The Right Approach to Minimal Boot Times

Andrew Murray
Senior Software Engineer

CELF Embedded Linux Conference Europe 2010
Senior Software Engineer, MPC Data
- Driver and kernel development
- Embedded applications development
- Windows driver development

Work Experience
- 4 years of experience working with embedded Linux devices
- Good track record in dramatically reducing customers’ boot time through MPC Data’s boot time reduction service: swiftBoot
  - Tight timescales often doesn’t permit nice, elegant and generic solutions – however this frustration had provided me with many ideas I wish to share today
  - I also wish to share my observations and experiences in boot time reduction
Agenda
Delivering Software Innovation

- Principals behind boot time reduction
- My approach to boot time reduction
- Case Study: MS7724 ‘Ecovec’
- Optimizing user space and function reordering
- Video Demonstration
- Conclusion and Q&A
The problem:
- Getting an embedded Linux based product from power-on to a useful state of functionality in an acceptable amount of time.

Many innovative solutions exist: Suspend / Hibernate / etc

This presentation focuses on cold-boot optimisation
- Specialising software for specific needs of a product
- And this works because prior to optimisation the software will be:
  - More General purpose
    - Likely to contain functionality your device doesn’t require which will result in more initialisation and a larger image
  - More Convenient and flexible
  - Likely to probe and detect hardware which you know will always be there which will contribute to boot delay.
- There is no silver bullet here – all that is required is: Disciplined Analysis + Common Sense + Pragmatic Approach
The *swiftBoot* Approach

Delivering Software Innovation

1. Identify boot time functionality
2. Measure boot time across the board
3. Remove unnecessary functionality
4. Optimise required functionality
5. Re-order initialisation
The **swiftBoot** Approach

*Delivering Software Innovation*

- Identify boot time functionality
- Measure boot time across the board
- Remove unnecessary functionality
- Optimise required functionality
- Re-order initialisation

- **Understand what functionality is required:**
  - Immediately after boot
  - Sometime after
- **The better your understanding the more able you are to specialise Linux and thus improve boot time**
Identify boot time functionality

Measure boot time across the board

Remove unnecessary functionality

Optimise required functionality

Re-order initialisation

- It is important to visualise what contributes to boot time
- Measuring boot time across the entire software stack is essential
- Without tools, gauging small boot delays can be impossible
- Being able to accurately measure boot time across the board will allow you to measure the effect of any changes you make...
- ...otherwise you’ll be lost in the dark
The swiftBoot Approach

Delivering Software Innovation

- Identify boot time functionality
- Measure boot time across the board
- Remove unnecessary functionality
- Optimise required functionality
- Re-order initialisation

- Unnecessary functionality will increase boot time due to
  - Increased image size (flash transfer time)
  - Time spent initialization during start up
- “But I might use this feature in the future, it’s nice to have”
  - Be strict and stick to the brief
The SwiftBoot Approach

Delivering Software Innovation

1. Identify boot time functionality
2. Measure boot time across the board
3. Remove unnecessary functionality
4. Optimise required functionality
5. Re-order initialisation

- Functionality you require can be optimised
  - It may already be optimised in a later version of sources
- This may involve:
  - Optimising flash timings
  - Removing unnecessary probing / delays
  - Refactoring code
  - Taking a new approach to problems
The swiftBoot Approach
Delivering Software Innovation

Identify boot time functionality → Measure boot time across the board → Remove unnecessary functionality → Optimise required functionality → Re-order initialisation

- Further improvements can be gained by doing things at different times:
  - Parallelisation
    - Using Arjan’s async framework (*kernel/async.c*)
  - Deferred loading of less important features
    - Loadable kernel modules
Case Study
Delivering Software Innovation

- Use the MS7724 ‘EcoVec’ as a case study for a home automation system

- Boot time functionality:
  - Responsive QT user interface

- Additional functionality:
  - Video capture/render (representing a security camera)

- Will describe tools, techniques and lessons along the way
### SH7724

**Delivering Software Innovation**

- **CPU**
- **MEMORY**
- **TIMER**
- **ANALOG**
- **I/O**
- **MULTIMEDIA**

#### Overview

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I-Cache</strong></td>
<td>32KB</td>
</tr>
<tr>
<td><strong>D-Cache</strong></td>
<td>32KB</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>256KB</td>
</tr>
<tr>
<td><strong>FPU</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MAC</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MMU</strong></td>
<td></td>
</tr>
<tr>
<td><strong>UBC</strong></td>
<td></td>
</tr>
<tr>
<td><strong>INTC</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DMAC x6</strong></td>
<td></td>
</tr>
<tr>
<td><strong>H-UDI</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CPG</strong></td>
<td></td>
</tr>
<tr>
<td><strong>LCDC</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Others</strong></td>
<td></td>
</tr>
<tr>
<td><strong>WDT</strong></td>
<td></td>
</tr>
<tr>
<td><strong>TMU x3</strong></td>
<td></td>
</tr>
<tr>
<td><strong>BSC</strong></td>
<td></td>
</tr>
<tr>
<td><strong>VPU5F</strong></td>
<td>H.264 D1@60fps 720p@30fps</td>
</tr>
<tr>
<td><strong>VOU</strong></td>
<td></td>
</tr>
<tr>
<td><strong>JPEG</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SPU 24-bit DSP</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CEU x2</strong></td>
<td>Camera I/F</td>
</tr>
<tr>
<td><strong>BEU x2</strong></td>
<td>Blend</td>
</tr>
<tr>
<td><strong>VEU x2</strong></td>
<td>Scaling</td>
</tr>
<tr>
<td><strong>2DG</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DDR2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MMC NAND</strong></td>
<td></td>
</tr>
<tr>
<td><strong>KeyScan</strong></td>
<td></td>
</tr>
<tr>
<td><strong>10/100 Ethernet</strong></td>
<td>MAC w/DMA</td>
</tr>
<tr>
<td><strong>I2C x2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SDIO x2</strong></td>
<td></td>
</tr>
<tr>
<td><strong>USB-HS Host</strong></td>
<td>Host or Device w/PHY x2</td>
</tr>
<tr>
<td><strong>SCIF x6</strong></td>
<td></td>
</tr>
<tr>
<td><strong>IrDA</strong></td>
<td></td>
</tr>
<tr>
<td><strong>GPIO</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ATAPI</strong></td>
<td></td>
</tr>
</tbody>
</table>
Case Study
Delivering Software Innovation

- Use the MS7724 ‘EcoVec’ as a case study for a home automation system

- Boot time functionality:
  - Responsive QT user interface

- Additional functionality:
  - Video capture/render (representing a security camera)

- Will describe tools, techniques and lessons along the way
Case Study
Delivering Software Innovation

- Use the MS7724 'EcoVec' as a case study for a home automation system

- Boot time functionality:
  - Responsive QT user interface

- Additional functionality:
  - Video capture/render (representing a security camera)

- Will describe tools and techniques along the way
Case Study
Delivering Software Innovation

- Typical Starting Point:
  - BootLoader: UBoot (2009-01)
  - OS: Linux (2.6.31-rc7)
  - Filesystem: Buildroot (2010.05), JFFS2, NOR Flash
  - Application: QT Embedded Opensource 4.6.2
**MS7724 Boot Process**

**Delivering Software Innovation**

<table>
<thead>
<tr>
<th>Baseline (seconds)</th>
<th>UBoot (2.58 s)</th>
<th>Kernel (1.30 s)</th>
<th>Filesystem Mount (~6.83 s)</th>
<th>Init Scripts (1.30 s)</th>
<th>GUI Application (7.43 s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-to-UI: 19.44 seconds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Component times mostly measured using GPIO and a logic analyser,
  - UBoot time measured between reset and GPIO line being asserted
  - Sources modified to toggle GPIO at key points:
    - UBoot: UBoot to kernel handover
      `(common/cmd_bootm.c:do_bootm)`
    - Kernel: Mount FS
      `(init/do_mounts.c:do_mount_root)`
    - Kernel: Init
      `(init/main.c:init_post)`
  - Used printk timings for the rest
  - Time to required boot time functionality: > 19 seconds!
Delivering Software Innovation

UBoot

Functionality Removal
- User Boot Delay – 1000 ms
- Image Verification – 374 ms
- Image Decompression
- USB, ROMImage, Filesystems – 195 ms

Functionality Optimisation
- Improve ‘memcpy’ code – 342 ms
- Eliminate use of console – 103 ms
- Reduced kernel size – 60ms

Functionality Re-ordering
- Read MAC from EEPROM – 124 ms
- Ethernet setup – 98 ms

Reduction: 2577 ms > 280 ms (89%)
UBoot (Before and After)

Delivering Software Innovation

Before (136 kB)

After (94 kB)
Compressed Kernel Images

- The purpose of the boot loader is to jump to an uncompressed kernel image in RAM
- A number of factors should be considered

Flash Throughput (10.50 MB/s)
Decompression Throughput (2.67 MB/s)

Compression Ratio (0.57)
Decompressor Speed
Speed of Flash (10.5 MB/s)

- If Flash throughput is greater than decompression throughput then an uncompressed image is quicker
Linux Kernel
Delivering Software Innovation

Functionality Removal
- Remove USB – 88 kB – 144 ms
- Remove keyboard driver – 4 ms
- Remove Filesystems – 300 kB – 0.8 ms
- Remove console output

Functionality Optimisation
- Remove delays in driver initialization – 400 ms
- Removing delays - 252 ms
- Prevent probing disconnected cameras – 200 ms
- Limiting memset size – 90 ms
- Improve performance of memset – 71 ms

Functionality Re-ordering
- Defer networking initialization – 166 kB – 20 ms

Reduction: 1301 ms -> 113 ms (91%)
Linux Kernel (Before and After)

Delivering Software Innovation
Userspace (Mount and Init scripts)

Functionality Removal
- Remove all init scripts – use a single init process – 1.32 s

Functionality Optimisation
- Statically link application with uClibc libraries
- Use SquashFS instead of JFFS2 ~6.81 s
- Improve performance of NOR memory driver

Functionality Re-ordering
- Start QT then later start video

Reduction: 8130 ms > 64 ms (99%)
Reducing the boot time of the QT application was the biggest challenge and very time consuming.

Un-optimized QT application was large and took 7.4 seconds to reach it’s main function!

Improvements reduce time to 0.3 seconds:

- Optimise flash access (arch/sh/kernel/io.c) -2.89s
- Removed features -2.16s
- Used statically linked uClibc -0.9s
- Stripped -0.16s
- Optimising executable -0.35s
- Read-ahead and block size -0.63s

Measurements show incremental effects against original binary (from left to right)
Why does QT take so long to start?

- Only a portion of the QT application is required to display a UI to the user
  - Event handling, additional forms, etc come later

- As Linux uses Demand Paging - when an executable is run only parts of the executable used are read from flash
  - This reduces unnecessary flash accesses and decreases application start up time

- However the application is on a block filesystem so when an entire block is retrieved at a time...

- ...This results in unnecessary flash access time if the required executable code is spread over the entire image
Function Reordering and Block Sizes

- Sections highlighted in red represent parts of executable required at start up.
- Most of these parts could fit in a single file-system block.
  - I.e. we could optimise the application such that only 2 blocks of flash are accessed rather than 4.
- Thus the executable can be optimised by:
  - Reducing block size
  - Eliminating FS readahead
  - Reordering executable

```
FS
MTD Layer
Block 1  Block 2  Block 3  Block 4  Block 5  Block 6  Block 7
NOR Flash
QT Application
NOR Flash
```
Function Reordering

Delivering Software Innovation

- GCC compiler features can be used to assist:
  - --finstrument-functions
  - --ffunction-sections

- Before the entry and after the exit of every function call – GCC will call two new functions when --finstrument-functions is used:
  - void __cyg_profile_func_entry (....)
  - void __cyg_profile_func_exit (....)

- These calls can be implemented to find out which functions are called when

- This information can be used to generate a custom linker script – when --function-sections is used each function lives in its own section.

- This way we can ensure all the required sections for startup are contained contiguously in flash

- (--gc-sections can also be helpful)
Function Reordering and Block Sizes

Delivery Software Innovation

Before:

<table>
<thead>
<tr>
<th>FS</th>
<th>MTD Layer</th>
<th>NOR Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>Block 2</td>
<td>Block 3</td>
</tr>
<tr>
<td>Block 4</td>
<td>Block 5</td>
<td>Block 6</td>
</tr>
<tr>
<td>Block 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

QT Application

NOR Flash

After:

<table>
<thead>
<tr>
<th>FS</th>
<th>MTD Layer</th>
<th>NOR Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1</td>
<td>Block 2</td>
<td>Block 3</td>
</tr>
<tr>
<td>Block 4</td>
<td>Block 5</td>
<td>Block 6</td>
</tr>
<tr>
<td>Block 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

QT Application

NOR Flash
Essential tools

Discrete events can be measured by toggling GPIO outputs and utilising a logic analyser,

Kernel events can be measured with:
- Printk timings,
- Initcall_debug and bootchart scripts,

Userspace events can be measured with ubootchart

These are just some of the many tools available
Case Study: Before and After
Delivering Software Innovation

Baseline (seconds)
- UBoot: 2.58 s
- Kernel: 1.30 s
- Filesystem Mount: ~6.83 s
- Init Scripts: 1.30 s
- GUI Application: 7.43 s

Power-to-UI: 19.44 seconds

After swiftBoot Modifications (seconds)
- UBoot: 0.28 s
- Kernel: 0.11 s
- Filesystem Mount: 0.02 s
- Init Scripts: 0.06 s
- GUI Application: 0.30 s

Power-to-UI: 0.77 seconds

A reduction of 96%!
## Summary

**Delivering Software Innovation**

<table>
<thead>
<tr>
<th>Modification In UBoot</th>
<th>Gain (ms)</th>
<th>Modification In Kernel</th>
<th>Gain (ms)</th>
<th>Modification In Userspace</th>
<th>Gain (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove boot delay</td>
<td>1000</td>
<td>Remove driver delays</td>
<td>652</td>
<td>Use squashfs</td>
<td>6830</td>
</tr>
<tr>
<td>Remove Image verification</td>
<td>374</td>
<td>Prevent probing disconnected cameras</td>
<td>200</td>
<td>Optimise flash accesses</td>
<td>2890</td>
</tr>
<tr>
<td>Optimise memcpy code</td>
<td>342</td>
<td>Remove USB</td>
<td>144</td>
<td>Remove unused features from QT</td>
<td>2160</td>
</tr>
<tr>
<td>Remove USB ROMImage, filesystems</td>
<td>195</td>
<td>Don’t allocate memory for unused camera components</td>
<td>90</td>
<td>Remove all init scripts</td>
<td>1300</td>
</tr>
<tr>
<td>Defer reading MAC address</td>
<td>124</td>
<td>Improve memset</td>
<td>71</td>
<td>Statically link QT with uclibc</td>
<td>900</td>
</tr>
<tr>
<td>Reduction due to kernel size</td>
<td>60</td>
<td>Defer network initialisation</td>
<td>20</td>
<td>Reduce readahead and block size</td>
<td>630</td>
</tr>
<tr>
<td>Remove delays in Ethernet init</td>
<td>98</td>
<td>Remove keyboard driver</td>
<td>4</td>
<td>Re-order QT application</td>
<td>350</td>
</tr>
<tr>
<td>Eliminate use of console</td>
<td>103</td>
<td>Remove filesystems</td>
<td>0.8</td>
<td>Strip QT application</td>
<td>160</td>
</tr>
</tbody>
</table>

**Total Gain** 2.2 s  **Total Gain** 1.2 s  **Total Gain** 15.2 s
Guiding Principles

Delivering Software Innovation

- **Observe and Record**
  - Measuring boot times is the only way to form a clear picture of what is contributing to boot time,
  - **Keep copious notes**

- **Tackle the biggest delays in the system first,**
  - Identify the largest delays and remove them to be most effective

- Be aware and try to understand varying boot times
- Remember the uncertainty principle
- Don’t forget testing
Conclusion & Call to action

Reducing cold boot time is like removing the longest links of a chain until you have just short links

- As a result boot time is a product of a system design and long links can be easily added
- Effort will always be required to remove and shorten links for a given system
- Holy grail is to reduce this amount of effort to nothing – some ideas towards this:
  - [Idealism] Asynchronous initialisation in the kernel by default
    - Many challenges here
    - This would reduce effect of delays in drivers
  - [Realism] Simple Caching framework for device probes
    - To eliminate probes for known hardware (generic device tree)
    - Could encompass LPJ, etc
Thank You
Any Questions?
Appendix
Initcall Debug

Delivering Software Innovation

- Add the following to your kernel command line:
  - initcall_debug (to add debug)
  - loglevel=0 (to reduce the impact this has on boot time)

- Ensure the following are set in your kernel configuration:
  - CONFIG_PRINTK_TIME (add timings to printk)
  - CONFIG_KALLSYMS (ensure symbols are there)
  - CONFIG_LOGBUF_SHIFT = 18 (ensure there is room in the log buffer)

- Copy the output of ‘dmesg’
- Type ‘cat output | ./scripts/bootgraph.pl > graph.svg’