Optimizing Boot Times

This document provides partner guidance for improving boot times for specific Android devices. Boot time is an important component of system performance as users must wait for boot to complete before they can use the device. For devices such as cars where cold boot-up happens more frequently, having a quick boot time is critical (no one likes waiting for dozens of seconds just to input a navigation destination).

Android 8.0 allows for reduced boot times by supporting several improvements across a range of components. The following table summarizes these performance improvements (as measured on a Google Pixel and Pixel XL devices).

Component	Improvement
Bootloader	• Saved 1.6s by removing UART log • Saved 0.4s by changing to LZ4 from GZIP
Device kernel	• Saved 0.3s by removing unused kernel configs and reducing driver size • Saved 0.3s with dm-verity prefetch optimization • Saved 0.15s to remove unnecessary wait/test in driver • Saved 0.12s to remove CONFIG_CC_OPTIMIZE_FOR_SIZE
I/O tuning	• Saved 2s on normal boot • Saved 25s on first boot
init.*.rc	• Saved 1.5s by paralleling init commands • Saved 0.25s by starting zygote early • Saved 0.22s by cpuset tune
Boot animation	• Started 2s earlier on boot without fsck triggered, much bigger on boot with fsck triggered boot • Saved 5s on Pixel XL with immediate shutdown of boot animation
SELinux policy	Saved 0.2s on by genfscon

Optimizing Bootloader

To optimize bootloader for improved boot times:

- For logging:
  - Disable log writing to UART as it can take a long time with lots of logging. (On the Google Pixel devices, we found it slows the bootloader 1.5s).
  - Log only error situations and consider storing other information to memory with a separate mechanism to retrieve.
- For kernel decompression, considering using LZ4 for contemporary hardware instead of GZIP (example patch (https://patchwork.kernel.org/patch/6810841/)). Keep in mind that different kernel compression options can have different loading and decompression times, and some options may work better than others for your specific hardware.
- Check unnecessary wait times for debouncing/special mode entry and minimize them.
- Pass boot time spent in bootloader to kernel as cmdline.
- Check CPU clock and consider parallelization (requires multi-core support) for kernel loading and initializing I/O.

Optimizing Kernel

Use the following tips to optimize the kernel for improved boot times.

Minimizing device defconfig

Minimizing kernel config can reduce kernel size for faster loading decompression, initialization and smaller attack surfaces. To optimize the device defconfig:

- **Identify unused drivers**. Review the `/dev` and `/sys` directories and look for nodes with general SELinux labels (which indicates those nodes are not configured to be accessible by user space). Remove any such nodes if found.

- **Unset unused CONFIGs**. Review the .config file generated by kernel build to explicitly unset any unused CONFIG that was turned on by default. For example, we removed the following unused CONFIGs from the Google Pixel:

```
CONFIG_ANDROID_LOGGER=y
CONFIG_IMX134=y
CONFIG_IMX132=y
CONFIG_OV9724=y
CONFIG_OV5648=y
CONFIG_GC0339=y
CONFIG_OV8825=y
CONFIG_OV8865=y
CONFIG_s5k4e1=y
CONFIG_OV12830=y
CONFIG_USB_EHCI_HCD=y
CONFIG_IOMMU_IO_PGTABLE_FAST_SELFTEST=y
CONFIG_IKCONFIG=y
CONFIG_RD_BZIP2=y
CONFIG_RD_LZMA=y
CONFIG_TI_DRV2667=y
CONFIG_CHR_DEV_SCH=y
CONFIG_MMC=y
CONFIG_MMC_PERF_PROFILING=y
CONFIG_MMC_CLKGATE=y
CONFIG_MMC_PARANOID_SD_INIT=y
CONFIG_MMC_BLOCK_MINORS=32
CONFIG_MMC_TEST=y
CONFIG_MMC_SDHCI=y
CONFIG_MMC_SDHCI_PLTFM=y
CONFIG_MMC_SDHCI_MSM=y
CONFIG_MMC_SDHCI_MSM_ICE=y
CONFIG_MMC_CQ_HCI=y
CONFIG_MSDOS_FS=y
# CONFIG_SYSFS_SYSCALL is not set
CONFIG_EEPROM_AT24=y
# CONFIG_INPUT_MOUSEDEV_PSAUX is not set
CONFIG_INPUT_HBTP_INPUT=y
# CONFIG_VGA_ARB is not set
CONFIG_USB_MON=y
CONFIG_USB_STORAGE_DATAFAB=y
CONFIG_USB_STORAGE_FREECOM=y
CONFIG_USB_STORAGE_ISD200=y
CONFIG_USB_STORAGE_USBAT=y
CONFIG_USB_STORAGE_SDDR09=y
CONFIG_USB_STORAGE_SDDR55=y
CONFIG_USB_STORAGE_JUMPSHOT=y
CONFIG_USB_STORAGE_ALAUDA=y
CONFIG_USB_STORAGE_KARMA=y
CONFIG_USB_STORAGE_CYPRESS_ATACB=y
CONFIG_SW_SYNC_USER=y
CONFIG_SEEMP_CORE=y
CONFIG_MSM_SMEM_LOGGING=y
CONFIG_IOMMU_DEBUG=y
CONFIG_IOMMU_DEBUG_TRACKING=y
CONFIG_IOMMU_TESTS=y
CONFIG_MOBICORE_DRIVER=y
# CONFIG_DEBUG_PREEMPT is not set
```

- **Remove CONFIGs that lead to unnecessary test runs on every boot**. While useful in development, such configs (i.e. CONFIG_IOMMU_IO_PGTABLE_FAST_SELFTEST) should be removed in a production kernel.

Minimizing driver size

Some drivers in the device kernel can be removed if the function is not used to reduce kernel size further. For example, if WLAN is connected through PCIe, the SDIO support is not used and should be removed during compile time. For details, refer to the Google Pixel kernel: net: wireless: cnss: add option to disable SDIO support.

Removing compiler optimization for size

Remove the kernel config for CONFIG_CC_OPTIMIZE_FOR_SIZE. This flag was originally introduced when the assumption was that smaller code size would yield hot cache hit (and thus be faster). However, this assumption is no longer valid as modern mobile SoCs have become more powerful.

In addition, removing the flag can enable the compiler warning for uninitialized variables, which is suppressed in Linux kernels when the CONFIG_CC_OPTIMIZE_FOR_SIZE flag is present (making this change alone has helped us uncover many meaningful bugs in some Android device drivers).

Deferring initialization

Many processes launch during boot, but only components in critical path (bootloader > kernel > init > file system mount > zygote > system server) directly affect boot time. Use the early kernel log to identify peripheral/components that are not critical to the start init process, then delay those peripherals/components until later in the boot process.

Optimizing I/O efficiency

Improving I/O efficiency is critical to making boot time faster, and reading anything not necessary should be deferred until after boot (on a Google Pixel, about 1.2GB of data is read on boot).

Tuning the filesystem

The Linux kernel read ahead kicks in when a file is read from beginning or when blocks are read sequentially, making it necessary to tune I/O scheduler parameters specifically for booting (which has a different workload characterization than normal applications).

Devices that support seamless (A/B) updates benefit greatly from filesystem tuning on first time boot (e.g. 20s on Google Pixel). An an example, we tuned the following parameters for the Google Pixel:

```
on late-fs
  # boot time fs tune
    # boot time fs tune
    write /sys/block/sda/queue/iostats 0
    write /sys/block/sda/queue/scheduler cfq
    write /sys/block/sda/queue/iosched/slice_idle 0
    write /sys/block/sda/queue/read_ahead_kb 2048
    write /sys/block/sda/queue/nr_requests 256
    write /sys/block/dm-0/queue/read_ahead_kb 2048
    write /sys/block/dm-1/queue/read_ahead_kb 2048

on property:sys.boot_completed=1
    # end boot time fs tune
    write /sys/block/sda/queue/read_ahead_kb 512
    ...
```

Miscellaneous

- Turn on the dm-verity hash prefetch size using kernel config DM_VERITY_HASH_PREFETCH_MIN_SIZE (default size is 128).
- For better file system stability and a dropped forced check that occurs on every boot, use the new ext4 generation tool by setting TARGET_USES_MKE2FS in BoardConfig.mk.

Analyzing I/O

To understand I/O activities during boot, use kernel ftrace data (also used by systrace):

```
trace_event=block,ext4 in BOARD_KERNEL_CMDLINE
```

To breakdown file access for each file, make the following changes to the kernel (development kernel only; do not use in production kernels):

```
diff --git a/fs/open.c b/fs/open.c
index 1651f35..a808093 100644
--- a/fs/open.c
+++ b/fs/open.c
@@ -981,6 +981,25 @@
 }
 EXPORT_SYMBOL(file_open_root);

+static void _trace_do_sys_open(struct file *filp, int flags, int mode, long fd)
+{
+       char *buf;
+       char *fname;
+
+       buf = kzalloc(PAGE_SIZE, GFP_KERNEL);
+       if (!buf)
+               return;
+       fname = d_path(&filp-<f_path, buf, PAGE_SIZE);
+
+       if (IS_ERR(fname))
+               goto out;
+
+       trace_printk("%s: open(\"%s\", %d, %d) fd = %ld, inode = %ld\n",
+                     current-<comm, fname, flags, mode, fd, filp-<f_inode-<i_ino);
+out:
+       kfree(buf);
+}
+
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
 {
        struct open_flags op;
@@ -1003,6 +1022,7 @@
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
+                       _trace_do_sys_open(f, flags, mode, fd);
```

Use the following scripts to help with analyzing boot performance.

- `packages/services/Car/tools/bootanalyze/bootanalyze.py` Measures boot time with a breakdown of important steps in the boot process.
- `packages/services/Car/tools/io_analysis/check_file_read.py boot_trace` Provides access information per each file.
- `packages/services/Car/tools/io_analysis/check_io_trace_all.py boot_trace` Gives system-level breakdown.

Optimizing init.*.rc

Init is the bridge from the kernel till the framework is established, and devices usually spend a few seconds in different init stages.

Running tasks in parallel

While the current Android init is more or less a single threaded process, you can still perform some tasks in parallel.

- Execute slow commands in a shell script service and join that later by waiting for specific property. Android 8.0 supports this use case with a new `wait_for_property` command.
- Identify slow operations in init. The system logs the init command exec/wait_for_prop or any action taking a long time (in Android 8.0, any command taking more than 50 ms). For example:

  ```
  init: Command 'wait_for_coldboot_done' action=wait_for_coldboot_done returned 0 took 585.012ms
  ```

  Reviewing this log may indicate opportunities for improvements.
- Start services and enable peripheral devices in critical path early. For example, some SOCs require starting security-related services before starting SurfaceFlinger. Review the system log when ServiceManager returns "wait for service" — this is usually a sign that a dependent service must be started first.
- Remove any unused services and commands in init.*.rc. Anything not used in early stage init should be deferred to boot completed.

Using scheduler tuning

Use scheduler tuning for early boot. Example from a Google Pixel:

```
on init
    # update cpusets now that processors are up
    write /dev/cpuset/top-app/cpus 0-3
    write /dev/cpuset/foreground/cpus 0-3
    write /dev/cpuset/foreground/boost/cpus 0-3
    write /dev/cpuset/background/cpus 0-3
    write /dev/cpuset/system-background/cpus 0-3
    # set default schedTune value for foreground/top-app (only affects EAS)
    write /dev/stune/foreground/schedtune.prefer_idle 1
    write /dev/stune/top-app/schedtune.boost 10
    write /dev/stune/top-app/schedtune.prefer_idle 1
```

Some services may need a priority boost during boot. Example:

```
init.zygote64.rc:
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server
    class main
    priority -20
    user root
...
```

Starting zygote early

Devices with file-based encryption can start zygote earlier at the zygote-start trigger (by default, zygote is launched at class main, which is much later than zygote-start). When doing this, make sure to allow zygote to run in all CPUs (as the wrong cpuset setting may force zygote to run in specific CPUs).

Optimizing boot animation

Use the following tips to optimize the boot animation.

Configuring early start

Android 8.0 enables starting boot animation early, before mounting userdata partition. However, even when using the new ext4 tool chain in Android 8.0, fsck is still triggered periodically due to safety reasons, causing a delay in starting the bootanimation service.

To make bootanimation start early, split the fstab mount into two phases:

- In the early phase, mount only the partitions (such as `system/` and `vendor/`) that don't require run checks, then start boot animation services and its dependencies (such as servicemanager and surfaceflinger).
- In the second phase, mount partitions (such as `data/`) that do require run checks.

Boot animation will be started much faster (and in constant time) regardless of fsck.

Finishing clean

After receiving the exit signal, bootanimation plays the last part, the length of which can slow boot time. A system that boots quickly has no need for lengthy animations which could effectively hide any improvements made. We recommend making both the repeating loop and finale short.

Optimizing SELinux

Use the following tips to optimize SELinux for improved boot times.

- **Use clean regular expressions (regex)**. Poorly-formed regex can lead to a lot of overhead when matching SELinux policy for `sys/devices` in `file_contexts`. For example, the regex `/sys/devices/.*abc.*(/.*)?` mistakenly forces a scan of all

/sys/devices subdirectories that contain "abc", enabling matches for both /sys/devices/abc and /sys/devices/xyz/abc. Improving this regex to /sys/devices/[^/]*abc[^/]*(/.*)? will enable a match only for /sys/devices/abc.

- **Move labels to** genfscon (https://selinuxproject.org/page/FileStatements#genfscon). This existing SELinux feature passes file-matching prefixes into the kernel in the SELinux binary, where the kernel applies them to kernel-generated filesystems. This also helps fix mislabeled kernel-created files, preventing race conditions that can occur between userspace processes attempting to access these files before relabeling occurs.

Tool and methods

Use the following tools to help you collect data for optimization targets.

Bootchart

Bootchart provides CPU and I/O load breakdown of all processes for the whole system. It doesn't require rebuilding system image and can be used as a quick sanity check before diving into systrace.

To enable bootchart:

```
$ adb shell 'touch /data/bootchart/enabled'
$ adb reboot
```

After boot up, fetch boot chart:

```
$ANDROID_BUILD_TOP/system/core/init/grab-bootchart.sh
```

When finished, delete /data/bootchart/enabled to prevent collecting the date every time.

Systrace

Systrace allows collecting both kernel and Android traces during boot up. Visualization of systrace can help in analyzing specific problem during the boot-up. (However, to check the average number or accumulated number during the entire boot, it is easier to look into kernel trace directly).

To enable systrace during boot-up:

- In frameworks/native/atrace/atrace.rc, change:

  ```
  write /sys/kernel/debug/tracing/tracing_on 0
  ```

  To:

  ```
  #write /sys/kernel/debug/tracing/tracing_on 0
  ```

  This enables tracing (which is disabled by default).

- In the device.mk file, add the following line:

  ```
  PRODUCT_PROPERTY_OVERRIDES +=    debug.atrace.tags.enableflags=802922
  ```

- In the device BoardConfig.mk file, add the following:

  ```
  BOARD_KERNEL_CMDLINE := ... trace_buf_size=64M trace_event=sched_wakeup,sched_switch,sched_blocked_reason,sched_
  ```

  For detailed I/O analysis, also add block and ext4.

- In the device-specific init.rc file, make the following changes:
  - **on property:sys.boot_completed=1** (this stops tracing on boot complete)
  - **write /d/tracing/tracing_on 0**
  - **write /d/tracing/events/ext4/enable 0**

- **`write /d/tracing/events/block/enable 0`**

After boot up, fetch trace:

```
$ adb root && adb shell "cat /d/tracing/trace" < boot_trace
./external/chromium-trace/catapult/tracing/bin/trace2html boot_trace --output boot_trace.html
```

Note: Chrome cannot handle files that are too large. Consider cutting the `boot_trace` file using `tail`, `head`, or `grep` for necessary portions. And I/O analysis often requires analyzing the captured `boot_trace` directly, as there are too many events.