Optimising boot time on Raspberry Pi Single Board Computers

Colophon

Release	1
Build date	16/07/2026
Build version	913ce7cd7d00

Legal disclaimer notice

TECHNICAL AND RELIABILITY DATA FOR RASPBERRY PI PRODUCTS (INCLUDING DATASHEETS) AS MODIFIED FROM TIME TO TIME ("RESOURCES") ARE PROVIDED BY RASPBERRY PI LTD ("RPL") "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW IN NO EVENT SHALL RPL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THE RESOURCES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

RPL reserves the right to make any enhancements, improvements, corrections or any other modifications to the RESOURCES or any products described in them at any time and without further notice.

The RESOURCES are intended for skilled users with suitable levels of design knowledge. Users are solely responsible for their selection and use of the RESOURCES and any application of the products described in them. User agrees to indemnify and hold RPL harmless against all liabilities, costs, damages or other losses arising out of their use of the RESOURCES.

RPL grants users permission to use the RESOURCES solely in conjunction with the Raspberry Pi products. All other use of the RESOURCES is prohibited. No licence is granted to any other RPL or other third party intellectual property right.

HIGH RISK ACTIVITIES. Raspberry Pi products are not designed, manufactured or intended for use in hazardous environments requiring fail safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, weapons systems or safety-critical applications (including life support systems and other medical devices), in which the failure of the products could lead directly to death, personal injury or severe physical or environmental damage ("High Risk Activities"). RPL specifically disclaims any express or implied warranty of fitness for High Risk Activities and accepts no liability for use or inclusions of Raspberry Pi products in High Risk Activities.

Raspberry Pi products are provided subject to RPL's Standard Terms. RPL's provision of the RESOURCES does not expand or otherwise modify RPL's Standard Terms including but not limited to the disclaimers and warranties expressed in them.

Document version history

Release	Date	Description
1	1 Jun 2026	Initial release

Scope of document

This document applies to the following Raspberry Pi products:

Single Board Computers / SBCs

Pi Zero			Pi Zero 2		Pi 1				Pi 2	Pi 3			Pi 4	Pi 5
-	W	H	W	WH	A	B	A+	B+	B	A+	B	B+	B	-
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Compute Modules

CM0	CM1	CM3	CM3+	CM4	CM4S	CM5
✓	✓	✓	✓	✓	✓	✓

Keyboard Computers

Pi 400	Pi 500	Pi 500+
✓	✓	✓

Introduction

The time taken for a device to go from being powered on to being usable is a critical metric in embedded product deployments. Whether the goal is faster recovery after a power cycle, a more responsive user boot time is rarely a single-lever problem. Every stage of the boot sequence - from BootROM to the first userspace service - contributes to the total, and each stage has its own optimisation surface.

This document covers boot time optimisation for Raspberry Pi devices. The primary focus is on Raspberry Pi specific components, eg, Bootloader EEPROM configuration, firmware configuration, and Raspberry Pi hardware behaviour. General Linux topics such as kernel tuning, initramfs, filesystems, and user level services are covered where they have a meaningful impact on a typical Raspberry Pi deployment.

Context

This document covers wall-clock time from the application of power to a defined point in Linux userspace, typically the point at which the device is considered *operationally ready*. This is can be a target systemd unit being reached, a service accepting connections, or a user session being available.

The following stages are addressed:

- Bootloader EEPROM - boot device selection and configuration
- Boot firmware and config.txt - hardware initialisation and firmware controlled feature selection
- Linux kernel - command line, initramfs, early init
- Filesystems - choice, mount options, fsck behaviour
- Cryptographic storage - LUKS overhead, algorithmic hardware support
- systemd - unit ordering, parallelism, and service configuration

The document primarily targets Raspberry Pi generations 4 and 5. Legacy devices, and others such as Raspberry Pi Zero 2 W, share the same general Linux optimisation surface, but have different bootloader paths and more constrained hardware. Relevant differences are noted where they apply.

Application start up time - the latency from a userspace service being started to it being ready to handle requests - is outside scope, as is display pipeline and desktop environment initialisation.

Measurement Target

“Boot time” is not a solitary number. The Raspberry Pi boot sequence has a series of distinct phases - EEPROM bootloader, GPU firmware, kernel decompression and initialisation, initramfs, root filesystem mount, and early systemd - each having their own elapsed time and optimisation scope. Treating them all cumulatively as a single aggregate figure obscures where time is actually being spent.

The useful definition of *operational readiness* depends on context, but for the purposes of this document it is the point at which the Linux kernel has been handed control and early userspace initialisation is underway. Stages beyond that are built on top of what is covered here, and are outside scope. Within that boundary, it is important to establish the progress of individual stages rather than relying on the total. For example, a firmware change that saves 200ms is invisible if measuring ‘time to SSH’. Tools covered in later sections give per-stage visibility, but selecting the correct measurement point is as important as the measurement itself.

Perceived boot time - the elapsed time a user or operator experiences waiting for the device - can be influenced independently of actual ‘boot time’ through progress indication or splash screen, etc. This is a common product technique but it is not a substitute for reducing real latency, and does not help with automated restart or recovery scenarios where there is no human observation.

Measuring Boot Time

Effective optimisation requires measurement of each boot phase.

Hardware Trigger

An oscilloscope or logic analyser provides 'hard-truth' timing characterisation and measurement, independent of timing information reported by the software stack. Typical measurement techniques can utilise a IO line or LED driven by software at the point(s) of interest. For example, to measure EEPROM duration and the interval between power application and the first 'software-visible' event.

Bootloader / EEPROM

Setting `BOOT_UART=1` in the EEPROM configuration or config.txt enables UART output from the bootloader with per-stage timestamps. This covers boot device enumeration, storage initialisation, and firmware load - stages that predate kernel logging entirely. See https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#eeprom-boot-flow

Firmware

`sudo vclog --msg` outputs a full timestamped firmware message log, including the interval from EEPROM hand-off (gen4) or BootROM (gen5) and gives visibility of firmware execution progress. This makes pathological behaviour such as unexpectedly long hardware detection or failed peripheral probing immediately observable.

Linux

`printk.time=1` prefixes every `dmesg` line with a timestamp relative to kernel start, giving a chronological view of kernel initialisation. This parameter is not enabled by default in the upstream Linux kernel source code. Its default state is determined by a compile-time configuration option (`CONFIG_PRINTK_TIME`), but many distributions (including Raspberry Pi OS) choose to enable it in their default builds. If not on by default, it can be enabled by adding it to the kernel command line via `cmdline.txt`.

`/proc/uptime` gives elapsed time since kernel start and is useful as a lightweight marker in scripts.

The Linux kernel itself provides extensive tooling and infrastructure support that can capture execution flow, logic, and state without depending on timestamps. While most tracers include timestamps for context, these tools focus on event-based logic and dynamic instrumentation. For example, the `function_graph` tracer (ftrace) supports generation of a nested call-stack showing order, depth and sequence of events and latency triggers like `irqsoff` can help measure execution flow when interrupts are disabled. These may be useful when investigating causes of a stall, which may be relevant when trying to pinpoint latencies observed in achieving the point of operational readiness in userspace. The Appendix contains some examples of utilising these tracers.

Userspace (systemd)

`systemd-analyze` and its subcommands provide userspace-stage visibility and are very useful for performing dependency chain analysis (critical-chain), service ordering, socket activation, masking unused units, etc. systemd starts all units in parallel subject to ordering constraints, with the goal being to minimise the depth of the critical chain rather than the total number of units. Units waiting on `systemd-udev-settle.service` are a common hidden bottleneck - this waits for all udev events to quiesce and can add several seconds on hardware with many devices. On older or more constrained hardware the impact is more severe - settle times of around 20 seconds have been observed on the original Model B. The tuning of journald can also impact startup time on busy systems with slow storage IO performance, where flushing / trimming of the journal are a prerequisite for later stages. Similarly, over the lifetime of the device, some storage media can degrade in performance which can impact persistent journaling and therefore 'perceived' system performance.

Raspberry Pi Tooling

Two bespoke packages can further help analyse the progress through userland.

- `rpi-analyse-boot` provides in-depth post boot analysis. After boot, `/run/rpi-analyse-boot.service/index.html` is created which gives system information plus graphs and critical path boot information. There is also a menu item installed to launch this in a browser if running a headed system and a UI is installed.
- `rpi-trace-boot` initiates a Perfetto (https://perfetto.dev/) capture that encompasses boot, allowing introspection and analysis of where individual processes spent their time. Perfetto is an open-source system profiling, application tracing, and performance analysis suite. Tracing the boot this way does incur a slight overhead, but the information is highly valuable and Perfetto is a powerful tool.

Optimisation

The following sections provide concrete recommendations for reducing the cumulative time it takes for the device to boot.

EEPROM / Bootloader Configuration

The EEPROM bootloader configuration is read from a structured text file and applied with `rpi-eeprom-config --apply <file>`. Changes take effect on the next boot. See https://www.raspberrypi.com/documentation/computers/configuration.html#manage-bootloader-config

Boot Order

`BOOT_ORDER` dictates (i) which boot devices are attempted and (ii) in what order. Use this to select the intended target boot device. See https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#BOOT_ORDER

Attempting boot modes that will not succeed, such as network boot on a device with no TFTP server, USB-MSD when no device is attached, adds measurable latency before the bootloader advances to the next entry.

Boot UART

`BOOT_UART=1` enables UART output from the bootloader with per-stage timestamps. This is useful during development, but should be disabled for production. Disable the UART output to eliminate the overhead from the serial port IO.

```
BOOT_UART=0
```

Network Install

Disabling `NET_INSTALL_ENABLED` prevents the bootloader from checking for a network install image. This defaults to `0` on CM5 but should be set explicitly on device deployments where network install is not required. Disabling `NET_INSTALL_AT_POWER_ON` stops the device briefly displaying the network install UI after a cold boot.

```
NET_INSTALL_ENABLED=0
NET_INSTALL_AT_POWER_ON=0
```

See https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#NET_INSTALL_ENABLED

Power Delivery

On devices such as Raspberry Pi 5, the bootloader performs USB Power Delivery negotiation with the power supply before proceeding. If the PSU does not support USB PD, this negotiation times out before the boot sequence continues. Setting `PSU_MAX_CURRENT` instructs the bootloader to skip PD negotiation and assume the supply can provide the specified current. Set to `3000` for a standard 3 A supply, or `5000` for high-current:

```
PSU_MAX_CURRENT=5000
```

This is one of the larger single gains available at the EEPROM stage on Pi 5 when a non-PD power supply is used. See https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#PSU_MAX_CURRENT

Firmware Configuration

`config.txt` is read from the boot partition before handing control to the kernel. Several default actions involve hardware probing that are unnecessary on headless or fixed-hardware deployments. See https://www.raspberrypi.com/documentation/computers/config_txt.html for all supported options.

Hardware Auto-Detection

By default, connected cameras and associated display hardware are probed via slow buses like I2C, and the appropriate device tree overlays are loaded automatically. On headless devices, disable both:

```
camera_auto_detect=0
display_auto_detect=0
```

Display

For headless deployments, disabling the framebuffer eliminates the firmware's HDMI I2C and EDID probe entirely:

```
max_framebuffers=0
```

On Raspberry Pi 4, the firmware also parses HDMI EDID data and passes the negotiated video mode to the kernel via the command line. `disable_fw_kms_setup=1` skips this, deferring mode selection to the kernel's KMS driver. On Raspberry Pi 5 this is the default behaviour. For headless Pi 4 deployments, setting both is recommended:

```
max_framebuffers=0
disable_fw_kms_setup=1
```

Note that setting `max_framebuffers=0` disables bootloader framebuffer support (simple fb in the kernel), which means waiting for DRM driver init before any OSD can be rendered. While this can reduce execution pre-Linux, it completely disables the ability for the device to present an early splash screen as part of Linux boot which can help with visually indicating boot progress.

PCIe

gen5 devices (Pi5/CM5) are not certified for PCIe Gen 3.0 rates, but 3.0 links may be stable. If booting from NVMe, boot-time latency may be reduced by enabling 3.0 link speeds when Linux boots via `config.txt`.

```
dtparam=pciex1_gen=3
```

See https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#pcie-gen-3-0

Peripheral IO

Disable the probe of HAT EEPROM if no HAT present:

```
force_eeprom_read=0
```

If not using a PoE HAT, disable its probe:

```
disable_poe_fan=1
```

Linux

`CONFIG_KALLSYMS=n` removes the symbol table from the kernel image and reduces the overall size of the image read from boot storage. The trade-off is losing symbol resolution in oops/stack traces. If possible, trimming the kernel configuration and paying attention to built-in features vs modules can help reduce load time and memory footprint during the initial stages of execution.

The overhead of serial port output can be reduced significantly if `quiet` is added to the kernel command line, via `cmdline.txt`. Alternatively, using `log_level` may be more suitable. `quiet` sets `log_level=4` (`KERN_WARNING`). Level 3 may be a good compromise to only display critical errors and crashes.

Initramfs

Reducing initramfs size, stripping unnecessary modules and tools, compression algorithm choice, or eliminating initramfs entirely if/where possible, is the ultimate way to improve this stage. Many systems rely on initramfs for early userspace preparation or to prime a 'safe' start-up, eg custom udev rules installing specific device nodes / aliases, filesystem recovery, fallback strategies

etc, so often it is not possible to eliminate initramfs entirely. That being the case, ensuring it can be loaded, decompressed and executed as quickly as possible is the goal.

The most appropriate compression choice depends on storage throughput vs CPU speed trade-off:

- lz4 - fast decompression, good for larger images, often best when storage is fast (eMMC, NVMe)
- zstd - good compression ratio with very reasonable decompression speed, generally the best all-round choice
- xz / lzma - often generates the smallest image, slowest decompression, usually the best when storage is severely constrained
- Set in /etc/initramfs-tools/initramfs.conf via `COMPRESS=zstd` or equivalent
- Critical to benchmark on target hardware, eg flushing caches then, e.g. time zcat initrd.img > /dev/null vs time unlz4 initrd.img > /dev/null to compare

Further Considerations

BootROM Timings

The immutable BootROM has inherent latencies of its own, as does the run-time calibration of the DRAM interface performed by early microcode. The time taken for these early stages to complete execution is outside the scope of what can be improved upon. Typically, the first timestamped print from `vclog` indicates the wall-clock time the bootloader commenced execution. EEPROM versions as of Feb 2026 include early logging information by default in the output of this command.

```
pi@cm5:~$ sudo vclog --msg
004537.734: PCI2 init
004537.740: PCI2 reset
004537.748: PCIe scan 00001de4:00000001
004537.756: RP1_CHIP_INFO 20001927
...
006769.389: Starting OS 6769 ms
```

A gen5 platform such as CM5 can currently boot the OS in approximately 7 seconds with default settings. While the options above can reduce this time further, large gains (measured in seconds) are not achievable. SBC platforms such as Pi5 enable additional boot features by default like network install. These increase boot time. The options above can help reduce this aggresively, yielding a significant reduction. Older SBC platforms like Pi3B+ still achieve a respectable OS boot time - see the Appendix.

Clocks

Up-to-date early boot code runs the GPU and ARM cluster at max safe clocks rates prior to Linux boot. There is no user intervention that can be made to change these speeds. After Linux boots, the CPU scaling governor can be set early to increase performance (and power consumption) and stable overclocking may be possible. See https://www.raspberrypi.com/documentation/computers/config_txt.html#overclocking-options and https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#frequency-management-and-thermal-control for further details.

RP1

Part of the Bootloader execution flow on gen5 devices involves probing RP1 and loading firmware. RP1 is critical to peripheral access enablement, but it's initialisation is largely a blocking operation.

Storage Media

If booting from non-volatile storage, the faster data can be read, the quicker the boot progress can advance to the next phase and the device can reach operational readiness.

- eMMC can operate at HS200 (200 MB/s) or HS400 (400 MB/s) with a suitable kernel driver and device capability
- SD cards: A1/A2 Application Performance Class rating specifies minimum random IOPS (A1: 1500/500 read/write IOPS, A2: 4000/2000)
- NVMe on CM4/CM5 via PCIe is significantly faster than eMMC and is worth considering for latency-sensitive boot paths

The Bootloader does not support the same fully-featured MMC host link-training algorithm and high-speed rates that Linux does because it's required to balance read performance with reliability. For example, on CM5 with eMMC, it first tries HS-SDR 8-bit (104 MB/s effective) falling back to lower rates if link-training fails. On the same hardware, Linux establishes a higher bus speed[1]:

```
[    2.870365] mmc0: CQHCI version 5.10
[    2.874151] mmc1: CQHCI version 5.10
[    2.911165] mmc0: SDHCI controller on 1000fff000.mmc [1000fff000.mmc] using ADMA 64-bit
[    2.994808] mmc0: Command Queue Engine enabled, 16 tags
[    2.999670] mmc0: new HS400 Enhanced strobe MMC card at address 0001
[    3.005033] mmcblk0: mmc0:0001 8GTF4R 7.28 GiB
```

On older generation devices, HS-SDR 4-bit (52MB/s effective) is attempted first before fallback. Interrogation of an SD card's capabilities further determines how the Bootloader attempts access.

> **Note**
>
> (1) Storage vendor differences being what they are, the established bus speed is an interface ceiling, not a guarantee of sustained device throughput.

Partition alignment

- Tuning alignment to the boot storage device erase block size ensures partition start offsets are not misaligned. Misalignment to this boundary can cause read-modify-write cycles on every write to the first and last erase block
- Verify with `parted -l`. For a device with 4MiB erase block size, partitions should start on 4 MiB boundaries
- Misalignment has negligible impact on read performance, but can affect write latency during boot if anything writes to misaligned partitions

Time Synchronisation

Waiting on NTP sync can often be a significant cause of delay in reaching operational readiness, particularly in systems that are required to validate security certificate expiry attributes. This can certainly be the case in kiosk browser deployments where the browser may block on being usable until the system has acquired time sync. There's often no large gains that can be made in reducing the time it takes to achieve time sync, but some best practices include:

- Increase the polling rate (ie, decreasing the interval between requests) can improve synchronisation metrics
- Use multiple, geo-located Stratum 1 NTP servers to minimise latency
- Alternative clients like `chrony` may be quicker to synchronising the system clock

Care should be taken to avoid unnecessarily gating execution of systemd services on time sync being achieved. For example, an application responsible for opening a socket connection to a control-channel data end-point need not wait for a browser service to start.

Filesystem

Filesystem type, mkfs args, mount options, fsck overhead, and partition alignment all affect IO time, which directly relates to latency and therefore operational readiness.

- ext4 is the safe default for Read-Write storage. Creating with `lazy_itable_init=0,lazy_journal_init=0` avoids on-device background inode table initialisation running during first boot and interfering with timing measurements.
- Read-only filesystems such as squashfs or EROFS eliminate journaling entirely, therefore removing any possibility of journal replay following an unclean shutdown. They also reduce filesystem size significantly and typically exceed the raw IO bandwidth of the underlying storage media via 'compression amplification' (ie, reading a small amount of compressed data from storage and expanding via decompression results in less physical IO per logical byte). Compression Amplification can yield significantly quicker system start-up times on devices with high performing CPUs and low latency IO paths, e.g. EROFS with a 16K block size on CM5 running Linux 6.18[1] can achieve greater than 3x the read performance in `rand64k-q4` fio tests compared to ext4[2].
- Utilising a filesystem block size equivalent to that of the target kernel's page size ensures optimum IO performance. On Pi5/CM5, a 16K filesystem block size has significantly lower latencies for read and write operations across different filesystems compared to 4K on the same hardware. The trade-off is an increase in overall filesystem size (and therefore usable capacity) and compatibility across other devices (LFS support not withstanding)[3].

Note

(1) The 2712 kernel has a 16K page size

Note

(2) Benchmark conducted on CM5 with Linux 6.18 using a 16K EROFS image. Results will vary by workload and storage device.

Warning

(3) While a 16K page size kernel may be beneficial, it does carry compatibility risks. Filesystems with a block size greater than the host system's page size may not be able to be mounted on that host.

Ext4

- `noatime` - eliminates access time writes on every file read, high impact on read-heavy boot workloads
- `data=writeback` - fastest journal mode, metadata is journaled but data writes are not ordered with respect to metadata
- `data=ordered` (default) - safer, but adds ordering overhead
- `commit=60` - extends the default filesystem journal commit interval from 5s to 60s, reduces IO commit frequency at the cost of a larger replay window after power loss
- Avoid `discard` (inline TRIM) on eMMC - it adds per-write latency. Often better to use `fstrim.timer` for periodic offline TRIM instead
- `barrier=0` - disables write barriers, improves throughput on devices without a write cache, but unsafe on power loss without other mitigations

fsck

- With a read-only rootfs, fsck is unnecessary and can be skipped. Authentication and verification of RO filesystems (eg via dm-verity) can ensure integrity.
- For writeable ext4, `tune2fs -c 0 -i 0` disables mount-count and time-based periodic fsck checks, although avoiding these health checks is not recommended.
- Journal replay on ext4 after an unclean shutdown can add several seconds, result in data loss and/or potential reconstruction of critial filesystem contents at boot. A read-only filesystem can eliminate this entirely.

Crypto

Signed Boot

Utilising signed boot will increase the time taken to reach booting the OS, but the increase is directly dependent on the amount of data undergoing authenticity checks.

1. `BootROM` -> `bootsys`: The SoC BootROM (immutable, fused into silicon) loads the second-stage bootloader (`bootsys`) from EEPROM (SPI flash) and verifies its signature.
2. `bootsys` -> bootmain: `bootsys` loads the customer's public key from EEPROM and checks it against the customer key hash stored in OTP. It then initialises SDRAM, verifies dependencies against build-time hashes, and loads `bootmain`.
3. `bootmain` -> `boot.img`: `bootmain` loads boot.img (a single, self-contained FAT formatted ramdisk containing all critical boot assets, eg linux kernel image, DTBs, config.txt, GPU firmware, etc) and `boot.sig` from the boot media. It verifies that the SHA-256 hash of boot.img matches `boot.sig`, and that the RSA signature in `boot.sig` can be validated using the customer's public key.
4. boot.img -> OS: `bootmain` uses the verified ramdisk as the source of GPU firmware (`start.elf`) for devices that require it, and starts the OS. With secure boot enabled and the verified boot.img loaded in memory, only files from the verified ramdisk are loaded and executed.

The authenticity of `boot.img` is checked in one pass. The larger this file is, the longer the check will take. Reducing the size of it should be a priority in order to optimise boot time on systems where signed boot is required.

LUKS

If utilising LUKS for filesystem encryption, cryptographic performance is heavily dependant on hardware capability. Choosing an optimal cipher spec starts with benchmarking the target hardware.

A less visible but significant contributor to unlock time is the Password-Based Key Derivation Function (PBKDF). During `cryptsetup luksFormat`, the PBKDF parameters are benchmarked on the target device and chosen so that key derivation deliberately takes approximately 2000ms - an intentional security measure to make brute-force attacks expensive. This cost is paid unconditionally on every unlock. The `--iter-time` argument controls the target duration in milliseconds, but reducing it weakens the key derivation security margin. It is worth accounting for this fixed overhead when budgeting total LUKS unlock time, particularly on devices where the 2000ms represents a meaningful percentage of the overall boot sequence.

The following comparison used the same SD card boot media, kernel and root filesystem.

```
pi@pi5:~$ cryptsetup benchmark --cipher aes-xts-plain64
# Tests are approximate using memory only (no storage IO).
# Algorithm |       Key |      Encryption |      Decryption
    aes-xts        256b      1598.2 MiB/s      1606.9 MiB/s
pi@pi5:~$ cryptsetup benchmark --cipher xchacha12,aes-adiantum-plain64
# Tests are approximate using memory only (no storage IO).
#            Algorithm |       Key |      Encryption |      Decryption
xchacha12,aes-adiantum        256b       605.1 MiB/s       625.2 MiB/s
```

```
pi@pi4:~$ cryptsetup benchmark --cipher aes-xts-plain64
# Tests are approximate using memory only (no storage IO).
# Algorithm |       Key |      Encryption |      Decryption
    aes-xts        256b        79.5 MiB/s       110.0 MiB/s
pi@pi4:~$ cryptsetup benchmark --cipher xchacha12,aes-adiantum-plain64
# Tests are approximate using memory only (no storage IO).
#            Algorithm |       Key |      Encryption |      Decryption
xchacha12,aes-adiantum        256b       247.5 MiB/s       261.2 MiB/s
```

Memory Pressure and Swap

Attempting to do too much during early boot on memory-constrained devices can trigger swapping to storage. On slow media such as SD cards this creates head-of-line blocking: the kernel cannot load file-backed pages needed to continue booting because the

storage pipeline is occupied writing swap pages out. The effect compounds quickly. For example, under these conditions `openat` syscalls have been observed to take 2 seconds and tiny (33-byte) reads take 2.44 seconds on Raspberry Pi Zero 2 W. Avoiding swap entirely on boot-critical paths, or using zram (which compresses pages in RAM rather than writing to storage), eliminates this type of problem. Careful review of which services and daemons are started during early boot is advisable on memory-constrained hardware.

Appendix

Linux tracing

The examples below generate output (`/sys/kernel/tracing/trace`) that can be loaded directly into Perfetto (ui.perfetto.dev). The UI parses the human-readable text format in the trace file and supports features such as search, zoom, and interactive timeline navigation. Perfetto's `trace_processor` CLI command can also be used:

```
$ trace_processor --httpd /path/to/trace
```

See https://perfetto.dev/docs/getting-started/ftrace and https://perfetto.dev/docs/visualization/large-traces for further details.

ftrace

This example uses the function_graph tracer to show a flow of exactly which kernel functions called others during a specific command (`ls`).

```
$ cd /sys/kernel/tracing/
$ echo 0 | sudo tee tracing_on
$ echo function_graph | sudo tee current_tracer
$ echo | sudo tee trace
$ echo vfs_read | sudo tee set_graph_function
$ echo 1 | sudo tee tracing_on
$ ls /
$ echo 0 | sudo tee tracing_on
$ $ sudo cat trace | head -n 20
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 1)               |  tty_ldisc_ref_wait() {
 0)   2.500 us    |  mutex_unlock();
 1)   0.574 us    |    ldsem_down_read();
 1)   3.759 us    |  }
 0)               |  vfs_read() {
 0)               |    rw_verify_area() {
 0)   0.352 us    |      security_file_permission();
 0)   1.019 us    |    }
 0)               |    pipe_read() {
 0)   0.315 us    |      mutex_lock();
 0)   0.351 us    |      mutex_unlock();
 0)   0.241 us    |      kill_fasync();
 0)   2.019 us    |    }
 0)   3.982 us    |  }
 1)               |  vfs_read() {
 1)               |    rw_verify_area() {
```

irqsoff

The irqsoff tracer captures the longest period that interrupts were disabled. It records the specific "start" and "stop" functions responsible for the delay.

```
$ cd /sys/kernel/tracing/
$ echo 0 | sudo tee tracing_on
$ echo irqsoff | sudo tee current_tracer
$ echo | sudo tee trace
$ echo 1 | sudo tee tracing_on
<fio workload>
$ echo 0 | sudo tee tracing_on
$ sudo cat trace
```

```
# tracer: irqsoff
#
# irqsoff latency trace v1.1.5 on 6.12.17
# --------------------------------------------------------------------
# latency: 1267 us, #1790/1790, CPU#3 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:4)
#    -----------------
#    | task: fio-1091 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#  => started at: _raw_spin_lock_irqsave
#  => ended at:   free_pcppages_bulk
#
#
#                    _------=> CPU#
#                   / _-----=> irqs-off/BH-disabled
#                  | / _----=> need-resched
#                  || / _---=> hardirq/softirq
#                  ||| / _--=> preempt-depth
#                  |||| / _-=> migrate-disable
#                  ||||| /     delay
#  cmd     pid     |||||| time  |   caller
#     \   /        ||||||  \    |    /
     fio-1091      3d..2.    0us : trace_hardirqs_off <-_raw_spin_lock_irqsave
     fio-1091      3d..2.    1us : preempt_count_add <-_raw_spin_lock_irqsave
     fio-1091      3d..3.    1us : get_pfnblock_flags_mask <-free_pcppages_bulk
     fio-1091      3d..3.    2us : __mod_zone_page_state <-__free_one_page
     fio-1091      3d..3.    3us : get_pfnblock_flags_mask <-free_pcppages_bulk
     fio-1091      3d..3.    3us : __mod_zone_page_state <-__free_one_page
...
     fio-1091      3d..3. 1265us : get_pfnblock_flags_mask <-free_pcppages_bulk
     fio-1091      3d..3. 1266us : __mod_zone_page_state <-__free_one_page
     fio-1091      3d..3. 1266us : _raw_spin_unlock_irqrestore <-free_pcppages_bulk
     fio-1091      3d..3. 1267us : _raw_spin_unlock_irqrestore <-free_pcppages_bulk
     fio-1091      3d..3. 1268us : tracer_hardirqs_on <-free_pcppages_bulk
     fio-1091      3d..3. 1272us : <stack trace>
 => free_pcppages_bulk
 => free_unref_page_commit
 => free_unref_folios
 => folios_put_refs
 => free_pages_and_swap_cache
 => __tlb_batch_free_encoded_pages
 => tlb_finish_mmu
 => exit_mmap
 => __mmput
 => mmput
 => do_exit
 => do_group_exit
 => __arm64_sys_exit_group
 => invoke_syscall
 => el0_svc_common.constprop.0
 => do_el0_svc
 => el0_svc
 => el0t_64_sync_handler
 => el0t_64_sync
```

In the above example, the trace reveals a 1.2ms latency spike on CPU 3 caused by a call to `free_pcppages_bulk()` during cleanup, where hard interrupts were disabled while freeing a large batch of pages.

Combining Linux Userspace and Kernel Events

The infrastructure underpinning the tracers in previous sections also provides a mechanism for userspace code to write events directly into the same kernel ring buffer. The result is a single, unified timeline of userspace activity interleaved with kernel events, eg scheduling decisions, interrupt handlers, memory reclaim operations - all on a shared timestamp axis. The diagnostic value of

this lies in being able to correlate userspace and kernel events together, using the same source. A stall observed in a userspace process can be examined alongside the kernel activity that coincided with it.

Whether the cause is a scheduling preemption, lock contention, a memory compaction event, or an interrupt storm, the answer is in a single trace without the need to correlate separate data sources after the fact. This approach has proven consistently useful when diagnosing latencies that manifest in userspace but are driven by kernel behaviour.

Two interfaces are available.

trace_marker

`trace_marker` is the simpler of the two. Writing any string to `/sys/kernel/tracing/trace_marker` inserts it into the ftrace ring buffer at the current timestamp, appearing in the output alongside kernel trace events. No registration is required.

With suitable structure to the inserted trace_marker string, it will appear in Perfetto as a graph. For example, custom userspace events can be displayed in Perfetto by writing specifically formatted strings, such as `B|<pid>|<name>` and `E|<pid>`, to `/sys/kernel/tracing/trace_marker`. Perfetto parses these ftrace/print events, mapping them into slices, counters, or instant events when ftrace/print is enabled in the trace configuration.

The following example instruments a boot-time service to bracket its restart, with scheduler events enabled to reveal any preemptions or wakeup delays that occurred during that interval:

```
$ cd /sys/kernel/tracing
$ echo 0 | sudo tee tracing_on
$ echo | sudo tee trace
$ echo 1 | sudo tee events/sched/sched_switch/enable
$ echo 1 | sudo tee events/sched/sched_wakeup/enable
$ echo 1 | sudo tee tracing_on
$ echo "networking: restart" > /sys/kernel/tracing/trace_marker
$ systemctl restart systemd-networkd
$ echo "networking: restart done" > /sys/kernel/tracing/trace_marker
$ echo 0 | sudo tee tracing_on
```

Examining the trace buffer:

```
$ cat trace
# tracer: nop
#
# entries-in-buffer/entries-written: 13481/13481   #P:4
#
#                                _-----=> irqs-off/BH-disabled
#                               / _-----=> need-resched
#                              | / _----=> hardirq/softirq
#                              || / _---=> preempt-depth
#                              ||| / _--=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
          <idle>-0       [002] d..2.    60.908817: sched_switch: prev_comm=swapper/2 prev_pid=0 prev_prio=120
          prev_state=R ==> next_comm=kworker/u16:0 next_pid=11 next_prio=120
...
            bash-443     [002] .....    99.819834: tracing_mark_write: networking: restart
          <idle>-0       [001] dNh2.    99.820125: sched_wakeup: comm=kworker/u16:0 pid=11 prio=120 target_cpu=001
          <idle>-0       [001] d..2.    99.820128: sched_switch: prev_comm=swapper/1 prev_pid=0 prev_prio=120
          prev_state=R ==> next_comm=kworker/u16:0 next_pid=11 next_prio=120
   kworker/u16:0-11      [001] d..2.    99.820140: sched_switch: prev_comm=kworker/u16:0 prev_pid=11 prev_prio=120
   prev_state=I ==> next_comm=swapper/1 next_pid=0 next_prio=120
...
     kcompactd0-44       [001] d..2.    99.899969: sched_switch: prev_comm=kcompactd0 prev_pid=44 prev_prio=120
     prev_state=S ==> next_comm=swapper/1 next_pid=0 next_prio=120
          <idle>-0       [003] dNh4.    99.986409: sched_wakeup: comm=rpi-connectd pid=398 prio=120 target_cpu=003
          <idle>-0       [003] d..2.    99.986415: sched_switch: prev_comm=swapper/3 prev_pid=0 prev_prio=120
          prev_state=R ==> next_comm=rpi-connectd next_pid=398 next_prio=120
    rpi-connectd-398     [003] d..2.    99.986450: sched_switch: prev_comm=rpi-connectd prev_pid=398 prev_prio=120
    prev_state=S ==> next_comm=swapper/3 next_pid=0 next_prio=120
          <idle>-0       [003] dNh4.    99.986942: sched_wakeup: comm=rpi-connectd pid=401 prio=120 target_cpu=003
```

```
          <idle>-0       [003] d..2.    99.986947: sched_switch: prev_comm=swapper/3 prev_pid=0 prev_prio=120
          prev_state=R ==> next_comm=rpi-connectd next_pid=401 next_prio=120
...
            bash-443     [001] .....   112.796388: tracing_mark_write: networking: restart done
```

The trace shows captures 12.98 second window during which bash (PID 443) logs the start and completion of a network service restart. Between these markers, the Linux scheduler manages idle states, wakes up kernel worker threads, and switches context between background tasks like memory compaction (kcompactd0) and Raspberry Pi connectivity services (rpi-connectd).

For example, using this method, it's possible to establish if a process has been involuntarily preempted:

```
$ echo 0 > tracing_on
$ echo > trace
$ echo 1 > events/sched/sched_switch/enable
$ echo 1 > events/sched/sched_wakeup/enable
$ echo 1 > options/context-info
$ echo dd > set_ftrace_filter 2>/dev/null
$ # Turn on the tracer immediately before running the test
$ echo 1 > /sys/kernel/tracing/tracing_on;taskset -c 1 dd if=/dev/urandom of=/scratch.img bs=4M count=500
conv=fdatasync;echo 0 > /sys/kernel/tracing/tracing_on
```

Examining the trace buffer:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 71522/147657   #P:4
#
#                                _-----=> irqs-off/BH-disabled
#                               / _-----=> need-resched
#                              | / _----=> hardirq/softirq
#                              || / _---=> preempt-depth
#                              ||| / _--=> migrate-disable
#                              |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
            bash-443     [001] d..2.  1885.644008: sched_switch: prev_comm=bash prev_pid=443 prev_prio=120
            prev_state=S ==> next_comm=swapper/1 next_pid=0 next_prio=120
          <idle>-0       [003] dNh2.  1885.647171: sched_wakeup: comm=rcu_preempt pid=17 prio=120 target_cpu=003
          <idle>-0       [003] d..2.  1885.647176: sched_switch: prev_comm=swapper/3 prev_pid=0 prev_prio=120
          prev_state=R ==> next_comm=rcu_preempt next_pid=17 next_prio=120
     rcu_preempt-17      [003] d..2.  1885.647195: sched_switch: prev_comm=rcu_preempt prev_pid=17 prev_prio=120
     prev_state=I ==> next_comm=swapper/3 next_pid=0 next_prio=120
          <idle>-0       [001] d..2.  1885.648555: sched_switch: prev_comm=swapper/1 prev_pid=0 prev_prio=120
          prev_state=R ==> next_comm=taskset next_pid=606 next_prio=120
              dd-606     [001] d..2.  1885.648959: sched_switch: prev_comm=taskset prev_pid=606 prev_prio=120
              prev_state=D ==> next_comm=swapper/1 next_pid=0 next_prio=120
          <idle>-0       [001] dNh2.  1885.649281: sched_wakeup: comm=taskset pid=606 prio=120 target_cpu=001
          <idle>-0       [001] d..2.  1885.649285: sched_switch: prev_comm=swapper/1 prev_pid=0 prev_prio=120
          prev_state=R ==> next_comm=taskset next_pid=606 next_prio=120
              dd-606     [001] d..2.  1885.650021: sched_switch: prev_comm=dd prev_pid=606 prev_prio=120
              prev_state=D ==> next_comm=swapper/1 next_pid=0 next_prio=120
...
     rcu_preempt-17      [003] d..2.  1885.997170: sched_switch: prev_comm=rcu_preempt prev_pid=17 prev_prio=120
     prev_state=I ==> next_comm=swapper/3 next_pid=0 next_prio=120
              dd-606     [001] d..2.  1886.383818: sched_switch: prev_comm=dd prev_pid=606 prev_prio=120
              prev_state=R+ ==> next_comm=kworker/1:3 next_pid=236 next_prio=120
     kworker/1:3-236     [001] d..2.  1886.383841: sched_switch: prev_comm=kworker/1:3 prev_pid=236 prev_prio=120
     prev_state=I ==> next_comm=dd next_pid=606 next_prio=120
```

The preemption signature is visible at timestamp 1886.383818. `prev_state=R+` means the process was in the `TASK_RUNNING` state - it was actively executing on the CPU and did not ask to sleep. The `+` (available in newer kernels) indicates the task was preempted while still runnable. A kernel thread (kworker/1:3, PID 236) targeted CPU 1 and forcibly displaced dd.

user_events

`user_events` (Linux 5.18+, CONFIG_USER_EVENTS=y) provides a more structured interface. Events are registered with the kernel via `/sys/kernel/tracing/user_events_data`, after which the application writes to a shared memory region. This removes the syscall

overhead of trace_marker for high-frequency instrumentation. Registered events appear in tracefs as first-class events, can be enabled and filtered via the standard event subsystem, and are visible to perf record and other consumers alongside kernel events.

For boot-time analysis, `trace_marker` is generally sufficient. `user_events` is more appropriate where the instrumented code path itself is performance-critical and the cost of a write(2) syscall per event would prevent accurate measurement. See https://www.kernel.org/doc/html/latest/trace/user_events.html for the full API.

Device Boot Times

The following show OS boot times for certain devices with up to date firmware and Raspberry Pi OS image (June 2026) using default settings.

Raspberry Pi 3B+

```
Raspberry Pi Bootcode

Read File: config.txt, 1298

Read File: start.elf, 3029888 (bytes)

Read File: fixup.dat, 7370 (bytes)

MESS:00:00:01.399082:0: boot-part: 0 fs-type: 0
MESS:00:00:01.401908:0: boot-part: 0 fs-type: 3
MESS:00:00:01.412464:0: brfs: File read: /mfs/sd/config.txt
MESS:00:00:01.417188:0: board: boardrev a020d4 otp a020d4
MESS:00:00:01.421581:0: brfs: File read: 1298 bytes
MESS:00:00:01.491993:0: brfs: File read: /mfs/sd/config.txt
MESS:00:00:02.018534:0: gpioman: gpioman_get_pin_num: pin DISPLAY_DSI_PORT not defined
MESS:00:00:02.025558:0: gpioman: gpioman_get_pin_num: pin DISPLAY_DSI_PORT not defined
MESS:00:00:02.034165:0: *** Restart logging
MESS:00:00:02.036651:0: brfs: File read: 1298 bytes
MESS:00:00:02.072593:0: gpioman: gpioman_get_pin_num: pin EMMC_ENABLE not defined
MESS:00:00:02.088790:0: HDMI0: hdmi_pixel_freq_limit: 162000000
MESS:00:00:02.842373:0: brfs: File read: /mfs/sd/initramfs8
MESS:00:00:02.846252:0: Loaded 'initramfs8' to 0x0 size 0xb2483c
MESS:00:00:02.865142:0: initramfs (initramfs8) loaded to 0x2e4db000 (size 0xb2483c)
MESS:00:00:02.880801:0: dtb_file 'bcm2710-rpi-3-b-plus.dtb'
MESS:00:00:02.884677:0: brfs: File read: 11683900 bytes
MESS:00:00:02.893635:0: brfs: File read: /mfs/sd/bcm2710-rpi-3-b-plus.dtb
MESS:00:00:02.898727:0: Loaded 'bcm2710-rpi-3-b-plus.dtb' to 0x100 size 0x89b4
MESS:00:00:02.921647:0: brfs: File read: 35252 bytes
MESS:00:00:02.933973:0: brfs: File read: /mfs/sd/overlays/overlay_map.dtb
MESS:00:00:02.974789:0: brfs: File read: 5971 bytes
MESS:00:00:02.980681:0: brfs: File read: /mfs/sd/config.txt
MESS:00:00:02.984693:0: dtparam: audio=on
MESS:00:00:02.995656:0: brfs: File read: 1298 bytes
MESS:00:00:03.011521:0: brfs: File read: /mfs/sd/overlays/vc4-kms-v3d.dtbo
MESS:00:00:03.064793:0: Loaded overlay 'vc4-kms-v3d'
MESS:00:00:03.169621:0: brfs: File read: 2760 bytes
MESS:00:00:03.178770:0: brfs: File read: /mfs/sd/overlays/disable-bt.dtbo
MESS:00:00:03.201086:0: Loaded overlay 'disable-bt'
MESS:00:00:03.247653:0: brfs: File read: 1073 bytes
MESS:00:00:03.252021:0: brfs: File read: /mfs/sd/cmdline.txt
MESS:00:00:03.256265:0: Read command line from file 'cmdline.txt':
MESS:00:00:03.262121:0: 'console=serial0,115200 console=tty1 root=PARTUUID=8b4815ac-02 rootfstype=ext4
fsck.repair=yes rootwait ds=nocloud;i=rpi-imager-1778840567672 cfg80211.ieee80211_regdom=GB'
MESS:00:00:03.470046:0: brfs: File read: 169 bytes
MESS:00:00:04.127223:0: brfs: File read: /mfs/sd/kernel8.img
MESS:00:00:04.131190:0: Loaded 'kernel8.img' to 0x200000 size 0x9af973
MESS:00:00:06.760289:0: Device tree loaded to 0x2e4d2000 (size 0x8f51)
MESS:00:00:06.767834:0: uart: Set PL011 baud rate to 103448.300000 Hz
```

```
MESS:00:00:06.774082:0: uart: Baud rate change done...
MESS:00:00:06.777497:0: uart: Baud rate change done...
MESS:00:00:06.784216:0: gpioman: gpioman_get_pin_num: pin SDCARD_CONTROL_POWER not defined
MESS:00:00:06.790801:0: Watchdog stopped
MESS:00:00:06.794412:0: arm_loader: Starting ARM with 948MB
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
```

Contact us for more information

Please contact applications@raspberrypi.com if you have any queries about this whitepaper.

Web: www.raspberrypi.com

Raspberry Pi

Raspberry Pi Ltd