Delivering Software Innovation

The Right Approach to Minimal Boot Times

Andrew Murray

Senior Software Engineer

CELF Embedded Linux Conference Europe 2010

Andrew Murray

Delivering Software Innovation

- 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

Principals

Delivering Software Innovation

- 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

Identify boot time functionality

Measure boot time across the board

Remove un-necessary functionality

Optimise required functionality

Re-order initialisation

The swiftBoot Approach

Delivering Software Innovation

Identify boot time functionality

Measure boot time across the board

Remove un-necessary functionality

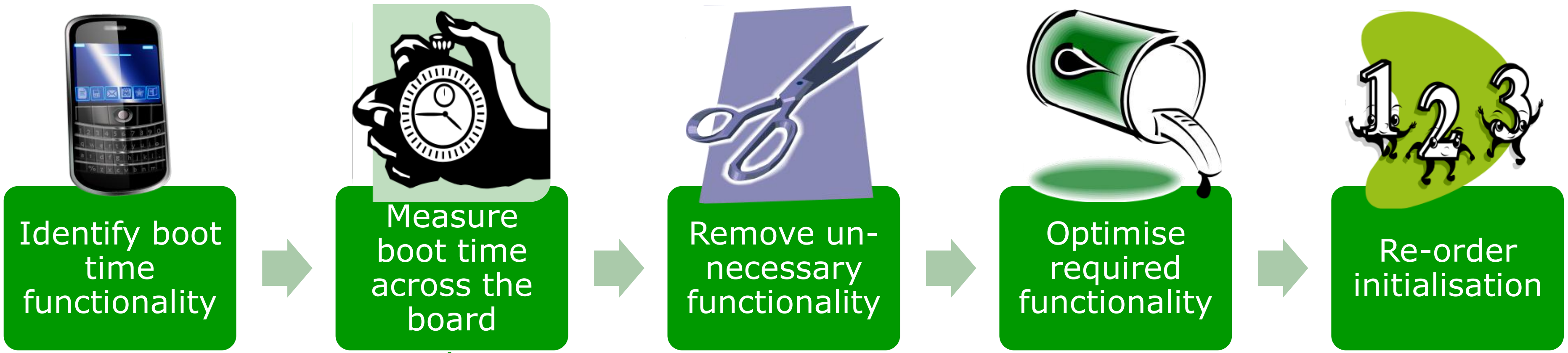
Optimise required functionality

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

The swiftBoot Approach

Delivering Software Innovation



- 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

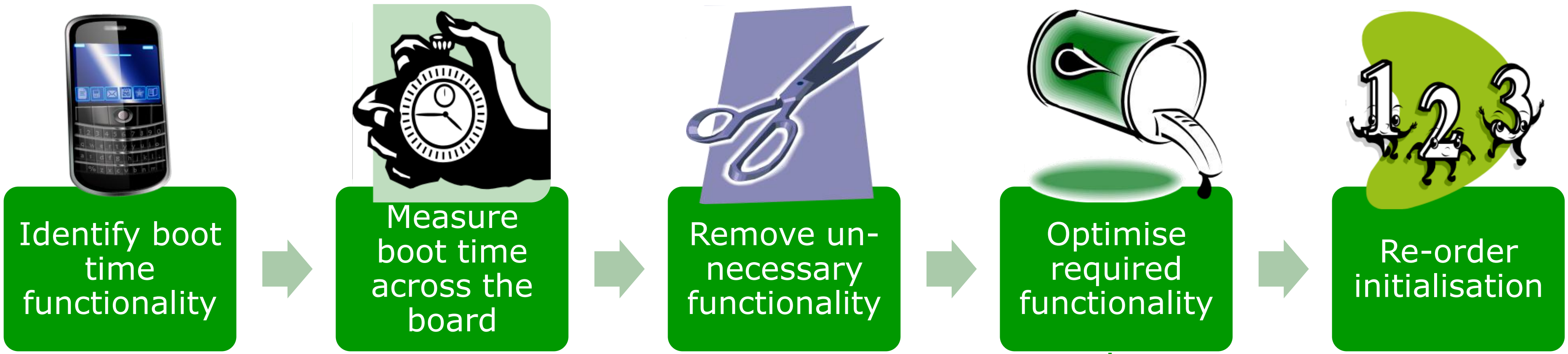
Delivering Software Innovation

1. Identify boot time functionality
2. Measure boot time across the board
3. Remove un-necessary functionality
4. Optimise required functionality
5. 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

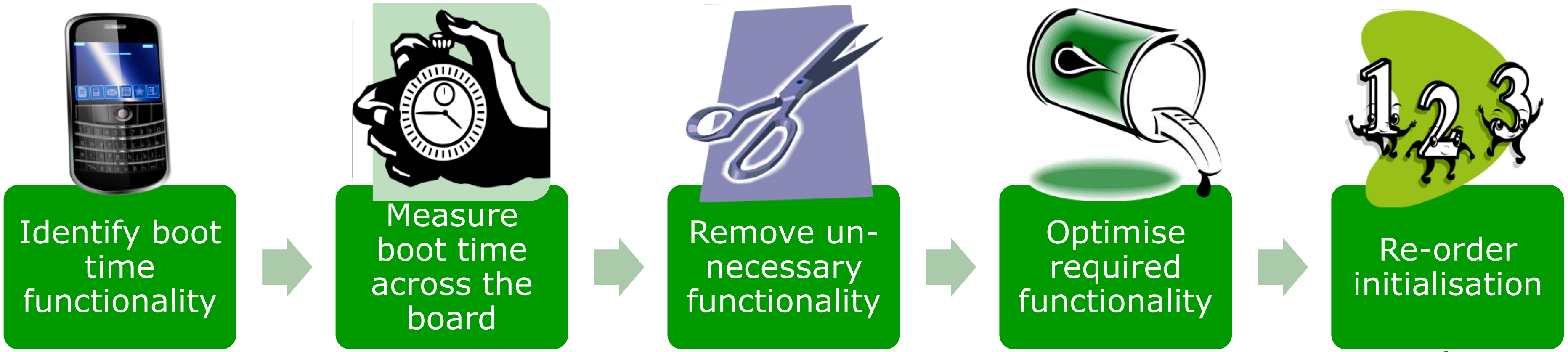
Delivering Software Innovation



- 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



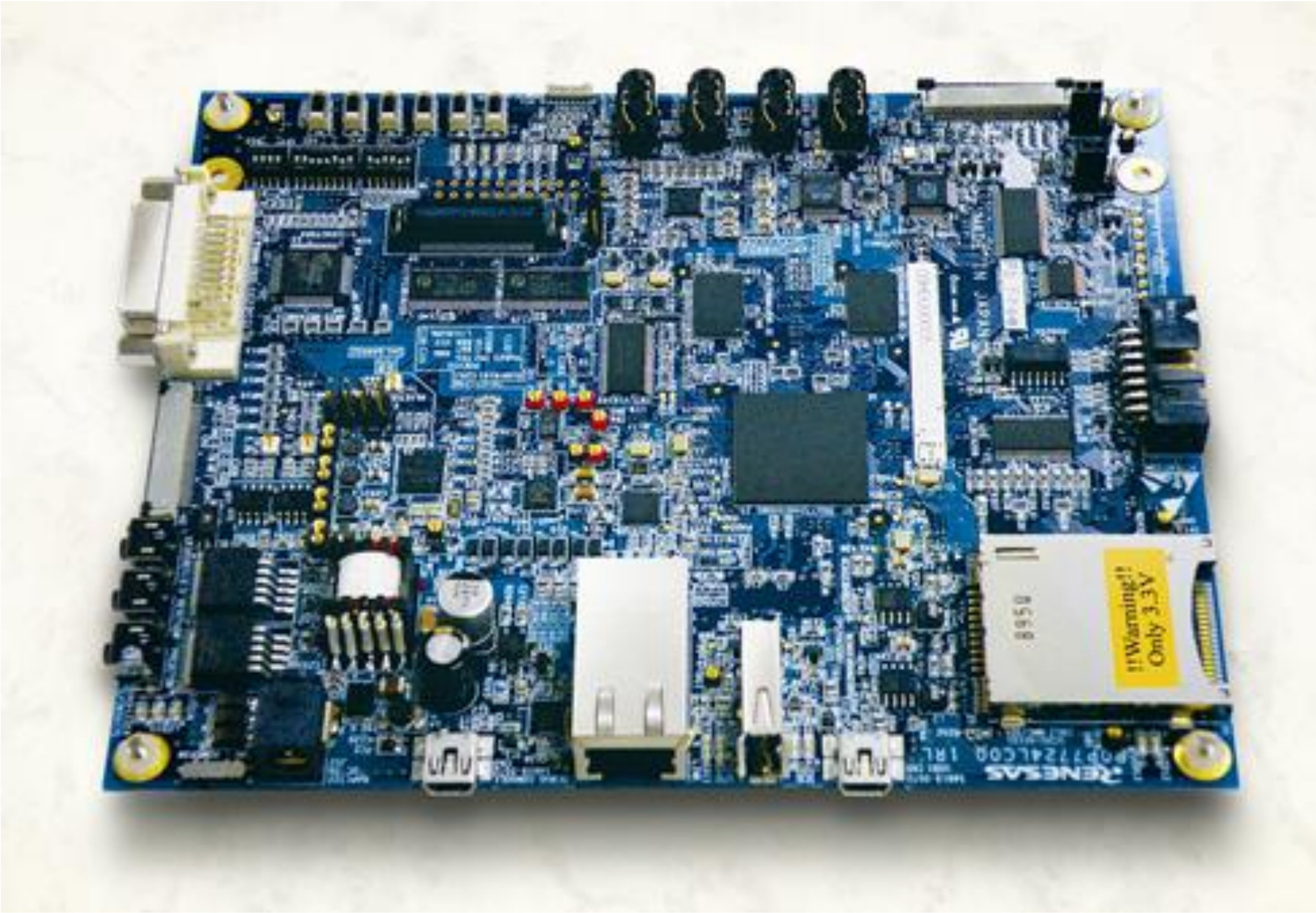
- Further improvements can be gained by doing things at different times:
  - Parallelisation
    - Using Arjan's async framework *(kernel/async.c)*
  - Deferred loading of less important features
    - Loadable kernel modules

Case Study

Delivering Software Innovation

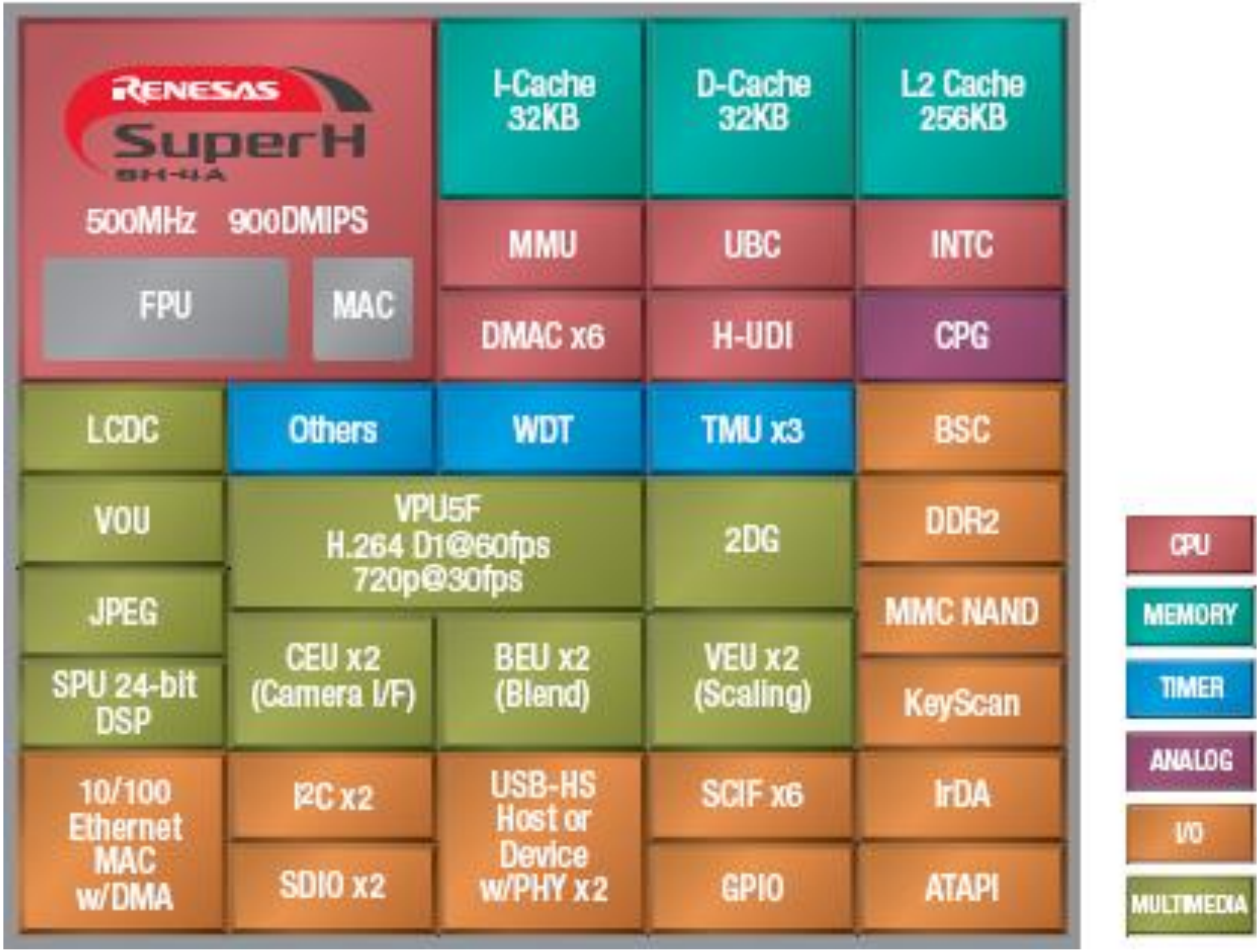
- Use the MS7724 ‘EcoVec’ as a case study for a home automation system
- Boot time functionality:
  - Responsive QT user interface
- Additional functionality:
  - Video capture/render (representing a security camera)
- Will describe tools, techniques and lessons along the way

MS7724



SH7724

Delivering Software Innovation



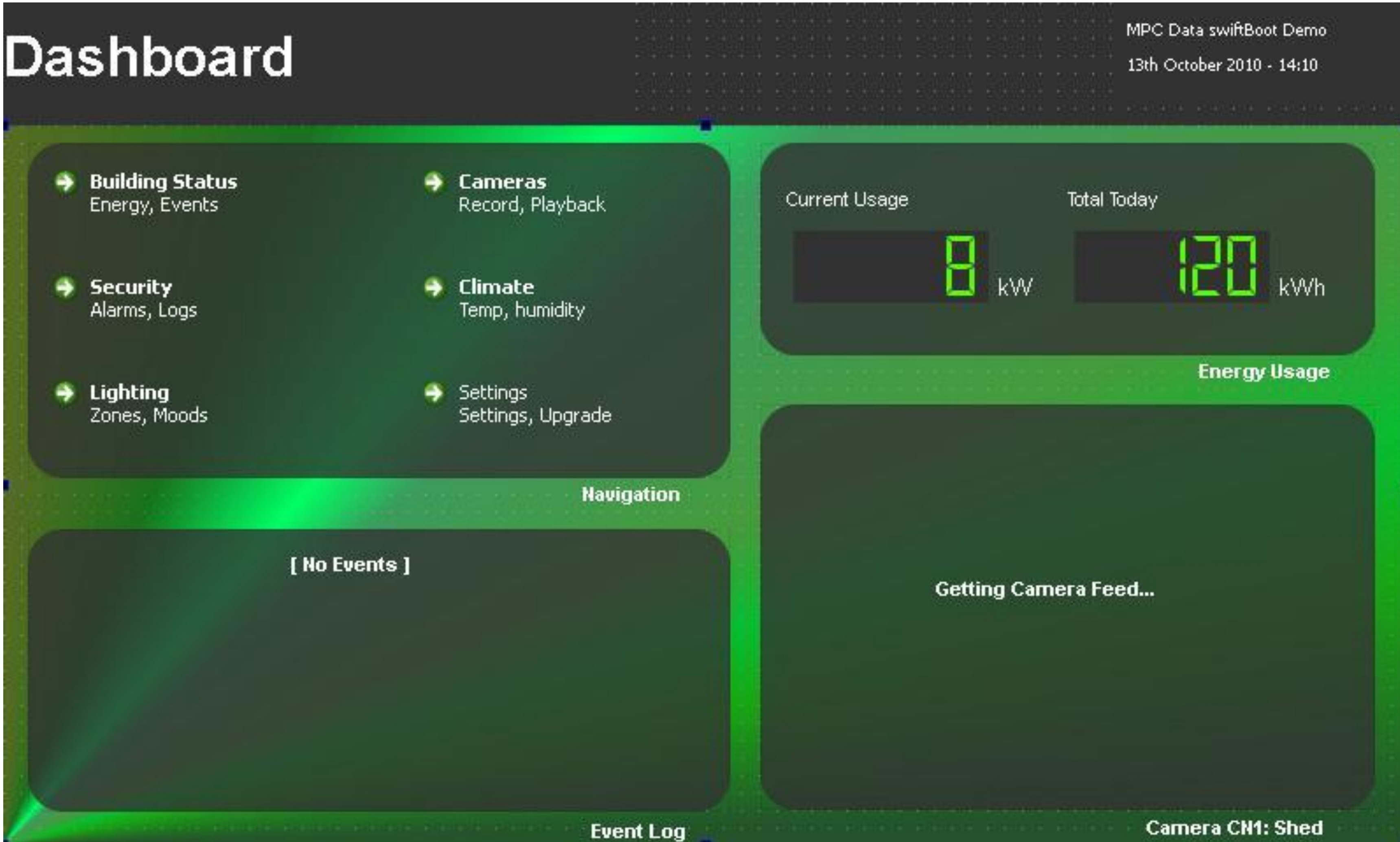
Case Study

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



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

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

- 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

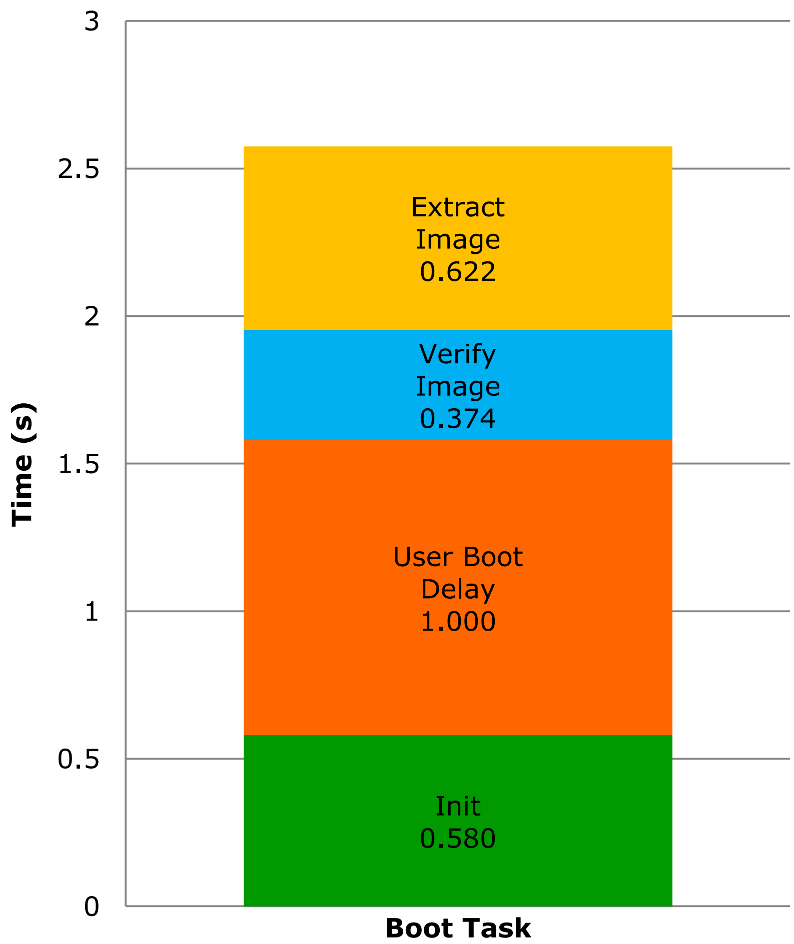
      (common/cmd_bootm.c:do_bootm)
    - Kernel: Mount FS

      (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!

UBoot

Delivering Software Innovation



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

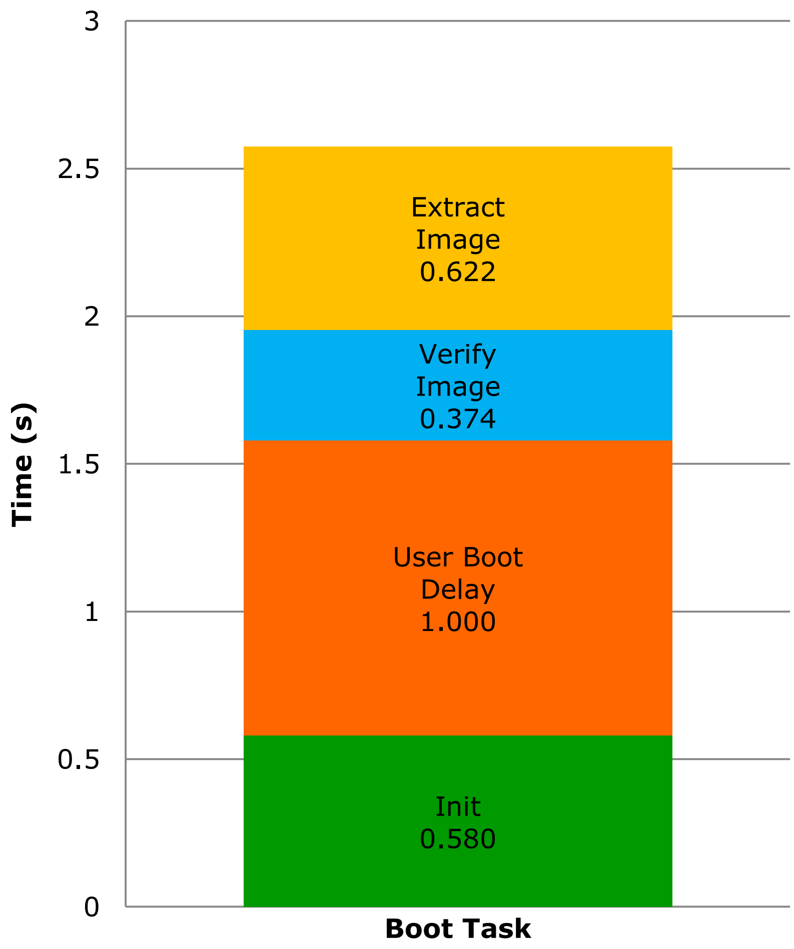
Functionality Re-ordering

- Read MAC from EEPROM – 124 ms
- Ethernet setup – 98 ms

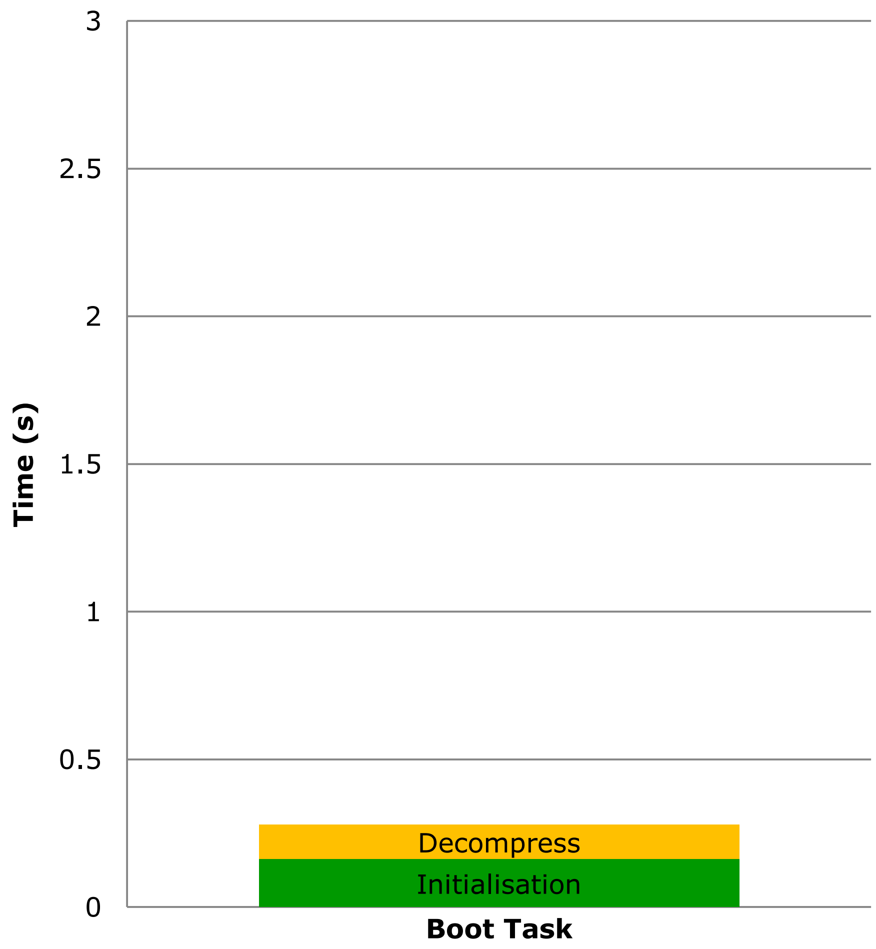
Reduction: 2577 ms > 280 ms (89%)

UBoot (Before and After)

Delivering Software Innovation



Before (136 kB)

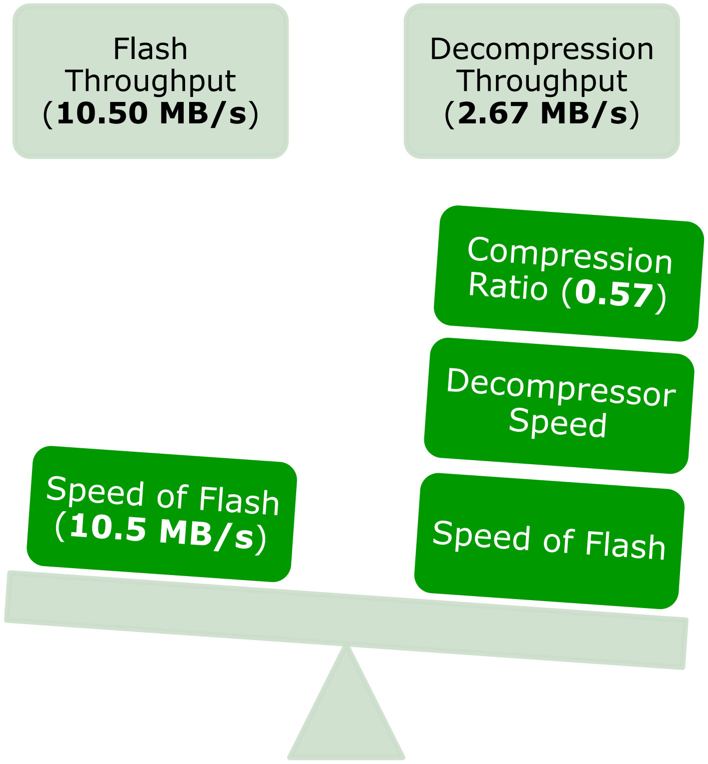


After (94 kB)

Compressed Kernel Images

Delivering Software Innovation

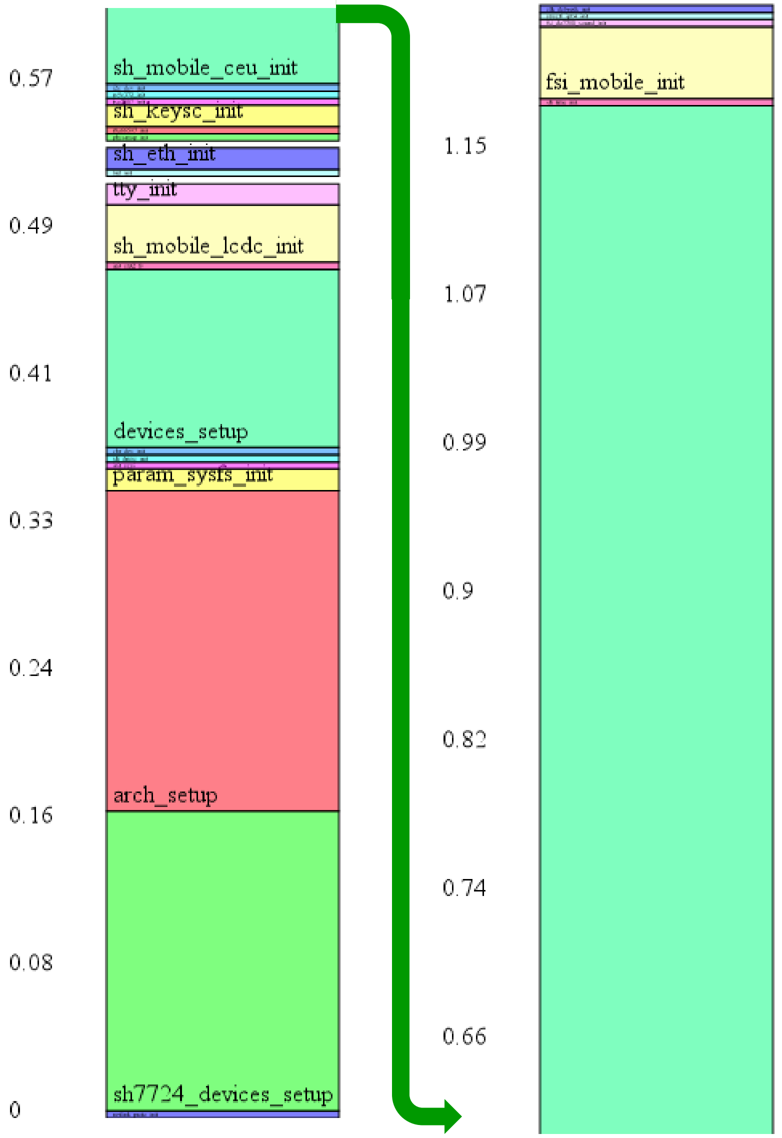
- The purpose of the boot loader is to jump to an uncompressed kernel image in RAM
- A number of factors should be considered



- 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

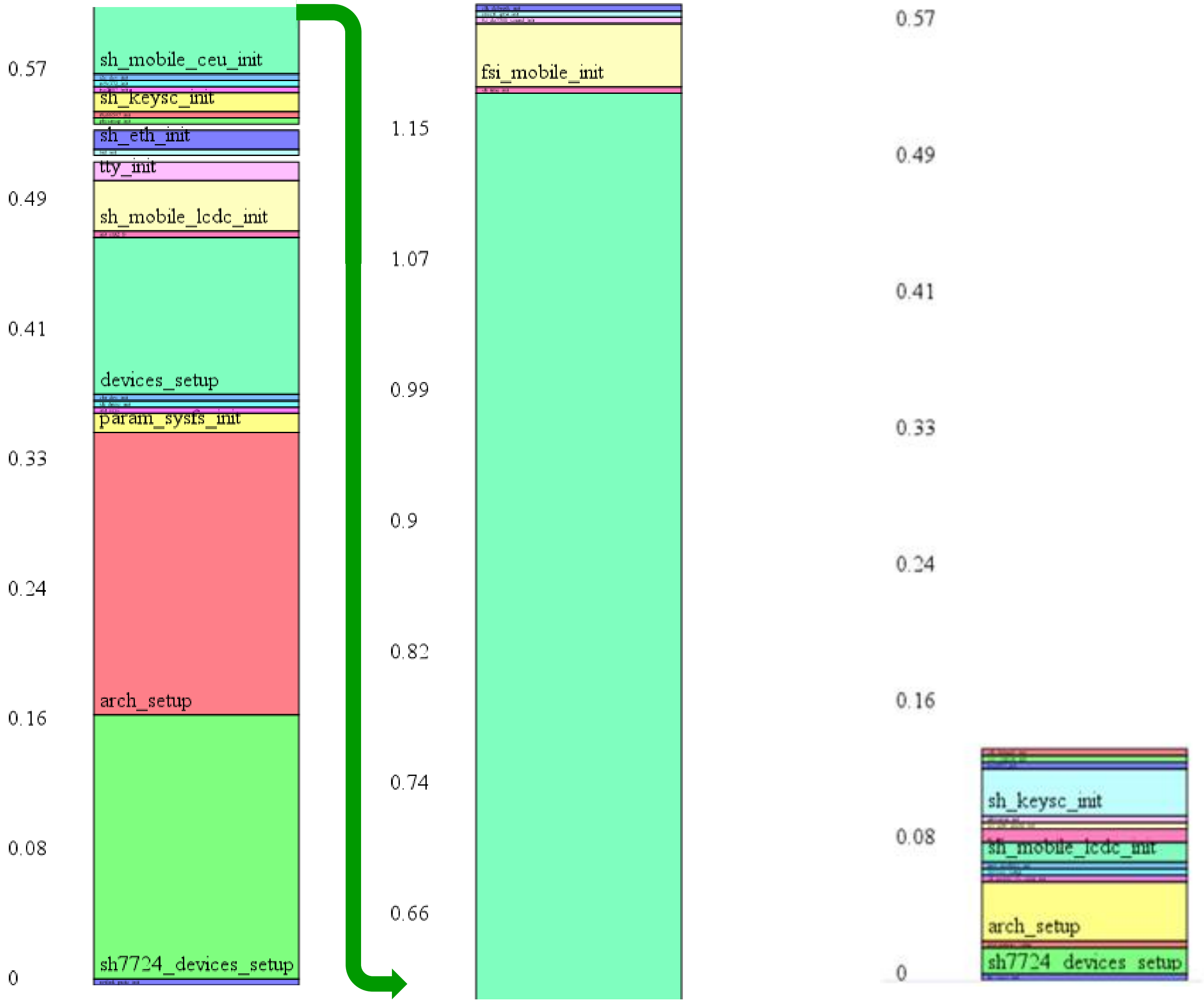
Functionality Re-ordering

- Defer networking initialization – 166 kB – 20 ms

Reduction: 1301 ms -> 113 ms (91%)

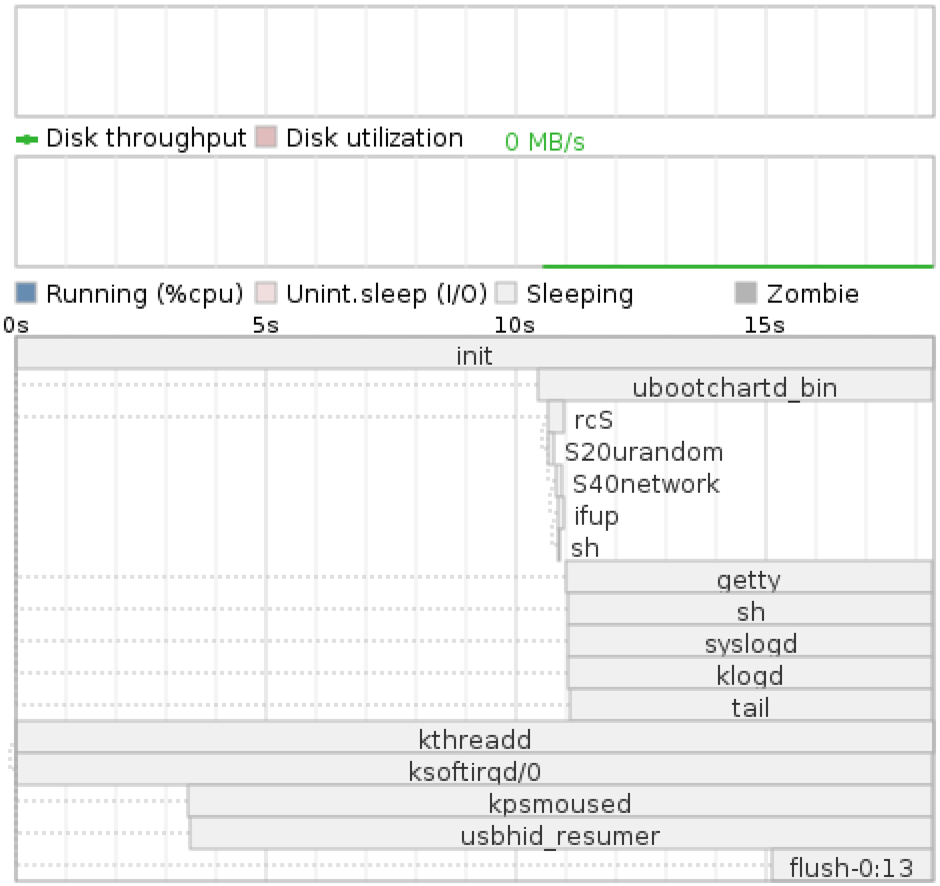
Linux Kernel (Before and After)

Delivering Software Innovation



Userspace (Mount and Init scripts)

Delivering Software Innovation



Example of a bootchart graphic

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

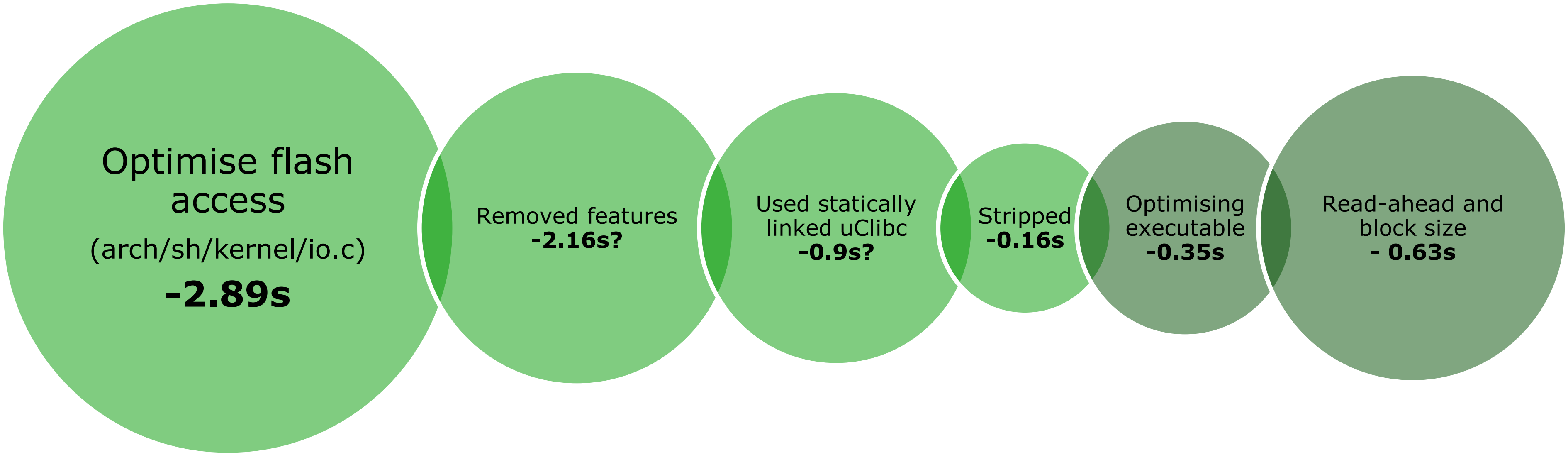
Functionality Re-ordering

- Start QT then later start video

Reduction: 8130 ms > 64 ms (99%)

QT

- 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:



Measurements show incremental effects against original binary (from left to right)

Why does QT take so long to start?

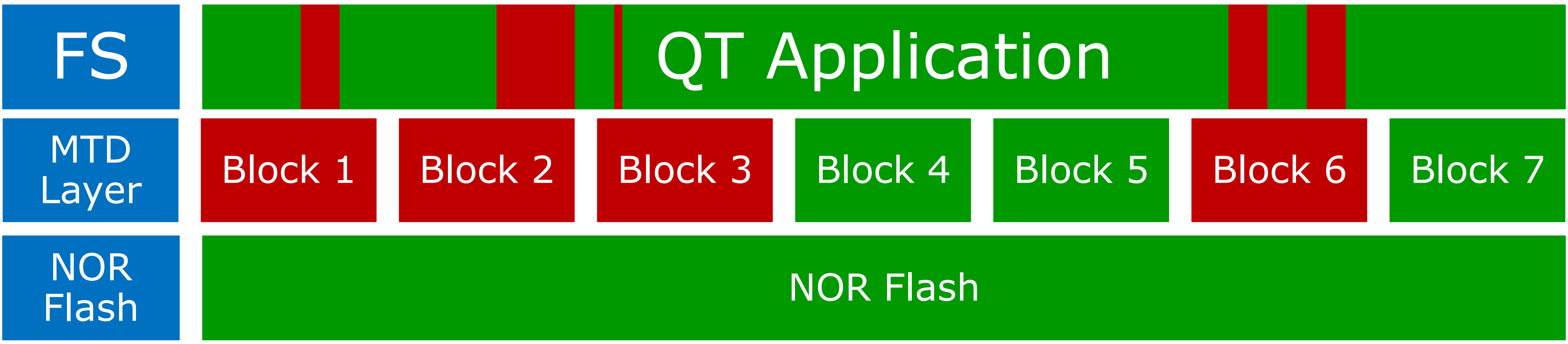
Delivering Software Innovation

- 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

Delivering Software Innovation

- 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



Function Reordering

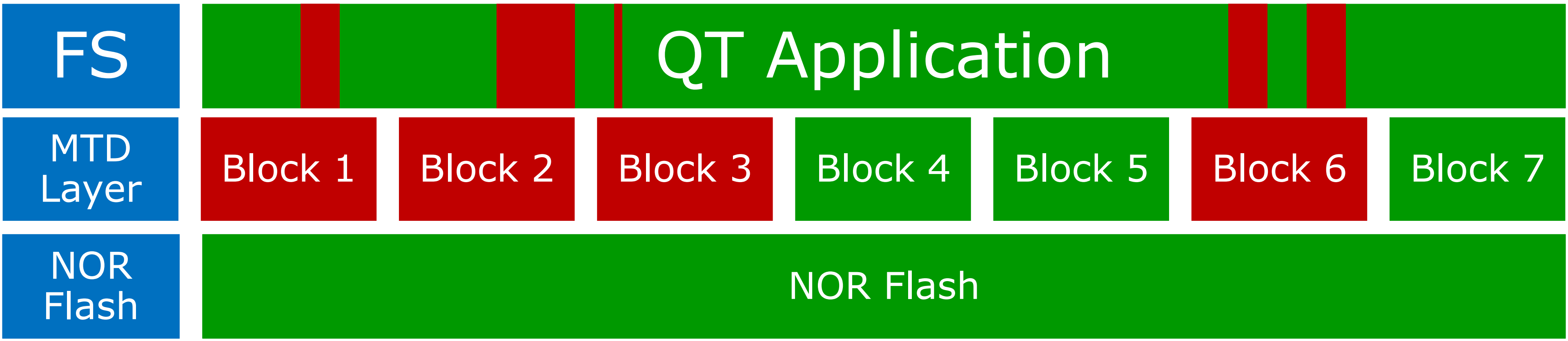
Delivering Software Innovation

- 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_enter (….)
  - 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)

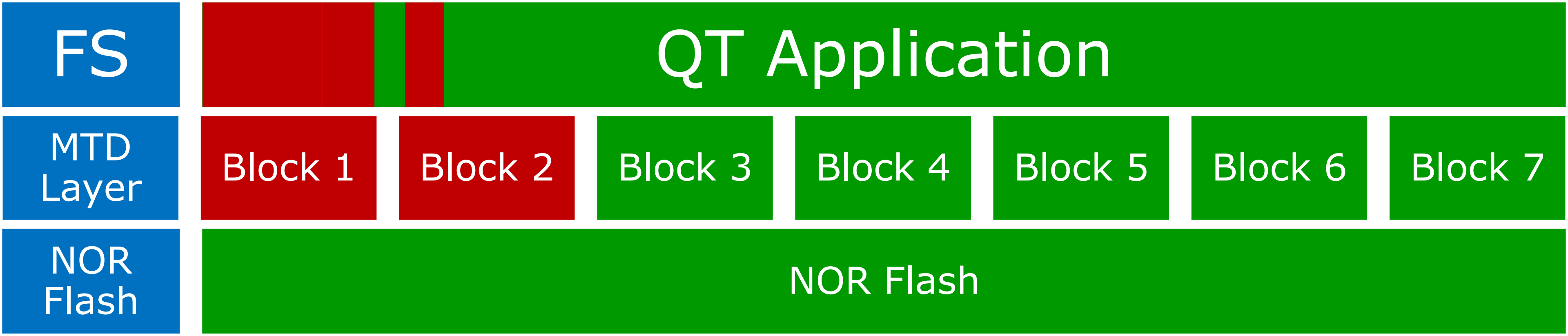
Function Reordering and Block Sizes

Delivering Software Innovation

- Before:



- After:



Essential tools

Delivering Software Innovation

- 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
  - http://code.google.com/p/ubootchart/
  - http://www.bootchart.org/
- These are just some of the many tools available

Case Study: Before and After

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

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

Delivering Software Innovation

Reducing cold boot time is a 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 printks)*
  - 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’