Lessons learnt in boot time reduction

Andrew Murray

Embedded Bits
Understand why and how
Why minimal boot times?

• There are lots of good reasons, why are you here?
• Common reasons are:
  • To improve user experience in mobile devices
  • To respond quickly after power loss
How

- Starting point:
  - We have a product that achieves required functionality after an unacceptable delay
  - The required functionality is achieved through a number of software tasks that we’ve defined and so can change
  - We can’t change the hardware or the required functionality
  - Software is generalised

- We reduce the cold boot time by:
  - Ensuring that we perform nothing more than the essential tasks required to achieve the required functionality whilst making efficient use of the hardware resources.
  - Thus:
    - Remove tasks that aren’t required
    - Optimise tasks that are required
    - Use all the hardware all the time
    - Challenge how we provide the functionality
2 Know your hardware
Know your hardware

• Understand the performance limits of the hardware
  • Determine what you can expect to achieve and strive to achieve it
  • How quickly can you expect to execute instructions? read data from storage? send I2C/SPI commands?
  • Software will rarely already be optimised for your hardware

• Focus on I/O
  • I/O is always a bottleneck
How much data?

• Use /proc/diskstats to measure userspace I/O at the end of boot

```bash
cat /proc/diskstats | grep mmcblk0p6
179 6 mmcblk0p6 1366 871 122870 42690 0 0 0 0 4 4590 42650
```

122870 sectors read x 512 = 60MB

• Data needed for boot is read from your hardware and takes time
• If you can read at 10MB/S, then your boot will certainly be greater than 6 seconds.
  • Ignoring compression for now
• Thus we optimise by **maximising I/O rate** and **minimising data amount**
Optimise I/O

• Read the specs for your hardware
• Calculate any necessary timings
• Ensure the driver is optimal
  • Consider supporting additional hardware features
• You will need to do this for both kernel and boot loader
• Optimising I/O comes first – later optimisations will depend on the performance of I/O.
Know your hardware

• Ensure you can use:
  • DMA
  • Hardware accelerators
  • Additional CPU cores
  • Caches

• Utilise maximum bus/clock rates
  • CPU
  • I2C/SPI
3 Use compression wisely
Use compression wisely

• I/O is often a bottle neck – it’s slow to read data from storage
• Compression is a trade off between CPU and I/O use
• Determine throughput of reading data directly and reading data with compression to determine optimal solution
• Performance of I/O and algorithms may vary between kernel and bootloader
• Compression is littered everywhere:
  • Compressed kernels (self decompressed, boot loader decompressed)
  • Filesystems
  • Applications
  • Compressibility?
4 The future is quicker
The future is quicker

• Your product development may not be using the latest versions of open source components
  • This wouldn’t be uncommon

• The platform vendors version of software is often used
  • It probably ‘just works’
  • It probably demonstrates the key hardware features you will use
  • It’s probably old
# Vendor provided BSPs

<table>
<thead>
<tr>
<th>Platform</th>
<th>Linux version</th>
<th>U-Boot version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mainline</td>
<td>v3.17 (Oct ‘14)</td>
<td>v2014.07 (Jul ‘14)</td>
</tr>
<tr>
<td>Freescale i.MX6</td>
<td>v3.0.35 (Jul ‘11)</td>
<td>v2009.08 (Aug ‘09)</td>
</tr>
<tr>
<td></td>
<td>v3.10.17 (Jun ‘13)</td>
<td>v2013.04 (Apr ‘13)</td>
</tr>
<tr>
<td>Variscite OMAP4</td>
<td>v3.4.x (May ‘12)</td>
<td>v2012.07 (Jul ‘12)</td>
</tr>
<tr>
<td>Gumstix Oveo DM37xx/OMAP35xx</td>
<td>v3.5.x (Jul ‘12)</td>
<td>v2013.10 (Oct ‘13)</td>
</tr>
<tr>
<td>Gumstix DuoVero OMAP4430</td>
<td>v3.6.x (Sep ‘12)</td>
<td></td>
</tr>
<tr>
<td>Gumstix Verdex Pro PXA270</td>
<td>v2.6.37 (Jan ‘11)</td>
<td>v2010.12 (Dec ‘10)</td>
</tr>
<tr>
<td>DVSDK 4 02 DM3730 OMAPL-138</td>
<td>v2.6.37 (Jan ‘11)</td>
<td></td>
</tr>
</tbody>
</table>

As of Oct ‘14
Take from the future

- Software often gets optimized over time
  - Upstream versions may be quicker with fewer bugs
  - Upstream versions may be slower with more bugs
- Look to the future and backport before reinventing the wheel
- Example...
commit 34fe8281d7323784544e94d2f7218f52b8a2899d
Author: Matthias Weisser <weisserm@arcor.de>
Date: Sun May 22 23:06:50 2011 +0000

arm: lib: memcpy: Do not copy to same address

In some cases (e.g. bootm with a elf payload which is already at the right position) there is a in place copy of data to the same address. Catching this saves some ms while booting.

Signed-off-by: Matthias Weisser <weisserm@arcor.de>

commit d8834a1323af72f6145bc81adad775185ef6065f
Author: Matthias Weisser <weisserm@arcor.de>
Date: Thu Mar 10 21:36:32 2011 +0000

arm: Use optimized memcpy and memset from linux

Using optimized versions of memset and memcpy from linux brings a quite noticeable speed (x2 or better) improvement for these two functions.

Here are some numbers for test done with jadecpu

<table>
<thead>
<tr>
<th></th>
<th>HEAD(1)</th>
<th>HEAD(1) +patch</th>
<th>HEAD(2)</th>
<th>HEAD(2) +patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset to prompt</td>
<td>438ms</td>
<td>330ms</td>
<td>228ms</td>
<td>120ms</td>
</tr>
<tr>
<td>TFTP a 3MB img</td>
<td>4782ms</td>
<td>3428ms</td>
<td>3245ms</td>
<td>2820ms</td>
</tr>
<tr>
<td>FATLOAD USB a 3MB img*</td>
<td>8515ms</td>
<td>8510ms</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>BOOTM LZO img in RAM where CRC is</td>
<td>3473ms</td>
<td>3168ms</td>
<td>592ms</td>
<td>592ms</td>
</tr>
<tr>
<td></td>
<td>615ms</td>
<td>615ms</td>
<td>54ms</td>
<td>54ms</td>
</tr>
<tr>
<td></td>
<td>3450s</td>
<td>3450s</td>
<td>450s</td>
<td>450s</td>
</tr>
</tbody>
</table>
The usual suspects

• String routines get called a lot
  • mem[chr|cpy|move|set|zero]

• OMAP4 Linux boot statistics:
  • 133,356 times transferring 34MB

• Over time these functions have improved

• Other functions in the kernel
  • memcpy_fromio
    • See arch/sh/kernel/io.c
    • See arch/powerpc/kernel/io.c
Ignore FS benchmarks
Ignore filesystem benchmarks

• Benchmarks are valuable but shouldn’t be used exclusively to determine choice
• It’s not easy to find a relevant benchmark
• Software evolves over time
  • Benchmark may not be relevant to your filesystem version – optimisations may be missing/present
• Lots of filesystem choice
  • Filesystem type
  • Block size
  • Use of compression
• There isn’t a best – each option balances CPU vs I/O in some way
  • So it depends on how fast/slow your CPU and I/O is (or how busy your system is)
Ignore filesystem benchmarks

- Boot time consists of:
  - Init time
  - Mount time
  - Read/write time
- Thus depends on your use case
- Of course your usecase, I/O and possibly CPU performance may improve over the course of optimisation effecting your filesystem choice
- Thus measure yourself – you can automate it
The right approach
The right approach

• Measure, measure, measure
• Instrument your software
  • Printk timestamps, host timestamps, initcall_debug, printk’s, tracing
  • But watch out for unintended side effects
• Identify long delays and tackle them first
• Change one thing at a time and measure the full effect
• Make notes, keep log and be prepared to start over
• Order is important
• Goal:
  • 1. Ensure I/O and CPU is 100% utilised during boot
  • 2. Ensure the I/O and CPU utilisation is efficient and necessary
• Consider full effect of a change
Don’t focus on the kernel

• If you perform any measurement you’ll quickly realise that the kernel is rarely the worse offender
• Typically the bootloader and userspace take up the majority of boot time
8 Low hanging fruit
Low Hanging Fruit

• There are trivial changes that generally always improve the boot time of the majority of devices
  • These are applied to the majority of boot time reduction projects
• No boot time presentation would be complete without these
U-Boot delay

• Eliminate boot delay with CONFIG_BOOTDELAY
• Allow development with CONFIG_ZERO_BOOTDELAY_CHECK
• Use CONFIG_AUTOBOOT_KEYED to prevent accidents

Boot Delay: CONFIG_BOOTDELAY - in seconds
Delay before automatically booting the default image;
set to -1 to disable autoboot.
set to -2 to autoboot with no delay and not check for abort
(even when CONFIG_ZERO_BOOTDELAY_CHECK is defined).

See doc/README.autoboot for these options that
work with CONFIG_BOOTDELAY. None are required.
CONFIG_BOOT_RETRY_TIME
CONFIG_BOOT_RETRY_MIN
CONFIG_AUTOBOOT_KEYED
CONFIG_AUTOBOOT_PROMPT
CONFIG_AUTOBOOT_DELAY_STR
CONFIG_AUTOBOOT_STOP_STR
CONFIG_AUTOBOOT_DELAY_STR2
CONFIG_AUTOBOOT_STOP_STR2
CONFIG_ZERO_BOOTDELAY_CHECK
CONFIG_RESET_TO_RETRY
Precalculated LPJ

- Use the ‘lpj’ argument in the kernel command line to prevent calibrating a delay loop on each boot

Calibrating delay loop... 1590.23 BogoMIPS (lpj=6213632)
Disable console output

• Add ‘quiet’ to the kernel command line to suppress kernel output during boot
• Suppress U-Boot output with:
  • CONFIG_SILENT_CONSOLE
  • CONFIG_SILENT_CONSOLE_UPDATE_ON_RELOC
  • CONFIG_SILENT_U_BOOT_ONLY
#define CONFIG_BOOTCOMMAND \
"if mmc rescan ${mmcdev}; then " \ 
  "if run loadbootenv; then " \ 
    "echo Loaded environment from ${bootenv};" \ 
    "run importbootenv;" \ 
    "if test -n $uenvcmd; then " \ 
      "echo Running uenvcmd ...;" \ 
      "run uenvcmd;" \ 
    "fi;" \ 
  "elif run loadbootscript; then " \ 
    "echo Loaded script from ${bootscr};" \ 
    "run bootscript; " \ 
  "else " \ 
    "if run loaduimage; then " \ 
      "run mmcboot; " \ 
    "fi; " \ 
  "fi; " \ 
"fi; " \ 
"if usb start; then " \ 
  "set autoload no; "\"
More fruit

- Reduce U-Boot environment size
- Ensure U-Boot reads the right amount of data and puts it in the right place
  - ‘nboot’ can help
- Image verification
- Optimise memset/memcpy
- Reduce size of binaries
  - Kernel (KALLSYMS, IKCONFIG), strip binaries, mklibs, etc
- Device node creation
- Init.d style boot
- Sleeps
9 Make it stick
Make it stick

• Boot time is likely to increase through subsequent development
  • And so boot time reduction can be a repeating exercise

• Reasons boot time doesn't stick:
  • Development of new features doesn’t consider boot time
  • Boot time enhancements can be inflexible, inconvenient and limiting

• Inconvenient optimisations have a habit of disappearing
  • No one ever removes a ‘no side effect’ optimisation such as improved NAND timings

• Maintaining a minimal boot time is difficult
Bad solutions

• Delay: Ethernet initialisation in U-Boot
• Bad solution: Ethernet not used in U-Boot so remove support
• Impact: No-one can TFTP – annoying for development

• Better solution:
  • Defer initialisation
  • Add a custom U-Boot command that re-enables Ethernet
  • Developers can add this command to their ‘bootcmd’
  • If U-Boot size is an issue – it can be conditionally removed for a release build
  • Run time options such as this that default to quick boot are great
  • Compile/Config time options may result in git commit fights
  • Don’t replace - complement
Bad solutions

• Delay: Udev
• Bad solution: Remove it and use hard coded device nodes
• Impact: It works great until the moment the kernel changes then everyone gets odd errors, eventually udev shows up again

• Better solution:
  • Use devtmpfs
  • Update drivers to use hard coded device number (if you must)
Make it stick

• Make boot time enhancements less annoying
  • It’s easy to use a sledge hammer
  • Find good solutions that preserve or replace ‘unneeded’ functionality
    • Sometimes unavoidable for tiny boot times
  • Aim for run time switches to enable them
    • Maybe use a hardware switch

• Automate boot time measurement
  • Use automation to build, deploy, boot and measure your boot time
  • Tie this in with git hooks
  • Find out at the earliest moment when your boot time increases
    • Helps share ownership of boot time reduction
Look in the right places
Look in the right places

• You’ll get to a point where...
  • You’ve read the internet and implemented its suggestions
  • You’ve implemented the low hanging fruit
  • You’ve think everything left is needed and it does what it needs to do
  • Where next?
Look in the right places

• You’ve probably been looking at big delays
  • Don’t forget the small delays that happen all the time
  • Harder to find – you need to consider both time and frequency
    • Profiling tools helps here
  • This is probably where the hard work begins

• Google ‘instrumentation/tracing’ instead of ‘reduce boot time’

• Don’t forget to group related delays to truly see impact
  • You need to do that fsck because you have chosen a file system that needs it

• Ask yourself if the tasks are really needed?
  • Do they directly contribute to achieving the boot time functionality?
    • Bootloaders, start up scripts, dynamic device nodes – nice but not essential
GStreamer example

• Gstreamer heavily uses plugins (shared libraries)
• When a Gstreamer application starts (gst_init) it will scan GST_PLUGIN_PATH for plugins and populate a registry used to find plugins
  • This takes time
  • This isn’t needed on every boot
• Example of a task that isn’t needed
• Used GST_REGISTRY_UPDATE to prevent scan and used a pre-provided registry
• Reduced data transfer during boot by 20MB saved 4 seconds
• Found by littering printf’s through the application
Look in the right places

• Look at:
  • Code you’ve written
  • Code your team has written
  • Code the board provider has written
  • Code the silicon vendor has written
  • Everything else
10 Find good tools
Tools can help

- Tools can save you time and provide clarity
  - There are a lot of tools available
  - Sometimes they can overwhelm you with information
  - Use them towards end of project
- You can do a lot with printk/printf and timestamps
- Perf, oprofile, strace are helpful too
  - Lots of presentations on these topics
  - “Linux Performance Tools” – Brendan Gregg, Netflix
- Bootchart can help visualise userspace performance
Boot chart for Your Product (Thu Mar 7 11:22:12 PST 2013)
uname: Linux 3.6.39-08109-gc545a22 #13 PREEMPT Thu Mar 7 09:15:22 PST 2013 am5tej
release:
CPU:
kernel options: mem=50M console=ttyS0,115200n8 rootwait root=/dev/mmcblk0p1 ro rootfstype=ext2 int=/bootchartd

time: 0:45

CPU (user + sys)  I/O (wait)

Disk throughput  Disk utilization

Running (%cpu)  Unint. sleep (%/s)  Sleeping  Zombie

int
bootchartd

rc
503uday
sh
rc
mount
logger
modprobe
udev
rc
dbus-daemon
590detect-new
sleep
599your-application

bluetoothd
dbus-daemon
app-monitor
your-application

sh
scan-media.sh
mount
web-app
radio-scanner

gst-plugin-scan
bluetooth
update-monitor
getty

kthread
ksoftirqd/0
kworker/0:
kworker/0:
khelper
kworker/1:
ir0/00-1-0020
mmcons/0

led0/mmcblk0p7:
scd4-dis-uninit

Embedded Linux Conference Europe 2014
Create an illusion
Create an illusion

• Why do you want a minimal boot time anyway?
  • Can you achieve it without a minimal boot time?
• Small boot time can be difficult to achieve
  • So why not create an illusion of a small boot time instead
  • ‘Smoke and Mirrors’ have a place
• Defer functionality that isn’t needed straight away
12 Expect disruption
Expect disruption

• Minimal boot times often result in a compromise
• This effects everyone that is developing the product
• Changes you wish to make may conflict with other parts of the product development
• Your changes may have knock on effects
• Be prepared to make and justify your case
  • How important is a minimal boot time?
  • You may find out that minimal boot time isn’t everything after all
Thank you

Questions?

Andrew Murray
Embedded Bits <amurray@embedded-bits.co.uk>