Linux FastBoot

Reducing Embedded Linux Boot Times

Embedded World Conference 2012

Michael Röder
Future Electronics Deutschland GmbH

Detlev Zundel
DENX Software Engineering GmbH
Agenda

- Optimization Basics and Principles
- Boot Process Analysis and Profiling Techniques
- Optimization of
  - Kernel-Space
  - User-Space
- U-Boot
  - History and Introduction
  - Using U-Boot's SPL for Fastboot
System Boot Process

- Cold-Start-Time := Time from Power-On to First-Available Use

- Various stages involved:
  - hardware-setup (clocks, memories, initialization)
  - boot loader (device initialization, kernel decompression)
  - kernel init
  - user init / application start time

- Time wasted due to
  - probing (unneeded flexibility)
  - redundant tasks
  - unnecessary tasks
  - debugging functionality (keep a separate setup for debugging)
Optimization Basics

- Analysis before Optimization
  
  “Premature optimization is the root of all evil. [...] A good programmer [...] will be wise to look carefully at the critical code; but only after that code has been identified:”

  (Donald Knuth)

- Take a step-by-step approach, verify results and detect possible undesired implications after each step

- Optimize big chunks first:
  
  \[ t_1 = 8s; \quad t_2 = 2s; \quad \text{speed-up by factor } 4 \text{ of } t_1 = 1/4 \times 8s = 2s \quad \Rightarrow \text{total speedup: } 150\%. \]
  
  \[ \text{speed-up by factor } 4 \text{ of } t_2 = 1/4 \times 2s = 0.5s \quad \Rightarrow \text{total speedup: } 17\%. \]
Understanding the Application

- First-Available-Use
  - the point of time in which the system „feels“ ready to the user => is in the eye of the beholder
  - partially available system is often enough for first steps
  - less urgent tasks can be postponed or partial unavailability be hidden from user

- Understanding APPLICATION (<= Realization) is critical
  - when do resources / functionality really have to be available?
  - dependencies on each other
  - BDD-like dependency tree shows potential for removal, parallelization and deferring (see next slide!)

- No rules-of-thumb, each application is different!!!
Optimization Principles

(1) Don't do it at all
   - leave out all unnecessary functionality
   - optimize for the current specification, not for future plans, ideas, extendability, …

(2) Do it faster
   - hardware is known => remove probing, scanning
   - use hardware specific compiler optimizations
   - remove unneeded functionality and flexibility
   - make the common case fast

(3) Do it in parallel (for independent tasks)
   - dependency diagram (previous slide) will show possibilities

(4) Do it later
   - when nobody notices, after First-Available-Use has been reached
Boot Process Analysis and Profiling

- **Host-Based Measurement Methods**
  - no modifications or performance implications on target
  - serial console with time-stamping terminal software
    - time_log, grabserial or RealTerm
    - use keywords to trigger / reset time upon events of interest
  - GPIO-toggling based methods

- **Target-Based Methods**
  - enable kernel timestamps (printk.time=1)
  - module_init_call tracing (initcall_debug)
    - use bootgraph.pl on host to generate bootgraph
    - great to verify / document improvements
Kernel Optimization

- Basis for Analysis: graph generated by bootgraph.pl
- Kernel Configuration
  - remove all debugging functionality
  - remove unnecessary device drivers
  - remove device drivers only needed after First-Available-Use (and load those later in the background using modprobe)
- Driver Configuration
  - preset information and remove probing wherever possible
  - re-use information from the bootloader (FDT, boot parameters)
- analyze driver init routine for long delay loops
  - call those init routines earlier (e.g. from the boot loader)
  - arrange such init routines in parallel
User-Space Optimization

- Combine all init scripts into a single one
  - remove unused and unrequired services
  - optimize remaining services
    - remove sanity checks, log-file creation
    - preset information wherever possible to avoid searching/probing
  - move everything not necessary for First-Available-Use to a separate script and start later (from application/cron)

- Avoid using udev
  - use mknod or a copy of the complete /dev tree instead
  - if hotplugging is required, enable devtmpfs

- For profiling use `echo "tag" > /dev/kmsg`
  (with timestamping serial console software)
U-Boot

- Started as a simple boot loader with network support for PowerPC
- Grew into multi-platform boot loader, supporting 14 architectures
- Wide support for mass storage, i.e. NOR, NAND, SPI-NOR, MMC, PATA, SATA
- Supports advanced peripherals like PCI, USB, etc.
- Powerful scripting capabilities
- Also used for hardware bring-up
- Can adapt to software development phases
U-Boot SPL

- „Classic“ CPUs start execution by fetching instructions directly over the address and data buses, i.e. ROM or NOR-Flash

- Modern storage devices cannot be attached directly and need special access methods, i.e. NAND is attached serially and needs page-wise accesses

- A CPU booting from one of these media can usually only load and start a single block of storage

- SPL of U-Boot was designed to re-use existing drivers and be such a small „Secondary Program Loader“
Using SPL for FastBoot

- Flexibility of U-Boot shows in memory footprint, several 100 kB are no exception

- Loading U-Boot itself becomes significant for execution time

- SPL framework allows to split U-Boot into
  - minimum necessary part to access storage  - and -
  - „the rest“

- This „rest“ can be „just another payload“, comparable to the Linux kernel

- Two scenarios:
  - „Fast path“: SPL loads and then starts Linux kernel
  - „Development or maintenance mode“: SPL loads U-Boot
Building SPL

- Mostly manual process comparable to normal "U-Boot configuration" through header file

- If SPL is supported on target architecture:
  - start with "#define CONFIG_SPL"
  - add necessary drivers, i.e. "CONFIG_SPL_NAND_SUPPORT", etc.
  - fill in blanks and control logic, i.e. payload switching and loading (e.g. "spl_board_prepare_for_linux()")

- Resulting SPL is only several kB
Linux as Payload

- Patches only entering mainline, so the design is still in flux and may change in the future

- Usually U-Boots fixes up parameters passed to the Linux kernel, i.e. the flat device tree

- `#define CONFIG_CMD_SPL` compiles U-Boot command capable of capturing the result of this pre-processing

- This parameter block is „frozen“ in a live system using regular U-Boot and is then used as an opaque block by SPL loader
Putting it all together

- In a properly configured system this is the fast path:
  - CPU loads SPL
  - SPL loads Linux kernel and parameter block
  - Linux boots

- When defined criteria holds, this is the development path:
  - CPU loads SPL
  - SPL loads U-Boot as payload
  - U-Boot command line starts
  - U-Boot can boot Linux as regular
Summary

- Boot Time Optimization is not...
  - an automatic (or automatable) process
  - black magic or an “art”

- Road to Success:
  - Thorough Analysis:
    - understand the desired application (“Specification”)
    - analyze existing implementation / system
  - Attentive Optimization
    - remove unnecessary parts
    - optimize the rest, to the extent that the specification permits
    - start with the big and easy parts
  - Enable Re-Use
    - document and verify all changes made during the optimization process, especially possible implications on expected functionality
    - document successes and failures of evaluated optimization attempts, take the same route for similar projects and hardware
THANK YOU