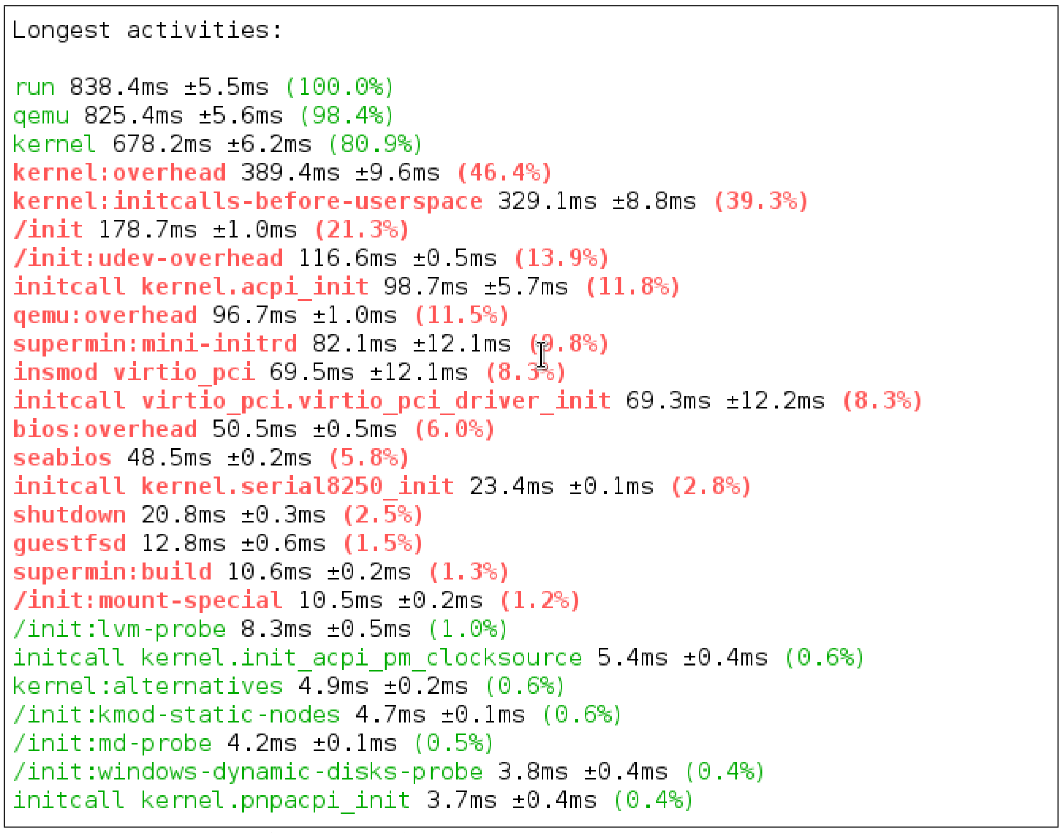
For consistency, all times displayed by the tool are in milliseconds (ms), and I try to use the same convention in this paper.

In this paper I'm using times based on my laptop, an Intel®Core^{TM}i7-5600U CPU @ 2.60GHz (Broadwell U). This does of course mean that these results won't be exactly reproducible, but it is hoped that with similar hardware you will get times that differ only by a scale factor.

5 glibc

Surprisingly the first problem is glibc. QEMU links to over 170 libraries, and that number keeps growing. A simple `qemu -version` takes up to 60ms, and examining this with `perf` showed two things:

- Ceph had a bug where it ran some `rdtsc` benchmarks in a constructor function. This is now fixed.
- The glibc link loader is really slow when presented with lots of libraries



```
Longest activities:

run 838.4ms ±5.5ms (100.0%)
qemu 825.4ms ±5.6ms (98.4%)
kernel 678.2ms ±6.2ms (80.9%)
kernel:overhead 389.4ms ±9.6ms (46.4%)
kernel:initcalls-before-userspace 329.1ms ±8.8ms (39.3%)
/init 178.7ms ±1.0ms (21.3%)
/init:udev-overhead 116.6ms ±0.5ms (13.9%)
initcall kernel.acpi_init 98.7ms ±5.7ms (11.8%)
qemu:overhead 96.7ms ±1.0ms (11.5%)
supermin:mini-initrd 82.1ms ±12.1ms (9.8%)
insmod virtio_pci 69.5ms ±12.1ms (8.3%)
initcall virtio_pci.virtio_pci_driver_init 69.3ms ±12.2ms (8.3%)
bios:overhead 50.5ms ±0.5ms (6.0%)
seabios 48.5ms ±0.2ms (5.8%)
initcall kernel.serial8250_init 23.4ms ±0.1ms (2.8%)
shutdown 20.8ms ±0.3ms (2.5%)
guestfsd 12.8ms ±0.6ms (1.5%)
supermin:build 10.6ms ±0.2ms (1.3%)
/init:mount-special 10.5ms ±0.2ms (1.2%)
/init:lvm-probe 8.3ms ±0.5ms (1.0%)
initcall kernel.init_acpi_pm_clocksource 5.4ms ±0.4ms (0.6%)
kernel:alternatives 4.9ms ±0.2ms (0.6%)
/init:kmod-static-nodes 4.7ms ±0.1ms (0.6%)
/init:md-probe 4.2ms ±0.1ms (0.5%)
/init:windows-dynamic-disks-probe 3.8ms ±0.4ms (0.4%)
initcall kernel.pnpacpi_init 3.7ms ±0.4ms (0.4%)
```

Figure 2: boot-analysis longest activities

and lots of symbols.

The second problem is intractable. We can't link to fewer libraries, because each of those libraries represents some feature that someone wants, like Ceph, or Gtk support (though if you remove the Gtk dependency the link time reduces substantially). And the link loader is bound by all sorts of obscure ELF rules (eg. symbol interposition) which we don't need but cannot avoid and make things slow.

When I said earlier that QEMU features don't slow things down, this is an exception.

We can run QEMU fewer times. There are several places where we need to run QEMU. Obviously one place is where we start the virtual machine, and the overhead there cannot be avoided. But also we perform QEMU feature detection by running commands like `qemu -help` and `qemu -devices \?` and libguestfs now caches that output.

6 QEMU

Libguestfs, Intel Clear Containers, and any future Docker container support we build will use `-kernel` and `-initrd` or their equivalent. In QEMU up to 2.6 on x86-64 this was implemented using an interface called `fw_cfg` and a PIO loop, and that is very slow. To load the kernel and very small initrd used by libguestfs takes around 700ms. In QEMU 2.7 we have added a pseudo-DMA mode which makes this step almost instant.

To see debugging messages from the kernel and to collect our benchmark results, we have to use an emulated 16550A UART (serial port). Virtio-console exists but isn't a good replacement because it can't be used to get BIOS and very early kernel messages. The UART is slow. It takes about 4μs per character, or approximately 1ms for 3 lines of text. Enabling debugging changes the results subtly.

To get serial console output from the BIOS, we use a Google-contributed option ROM called SGABIOS. It quickly became clear that SGABIOS introduced a 260ms boot delay. This happened because it expects to be talking to a real serial terminal, so it sends control sequences to query the width and height of this "terminal". These weren't being answered by the actual reader (libguestfs simply reads). The solution was to modify libguestfs to respond to the control sequence with a dummy reply, which reduced the delay to almost nothing.

7 libvirt

Libguestfs can optionally use libvirt to manage the QEMU process. When I did this it was obvious that libvirt was adding a (precisely) 200ms delay. I tracked this down to a poorly implemented polling loop in libvirt, waiting for the QEMU monitor socket to be created by QEMU. I fixed it by changing the loop to use exponential backoff. A better fix would involve passing pre-created file descriptors to QEMU.

8 SeaBIOS

SeaBIOS wastes time probing for boot devices even though we will use the `linuxboot` option ROM to boot (via `-kernel`). By building a `bios-fast.bin` variant of SeaBIOS with many unused features disabled we can reduce the time spent inside the BIOS from about 63ms to about 19ms.

9 kernel

PCI probing is slow, taking around 95ms for a guest with just two virtio-scsi drives. It turns out that it's not the scanning of the PCI device space which is slow, but the initialization of each device as it is found. QEMU's i440fx machine model exports some legacy devices which cannot be switched off, and that is unhelpful.

I implemented experimental support for parallel PCI probing using the kernel "async" feature. With 1 vCPU this slows things down very slightly as expected. With 4 vCPUs performance improved by about 30%. Unfortunately we can't use it because of the next point.

You would think multiple vCPUs would be better and faster than 1 vCPU, but that is not the case. It actually has a large negative impact on performance. Switching from 1 to 4 vCPUs increases the boot time by over 200ms. About 25ms is spent starting each secondary CPU (in `check_tsc_sync_target`). This can be avoided by setting `tsc.reliable=1` but no one can tell me if this is safe. But most of the extra time just disappears between the cracks – for example, PCI probing just slows down, but for no readily apparent reason. It seems as if the overhead of spinlocks or RCU or whatever hurts general performance. Or perhaps there is some scheduling problem on the host since it only has 4 physical CPUs.

When the kernel runs, it does some BIOS stuff, and there's a long delay (about 80ms) before `start_kernel` is entered.

Another unavoidable overhead is `ftrace` which must modify every function in the kernel. This takes 20ms. You can't disable ftrace at run time, the only option is to compile it out, but that breaks so many useful features that we'd never persuade a distro kernel to do that.

If your kernel has crypto functions, then it will spend 18ms testing them at boot. Herbert Xu accepted my patch to add a `cryptomgr.notests` flag which bypasses this.

As we are presenting an emulated 16550A UART, `serial_8250_init` runs, and this spends 25ms checking that the UART is really a 16650A (does it work, does it have a FIFO?), and (unsurprisingly) yes it is. This is a totally useless waste of time, but I have not managed to come up with a patch or even with an approach for how to avoid this that is acceptable upstream.

But the main problem is none of the above. It's simply the small amount of time taken to run many many initcalls. For a distro kernel this can be around 690ms (with serial debugging enabled which exaggerates the effect somewhat). One way to avoid that would be to compile some sort of custom kernel, and even though this approach is not acceptable for Fedora I did explore this, trying both a cut down distro kernel, and also a super-minimal kernel.

- The cut down distro kernel works by removing any subsystem that has a $>$ 1ms initcall overhead. These include:
  - auditing
  - big_key
  - ftrace
  - hugetlbfs
  - input_leds
  - joydev
  - joysticks
  - keyboards
  - kprobes
  - libata
  - mice
  - microcode
  - netlabel
  - profiling support
  - quota
  - rtc_drv_cmos
  - sound card support
  - tablets
  - touchscreens

  - USB
  - zbud
  - zswap

  That reduces the time taken running initcalls before userspace by about 20%. There is some scope for reducing this a bit more by going even further down the "long tail" of subsystems.

- For my second test I started with an absolutely minimal kernel config (`allnoconfig`), and built up the configuration until I got something that booted. That reduces the time taken running initcalls before userspace by about 60% (down to 288ms).

With a minimal kernel, we can get total boot times down to the 500-600ms range, but not any lower.

10 udev

udevd takes about 130ms to populate `/dev`.

The rules are monolithic, entwined together and resist modification, and starting a new set of rules from scratch looks like it would be a constant game of catch up.

11 initrd

We use a program called supermin (`http://libguestfs.org/supermin.1.html`) to construct the initrd which is responsible for loading enough kmods to mount the real root filesystem and pivoting into it.

Because of PIO loading of the initrd in earlier versions of QEMU, it was very important to construct as small an initrd as possible, and supermin was not doing a very good job of that. However once I started to analyze the situation there were some easy wins (now all upstream):

- We were adding all virtio kmods to the appliance plus any dependencies, with the starting set being constructed using the wildcard "`virtio*.ko`". The wildcard pulls in `virtio-gpu.ko` which depends on `drm.ko` and both are quite large. Since we are only interested in non-graphical VMs, I was able to blacklist `virtio-gpu.ko` and that reduced the total size of the initrd.

- We use a small C init program to load the kmods and mount the root filesystem, and this must be statically linked so we don't have to include a separate libc in the initrd. However glibc produces enormous static binaries (800KB+). Switching to using dietlibc allows us to build the same program to a 22KB binary, about $\frac{1}{40}$th of the size.
- We initially used xz-compressed kmods. These are smaller, reducing PIO loading time (but making not a lot of difference to DMA) but they are very slow to decompress. Switching to using uncompressed kmods produced a small reduction in boot time, and simplified the init code.
- Stripping kmods (with `strip -g`) is very important for reducing the size of the initrd.

The resulting initrd is about 126KB for the minimal kernel, or 347K for the standard Fedora kernel.

12 libguestfs

Finally there is libguestfs itself which glues everything together and provides the initial `/init` script. There were several savings to be made:

- When we are not debugging, we were still reading the verbose kernel output over the slow UART, and then throwing it away. The solution was to add the `quiet` option. That reduced boot time by about 1,000ms, the single largest saving.
- We used to run the `hwclock` command. With kvmclock it turns out this is not necessary, and removing it saved 300ms.
- We used to run `qemu -help` and `qemu -version`. Drew Jones pointed out the obvious: the help output contains the version number, so that reduces the number of times we need to run QEMU and suffer the glibc slow link loader overhead (and in the final version of libguestfs we also memoize QEMU output, reducing it further).
- We used to run SGABIOS unconditionally, but it is only necessary to use it when debugging. When we're not debugging we can omit it and save loading it at all.

- Running `ldconfig` in the appliance to update the link loader cache took 100ms, but we found a way that we don't need to run it at all.

13 Memory usage and DAX

I was pleasantly surprised that Intel had implemented a virtual NVDIMM, and ext4 + DAX is also working in modern kernels, and it was a relatively trivial job to implement DAX.

However I'm not certain that the benefits are clear, nor that I'm measuring things correctly.

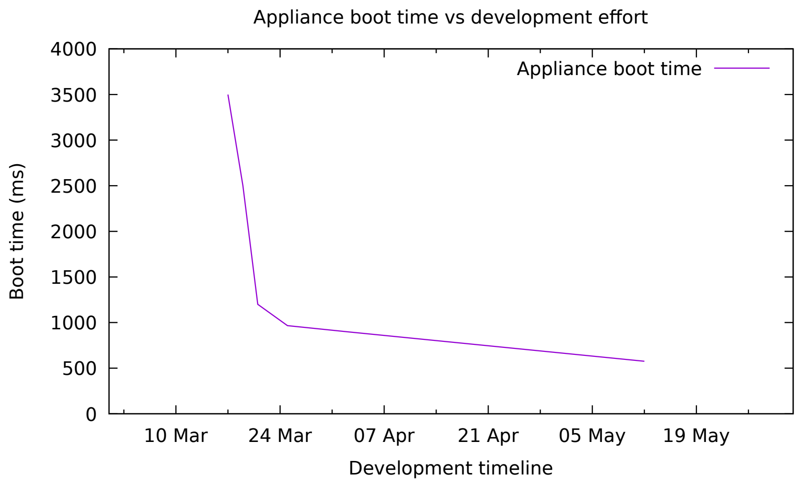
Inside the guest you can run `free -m` with and without DAX:

```
              total     used     free  shared buff/cache available
Without DAX:    485        3      451       1         30       465
With DAX:       485        3      469       1         12       467
```

The MaxRSS of QEMU reduces by about 5 MB when DAX is enabled.

14 Conclusions

This graph is just for a bit of fun:



There were a few false starts at the beginning of March (2016) where I was exploring how we might benchmark QEMU. But once I had written the right tools to analyze the boot process, two quick wins brought the time down from 3.5 seconds to 1.2 seconds in the space of a few days. It's worth noting that the libguestfs appliance had been booting in approx. 3-4 seconds for literally half a decade.

Getting the time under 600ms took a few weeks longer, and without some breakthrough in the kernel or udev, I cannot see us getting the time under 500ms.

Performance is everyone's job, but it sometimes feels like few people care about a use case which is considered esoteric. Yet this does affect everyone:

- If we can use virtualization as an extra layer of security around operations, whether that is Docker, or Qubes, or libvirt-sandbox, or libguestfs, that benefits everyone.
- The same concerns about boot speed are raised over and over again by the embedded community. If your digital camera is slow to switch on, it might be running initcalls for subsystems that it will never use. (Many references here: `http://elinux.org/Boot_Time`)

Hopefully this paper will persuade developers to think twice before adding an unnecessary delay loop, inserting a useless boot splash screen, or creating another initcall.