Cheap Linux boot time reduction techniques

Michael Opdenacker

Genivi All Members Meeting
3-6 May, 2011
Dublin, Ireland

Michael Opdenacker
Linaro Community Manager
http://linaro.org
michael.opdenacker@linaro.org

Reducing boot time

Why trying to reduce boot time?

To achieve better user perception

Traditional solutions

Expose the user to relativistic acceleration

$$\tau = t\sqrt{1 - (v^2/c^2)}$$

- Major drawback: the user gets to far from the device to see it boot faster.

Time travel

- Drawback: the user gets 2 devices in his hands for a certain amount of time.

Distract the user

Make the boot process faster

Alternative solutions

- Suspend to RAM and resume
  Used in Android phones. Not acceptable for devices which can stay idle for a long time (e.g. digital cameras)
- Hibernate to flash and resume
  Used in SONY digital cameras (booting in about 1s). Used in credit card payment terminals (our customer).
  Requires sufficient amount of RAM.

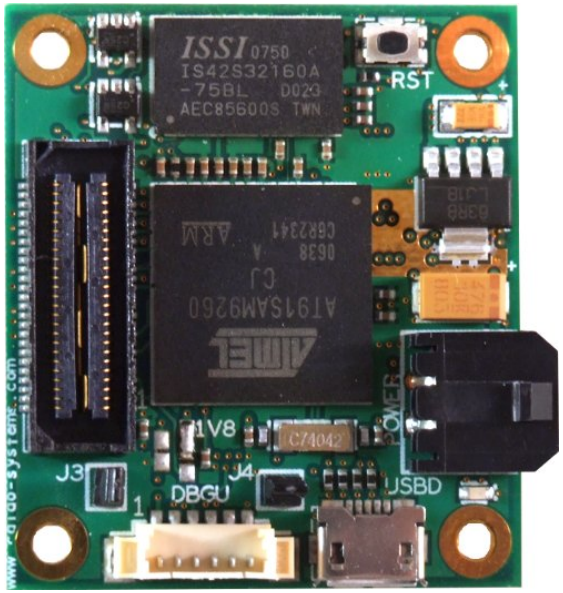
We will focus on reducing “cold” boot time, from power on to the execution of the system application.

The techniques shown should be applicable to all kinds of systems and distributions.

Our test system

Hardware

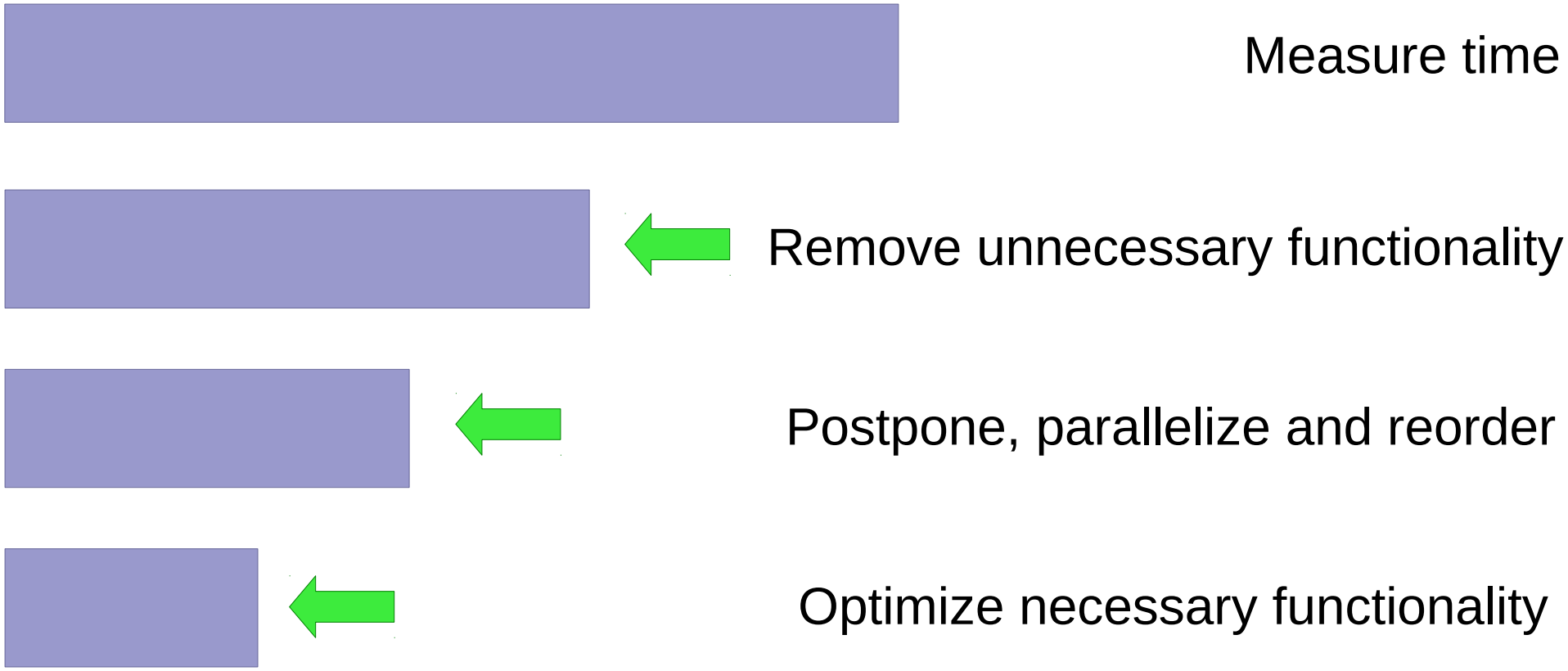
- TNY-A9260 board from CALAO Systems
- AT91SAM9260 CPU at 180 MHz
- RAM: 64 MB, NAND flash: 256 MB
- Serial port
- USB device port (used for networking)



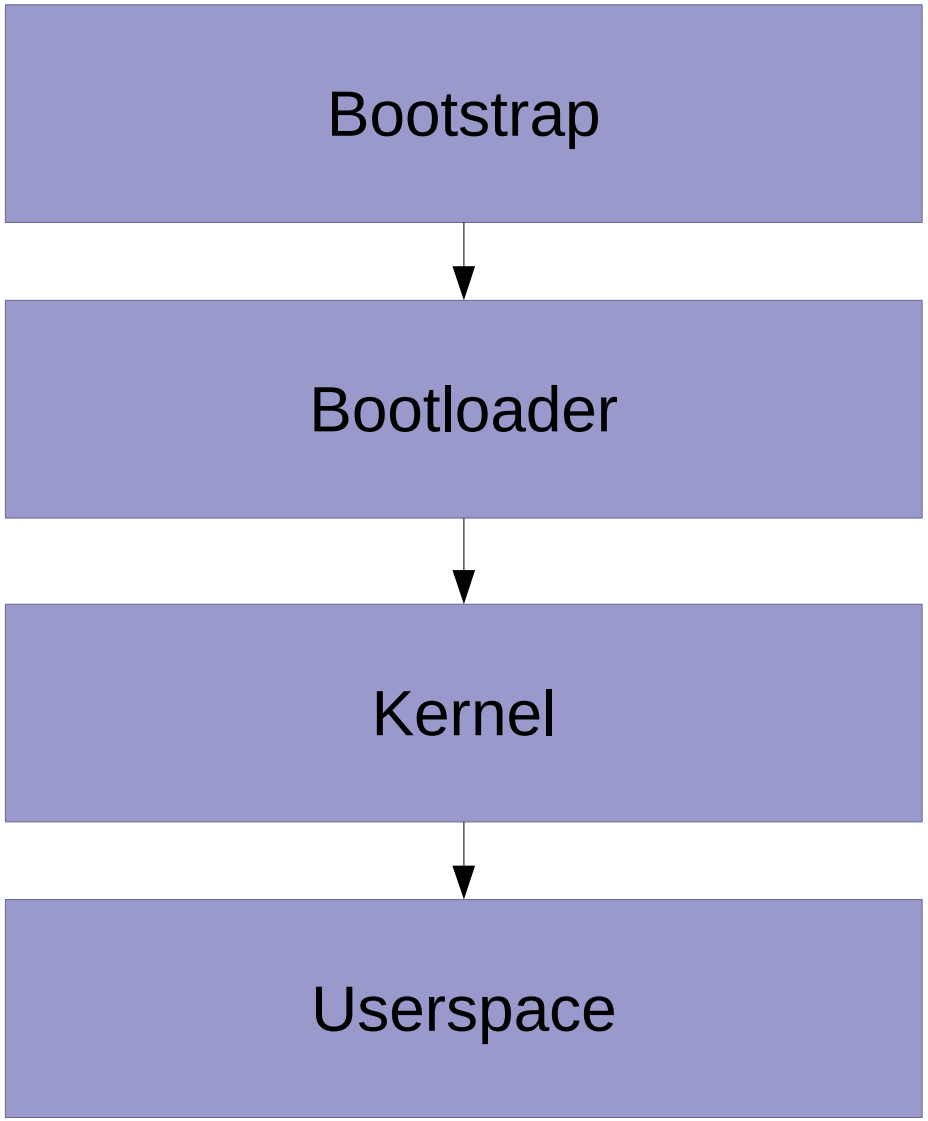
Software

- Simple system built with BusyBox
- Mounting a JFFS2 partition with JPG photos on it (204 MB)
- Starting a BusyBox web server to view and upload photos
- Initial boot time: 37.75 s

Methodology



Boot time components



Bootstrap and bootloader time

Measure time - Grabserial

- From Tim Bird
  http://elinux.org/Grabserial
- A simple script to add timestamps to messages coming from a serial console.
- Key advantage: starts counting very early (bootstrap and bootloader)
- Another advantage: no overhead on the target, because run on the host machine.

Bootloader - Unnecessary functionality

U-Boot is full of features for development and debugging

- U-Boot boot delay
- Recompile U-boot without Ethernet, USB, filesystems...
- You can even disable the console.

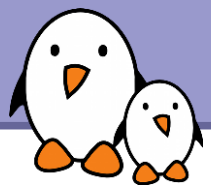
You could even switch to a simpler boot loader (more expensive)

- Qi bootloader from OpenMoko
  http://gitorious.org/+0xlab/0xlab-bootloader/qi-bootloader
- BareBox supports an increasing number of boards
  http://barebox.org

Optimize functionality

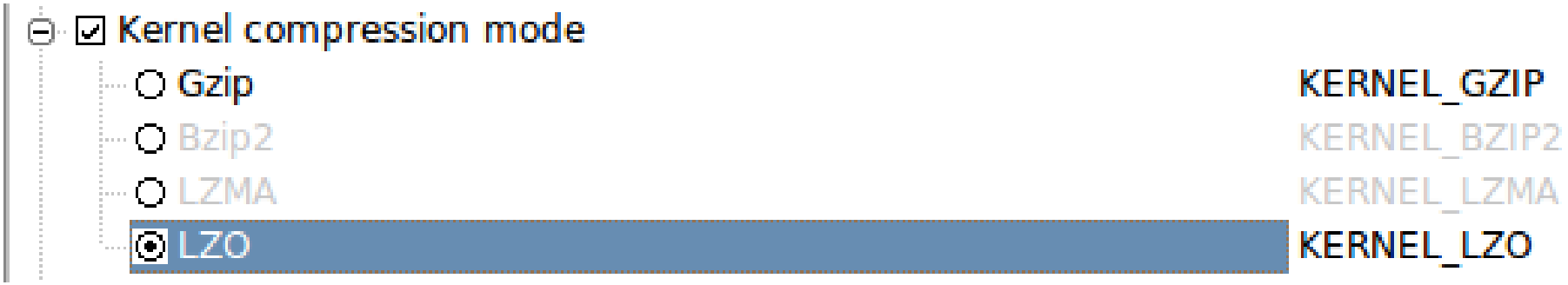
Accelerate kernel copy to RAM

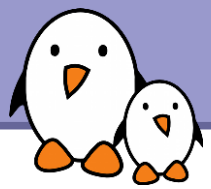
- Rebuild the kernel with `CONFIG_CC_OPTIMIZE_FOR_SIZE`
- Rebuild the kernel without unneeded drivers and features:
  A smaller kernel is faster to copy. See our later slide about kernel size reduction.
- Depending on flash read throughput and CPU performance, choose between various kernel compression schemes:
  - Gzip
  - Bzip2
  - LZMA
  - XZ (improved LZMA compression for executable code)
  - No compression



LZO kernel decompression

- LZO is a compression algorithm that is much faster than Gzip and other compressors, at the cost of a slightly degraded compression ratio (+10%)
- Albin Tonnerre from Bootlin added support for LZO compressed kernels. See http://lwn.net/Articles/350985/





Kernel decompression benchmarks

	Gzip	LZO	Uncompressed
Kernel size	1.33Mb	1.45Mb	2.45Mb
Bootloader + kernel load time	0.30s	0.33s	0.60s
Early kernel init time	0.52s	0.33s	0.02s
Total time	0.82s	**0.66s**	**0.62s**

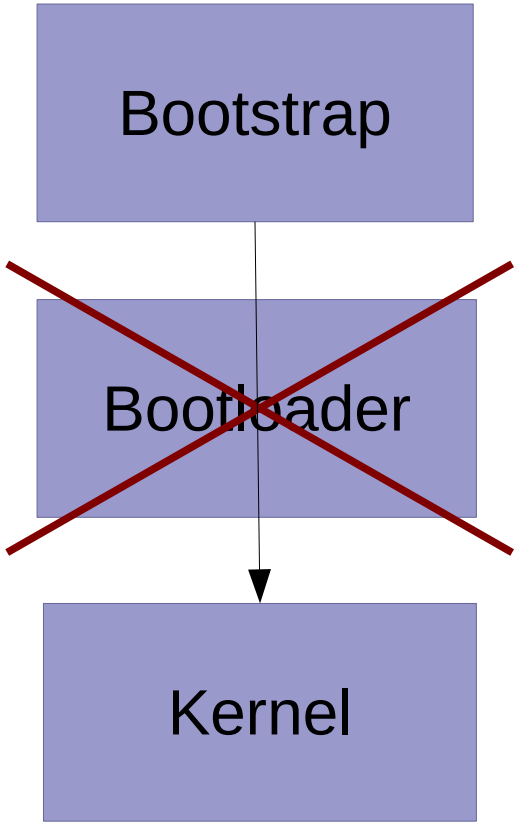
- Bzip2, LZMA and XZ are not tested here but are slower than Gzip and LZO
- See https://bootlin.com/blog/lzo-kernel-compression/

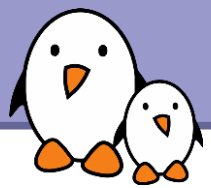
Skip the bootloader (1)

In our AT91 case

- Idea: make a slight change to the bootstrap code to directly load and execute the Linux kernel image instead of the U-boot one.
- Rather straightforward when boot U-boot and the kernel are loaded from NAND flash.
- Requires to hardcode the kernel command line in the kernel image (`CONFIG_CMDLINE`)
- Requires more development work when U-boot is loaded from a different type of storage
- Time savings: about 2 s

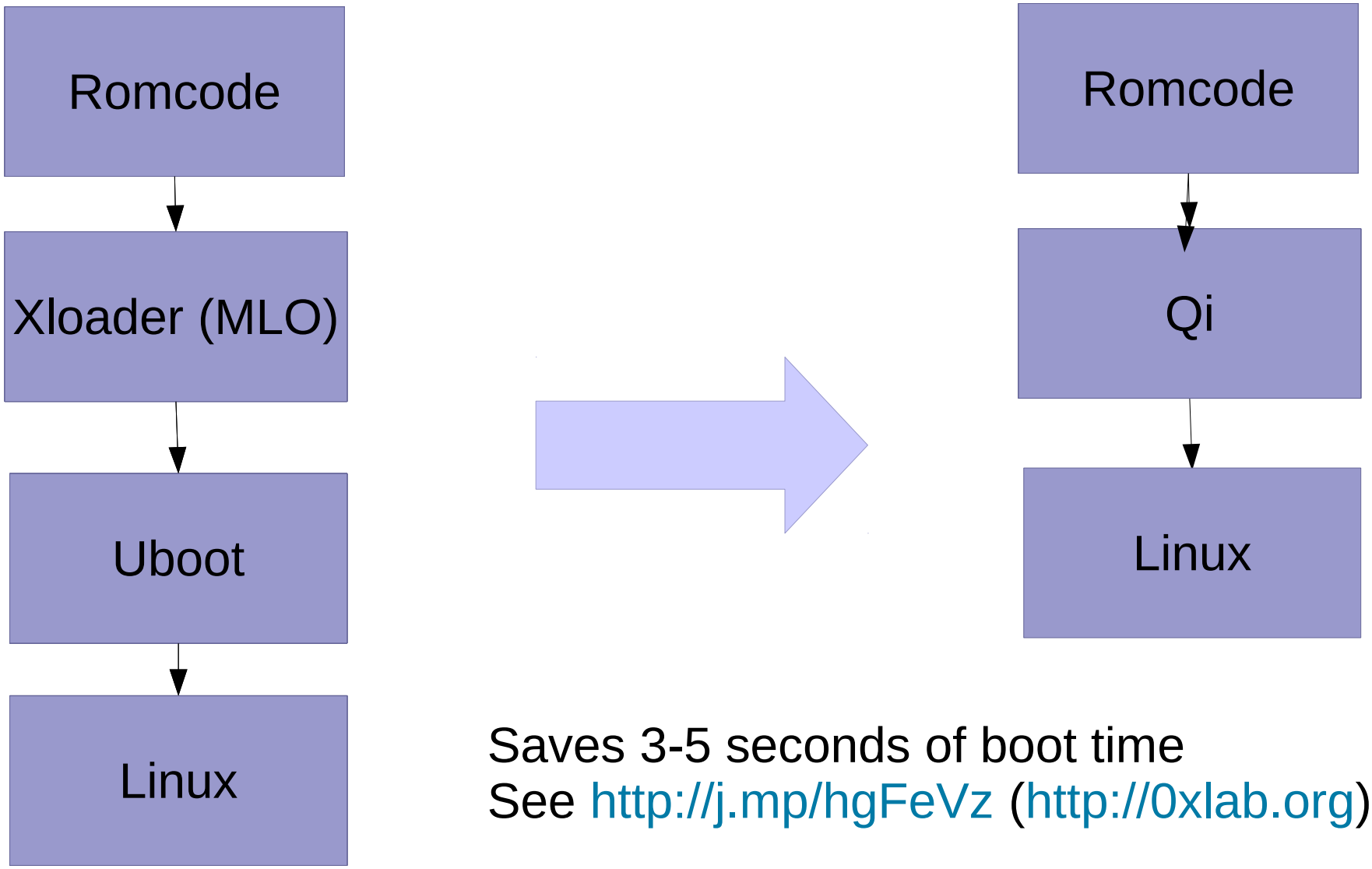
https://bootlin.com/blog/at91bootstrap-linux/





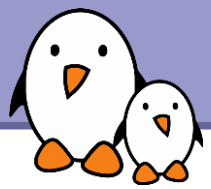
Skip the bootloader (2)

TI OMAP case



Saves 3-5 seconds of boot time
See http://j.mp/hgFeVz (http://0xlab.org)

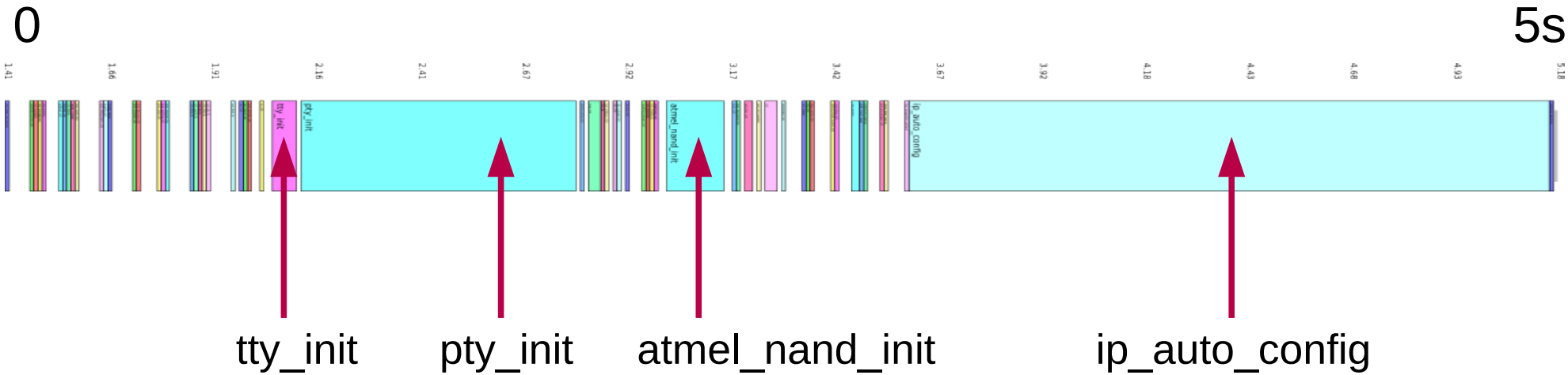
Kernel boot time



Measure time: boot tracer

`CONFIG_BOOT_TRACER` in kernel configuration

- Introduced in Linux 2.6.28
  Based on the ftrace tracing infrastructure
- Allows to record the timings of initcalls
- Boot with the `initcall_debug` and `printk.time=1` parameters, run `dmesg > boot.log` and on your workstation, run `cat boot.log | perl scripts/bootgraph.pl > boot.svg` to generate a graphical representation



Kernel - Remove unnecessary functionality

- Make sure you have no unused kernel drivers
  If devices are not in the critical boot path, load their drivers later.
- Also remove unused networking protocols, filesystems, debugging features...
- If you have a dedicated system, you can even disable standard kernel features with `CONFIG_EXPERT`
  (previously `CONFIG_EMBEDDED`)
- If possible, disable support for loadable kernel modules and make all your drivers static (smaller kernel once more)

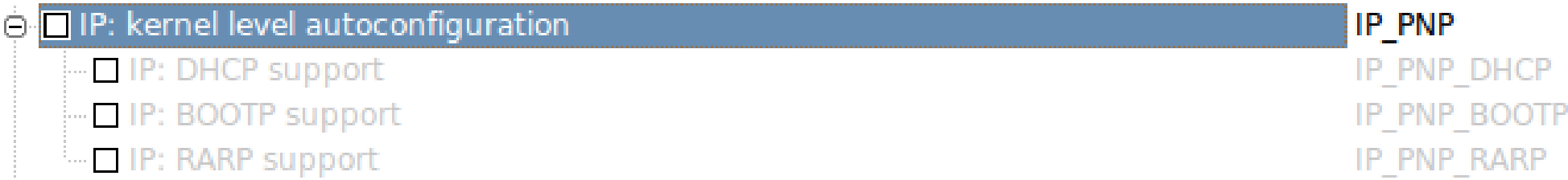
Removing functionality: console output

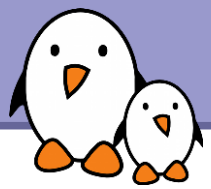
- The output of kernel bootup messages to the console takes time! Even worse: scrolling up in framebuffer consoles! Console output not needed in production systems.
- Console output can be disabled with the `quiet` argument in the Linux kernel command line (bootloader settings)
- Example:
  `root=/dev/ram0 rw init=/startup.sh quiet`
- You can still see the messages through the `dmesg` command.

See http://elinux.org/Disable_Console

Optimizing - Disable IP auto config

- Stopped initializing the IP address on the kernel command line (old remains from NFS booting, was convenient not to hardcode the IP address in the root filesystem.)
- Instead, did it in the `/etc/init.d/rcS` script.
- This saved 1.56 s!
- You will save even more if you had other related options in your kernel (DHCP, BOOP, RARP)





Optimizing - Reducing the number of PTYs

- PTYs are needed for remote terminals (through SSH)
  They are not needed in our dedicated system!
- The number of PTYs can be reduced through the `CONFIG_LEGACY_PTY_COUNT` kernel parameter.
  If this number is set to 4, we save 0.63 s.
- As we're not using PTYs at all in our production system, we disabled them with completely with `CONFIG_LEGACY_PTYS`.
  We saved 0.64 s.
- Note that this can also be achieved without recompiling the kernel, using the `pty.legacy_count` kernel parameter.

Optimizing - Skip probes

- The kernel and drivers probe the hardware at each boot.
  But the hardware is always the same!
- Example: delay loop calibration:
  - At each boot, the Linux kernel calibrates a delay loop (for the `udelay` function). This measures a `loops_per_jiffy` (`lpj`) value.
  - You just need to measure this once! Find the `lpj` value in kernel boot messages. Example:

```
Calibrating delay loop... 99.73 BogoMIPS (lpj=498688)
```

  - At the next boots, start Linux with the below option:
    `lpj=<value>`
  - It saved us 0.18 s
- Check kernel drivers for probing parameters

Optimizing filesystem performance

- The time to mount the root filesystem is a major component of kernel boot time.
- JFFS2 example:
  `CONFIG_JFFS2_SUMMARY` dramatically reduces mount time. No longer needed to scan the whole filesystem at mount time, because collected information is now stored in flash.

  Switching this on saved 27.86 s!
- Very cheap to switch to other filesystems.
  See our benchmarks and presentation on
  http://elinux.org/Flash_Filesystem_Benchmarks

Initramfs: the silver bullet

- Linux can boot on an intermediate root filesystem in RAM, called the initramfs. Its contents can be included in the kernel image.
- Using the file cache, this filesystem doesn't need any drivers to work (no filesystem driver, no disk driver). Hence, it can be accessed very early in the kernel boot process.
- This allows to be in userspace within a few hundred milliseconds after power on.
- System builders use it to show very early signs of life, such as a splashscreen. Even if the boot process isn't complete yet, this definitely helps with user perception.
- Use LZO Initramfs compression to save time: (`INITRAMFS_COMPRESSION_LZO`)

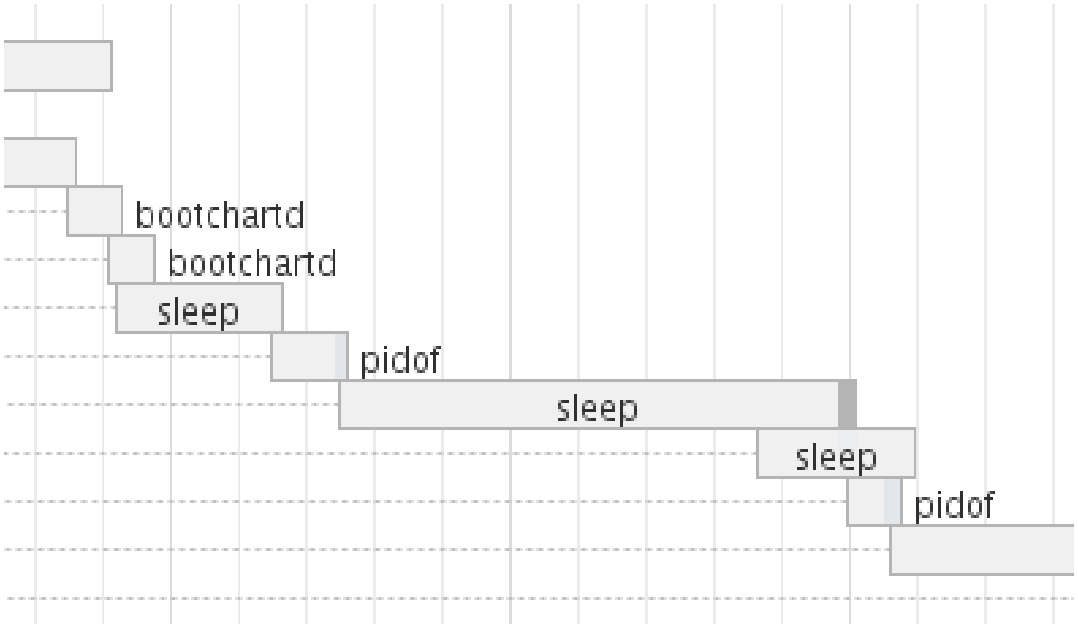
Other kernel optimizations

- NAND: just check for bad blocks once
  Atmel: see http://patchwork.ozlabs.org/patch/27652/
- Fast boot, asynchronous initcalls in drivers
  http://lwn.net/Articles/314808/
  Mainlined, but API still used by very few drivers.
  Mostly useful when your CPU has idle time in the boot process.
- Use deferred initcalls
  See http://elinux.org/Deferred_Initcalls

Userspace boot time

User space: measure time

- If you are using Grabserial, you can still send message to the kernel console from your applications, by writing to `/dev/kmsg`.
- Use utilities to track processes run in the boot sequence: http://elinux.org/Bootchart



Measure time: strace

System call tracer
http://sourceforge.net/projects/strace/

- Mainly useful for your main application
- Can be built by your cross-compiling toolchain generator (crosstool-ng for example)
- Allows to see what any of your processes is doing: accessing files, allocating memory...
  Very useful to detect waste of time.
- Usage:
  `strace <command>` (starting a new process)
  `strace -p<pid>` (tracing an existing process)

See `man strace` for details.

strace example output

```
> strace cat Makefile
execve("/bin/cat", ["cat", "Makefile"], [/* 38 vars */]) = 0
brk(0)                                  = 0x98b4000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7f85000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=111585, ...}) = 0
mmap2(NULL, 111585, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f69000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320h\
1\0004\0\0\0\344"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1442180, ...}) = 0
mmap2(NULL, 1451632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb7e06000
mprotect(0xb7f62000, 4096, PROT_NONE)   = 0
mmap2(0xb7f63000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x15c) = 0xb7f63000
mmap2(0xb7f66000, 9840, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0xb7f66000
close(3)                                = 0
```

measure time: ltrace

A tool to trace library calls used by a program
and all the signals it receives

- Very useful complement to strace,
  which shows only system calls.
- Of course, works even if you don't have the sources
- Allows to filter library calls with regular expressions,
  or just by a list of function names.
- Manual page: http://linux.die.net/man/1/ltrace

See http://en.wikipedia.org/wiki/Ltrace for details

ltrace example output

```
ltrace nedit index.html
sscanf(0x8274af1, 0x8132618, 0x8248640, 0xbfaadfe8, 0) = 1
sprintf("const 0", "const %d", 0)                      = 7
strcmp("startScan", "const 0")                         = 1
strcmp("ScanDistance", "const 0")                      = -1
strcmp("const 200", "const 0")                         = 1
strcmp("$list_dialog_button", "const 0")               = -1
strcmp("$shell_cmd_status", "const 0")                 = -1
strcmp("$read_status", "const 0")                      = -1
strcmp("$search_end", "const 0")                       = -1
strcmp("$string_dialog_button", "const 0")             = -1
strcmp("$rangeset_list", "const 0")                    = -1
strcmp("$calltip_ID", "const 0")                       = -1
...
```

ltrace summary

Example summary at the end of the `ltrace` output (`-c` option)

```
Process 17019 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.000050          50         1           set_thread_area
  0.00    0.000000           0        48           read
  0.00    0.000000           0        44           write
  0.00    0.000000           0        80        63 open
  0.00    0.000000           0        19           close
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         2 access
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         1           mprotect
  0.00    0.000000           0        19           mmap2
  0.00    0.000000           0        50        46 stat64
  0.00    0.000000           0        18           fstat64
------ ----------- ----------- --------- --------- ----------------
100.00    0.000050                   288       111 total
```

Measure time: use a profiler

Profile your application to detect unnecessary activity or performance issues.

- For example, a profiler can tell you in which functions most of the time is spent.
- Valgrind (http://valgrind.org/) is the most popular profiler
  - Now available for the `arm` architecture, thanks to Linaro (armv7 only: Cortex A8, A9 and A5)
  - Complete suite of profiling tools, in particular:
    Cachegrind: sources of cache misses and function statistics.
    Massif: sources of memory allocation.

Optimize and remove services

If you are using a distribution or an automatically generated root filesystem

- Remove services you don't need (ssh...), or start them later.
- Start your services directly from a single startup script.
  This eliminates multiple calls to `/bin/sh`.
- Remove udev (or mdev) if you just need them for device files.
  Use devtmpfs (`CONFIG_DEVTMPFS`) instead,
  automatically managed by the kernel, and cheaper.
- This saves tens of seconds with root filesystems generated with OpenEmbedded (for example).

Shells: reducing forking

- `fork` / `exec` system calls are very expensive.
  Because of this, calls to executables from shells are slow.
- Even executing `echo` in busybox shells results in a `fork` syscall!
- Select `Shells -> Standalone shell` in busybox configuration to make the busybox shell call applets whenever possible.
- Pipes and back-quotes are also implemented by `fork` / `exec`.
  You can reduce their usage in scripts. Example:
  `cat /proc/cpuinfo | grep model`
  Replace it with: `grep model /proc/cpuinfo`

See http://elinux.org/Optimize_RC_Scripts

Optimizing: executables and libraries (1)

- Use statically linked applications
  (less CPU overhead, less libraries to load). At least true for small root filesystems.
- Use a lighter C library reduced to the minimum
  (uClibc or eglibc). Can save up to 1 MB.

C program	*Compiled statically*	
	glibc	*uClibc*
Plain “hello world”	475 K	25 K
Busybox	843 K	311 K

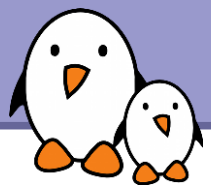
Optimizing: executables and libraries (2)

- Strip your executables and libraries, removing ELF sections only needed for development and debugging.
  `strip` command provided by your toolchain:
  `arm-linux-strip potato`
- `superstrip`:
  http://muppetlabs.com/~breadbox/software/elfkickers.html
  Goes beyond strip and can strip out a few more bits that are not used by Linux to start an executable.

	Hello World	***Busybox***	***Inkscape***
Regular	4691 B	287783 B	11397 KB
stripped	2904 B (-38 %)	230408 B (-19.9 %)	9467 KB (-16.9 %)
sstripped	1392 B (-70 %)	229701 B (-20.2 %)	9436 KB (-17.2 %)

Using processor acceleration instructions

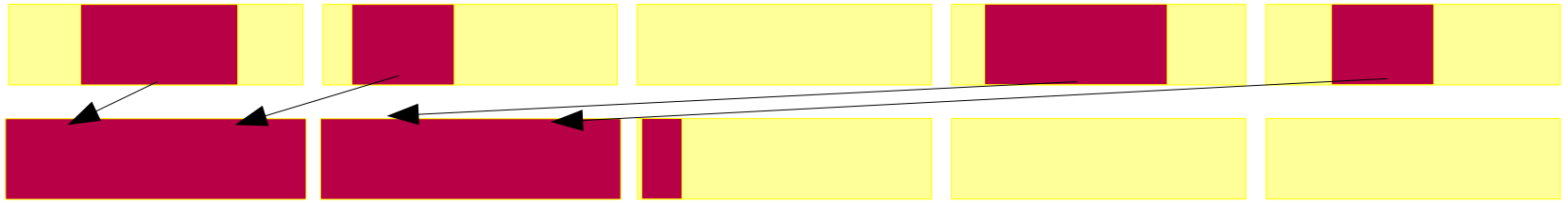
- liboil - http://liboil.freedesktop.org/
  Library of functions optimized for special instructions from several processors (Altivec, MMX, SSE, etc.)
- Mainly functions implementing loops on data arrays: type conversion, copying, simple arithmetics, direct cosine transform, random number generation...
- Transparent: keeps your application portable!
- Linaro has optimized C library functions (memset, memcpy...) for recent arm cores, using NEON instructions. See https://launchpad.net/cortex-strings.



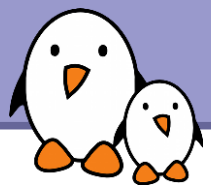
Linker optimizations

Group application code used at startup

- Find the functions called during startup
- Create a custom linker script to put them all together in the same section, using the `-ffunction-sections` gcc option.
- Particularly useful for flash storage with rather big MTD read blocks. As the whole read blocks are read, you end up reading unnecessary data.



See http://j.mp/m4d1Q6



Results

- This test case:
  Initial boot time: 38 s
  Final boot time: approximately 4 s
- Customer audit 1: AMD Geode board with X graphics
  From 32 to 10 seconds in only 3 days
- Customer audit 2: AT91SAM9263 based system
  From 32 to 7 seconds in only 2 days
- Lots of techniques not applied yet.
  I just wanted to show how much you can achieve with limited time and effort.
  With more effort, it is possible to boot any system within 1-5 seconds after power-on.

Conclusion

These optimizations are cheap!

- Find the low hanging fruit
  and divide your boot-time by 2 or 3 in a few days.
- None of them require any re-redesign
  Another team can take care of them, and this can be done very late in product development.
- With the exception of bootstrap tricks, and application fixes, you won't have any extra development to do. You may just have to recompile your bootloader, kernel and root filesystem.
- Quick learning curve: very easy to reuse the same techniques in future products.

Don't tell your boss that just a few days were enough! And with the extra spare time, contribute to community projects ;-)

Thanks!

Questions?

Other techniques?

Slides available on https://bootlin.com/pub/conferences/2011/genivi/

See also http://elinux.org/Boot_Time