Update on boot time reduction techniques, with figures

Michael Opdenacker
Bootlin
michael.opdenacker@bootlin.com

Clipart: `http://openclipart.org/detail/46075/stop-watch-by-klaasvangend`

Michael Opdenacker

- CEO and Embedded Linux engineer at Bootlin
  - Embedded Linux **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux **training**, Linux driver development training and Android system development training, with materials freely available under a Creative Commons license.
  - `http://bootlin.com`
- Conducted several boot time reduction projects, and preparing a workshop on the topic.
- Living in **Orange**, south of France.

About this presentation

- It is based on our new Linux boot time training materials: `http://bootlin.com/doc/training/boot-time`.
- That's where you will find extensive details about Linux boot time reduction methodology and resources.
- Here, we are focusing on
  - New resources
  - Techniques that we hadn't documented yet, and that we used in recent projects.
  - Benchmarks made recently
  - Details that you may have missed
- Thanks to
  - Alexandre Belloni, co-author of this document.
  - Atmel Corporation, for funding the development of the first version of these materials, and for providing boards.

Why reduce boot time?

To make a fortune

- Hi California startup creators!
- Here is an opportunity to make millions and change people's lives:
  - During the BA flight to San Franciso yesterday, they had to reboot the "Highlife Entertainment System". The lady warned that it could take up to 20 minutes.
  - It took 16 minutes to start showing "System being reset, please wait".
  - It was up and running in about 18 minutes.
  - The lady warned: "Please don't touch the screen during the reboot process."

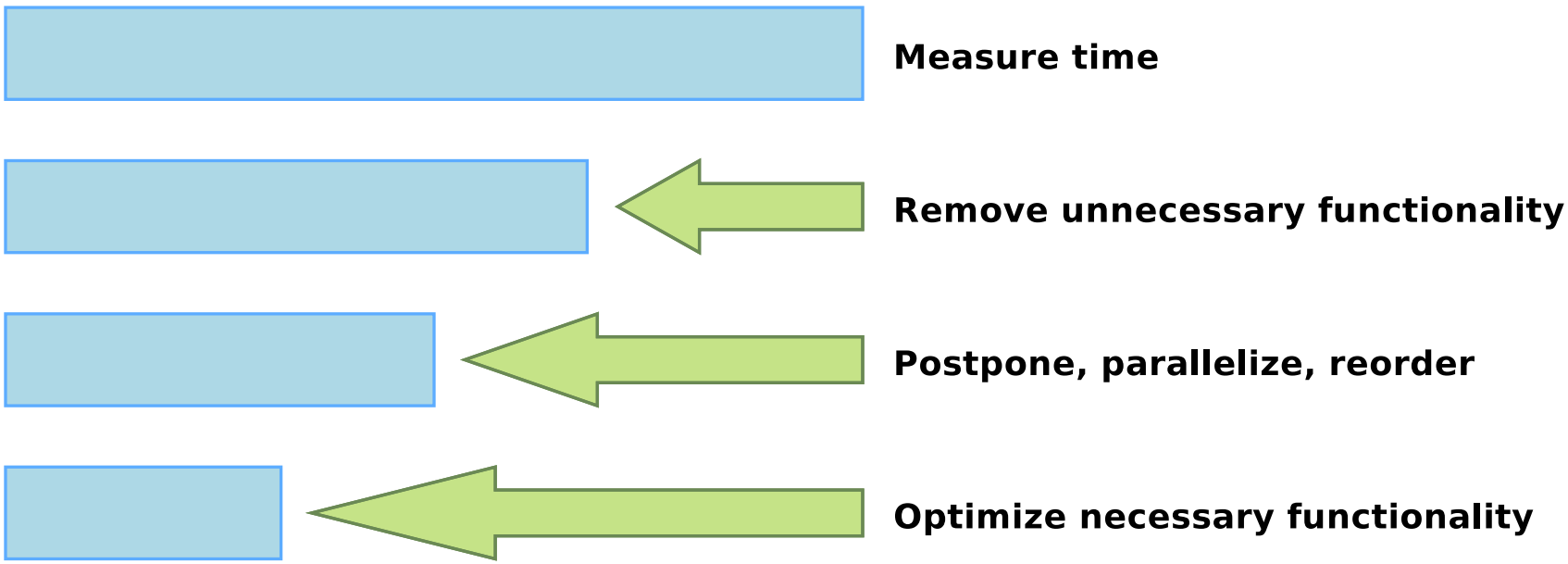
Because you don't want to let...

Because you don't want to let...

Chuck Norris reduce boot time for you!

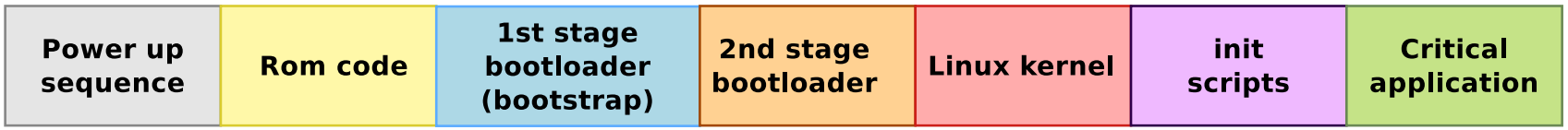
Image credits: `http://commons.wikimedia.org/wiki/File:Guns_%26_Ammo_4.jpg`

Boot time reduction methodology



Boot time components

Generic boot sequence



We are focusing on reducing *cold* boot time, from power on to the critical application.

What to optimize first

Start by optimizing the last steps of the boot process!

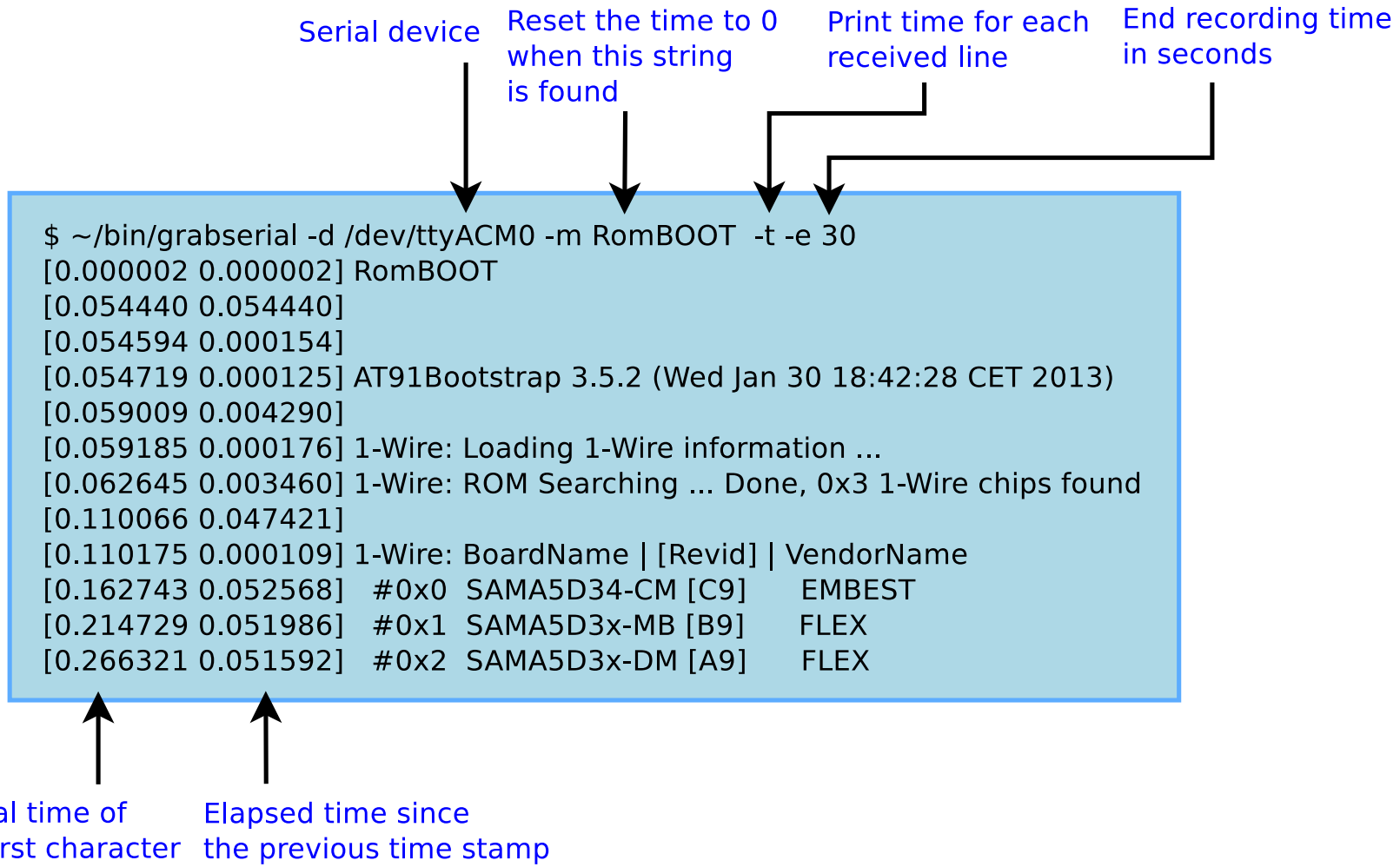
- Don't start by optimizing things that will reduce your ability to make measurements and implement other optimizations.
- Start by optimizing your applications and startup scripts first.
- You can then simplify BusyBox, reducing the number of available commands.
- The next thing to do is simplify and optimize the kernel. This will make you lose debugging and development capabilities, but this is fine as userspace has already been simplified.
- The last thing to do is implement bootloader optimizations, when kernel optimizations are over and when the kernel command line is frozen.

Measuring

grabserial

- From Tim Bird: `http://elinux.org/Grabserial` (Hi Tim!)
- A Python 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.
- Drawbacks: may not be precise enough.
  Can't measure power-up time.

Using grabserial



Caution: `grabserial` shows the arrival time of the **first character** of a line. This doesn't mean that the entire line was received at that time.

Filesystem optimizations

Filesystem impact on performance

Tuning the filesystem is usually one of the first things we work on in boot time projects.

- Different filesystems can have different initialization and mount times. In particular, the type of filesystem for the root filesystem directly impacts boot time.
- Different filesystems can exhibit different read, write and access time performance, according to the type of filesystem activity and to the type of files in the system.
- Fortunately, changing filesystem types is quite cheap, and completely transparent for applications. Just try several filesystem options, as see which one works best for you!

Block filesystems

For block storage (media cards, eMMC...)

- ext4: best for rather big partitions, good read and write performance.
- xfs, jfs, reiserfs: can be good in some read or write scenarii as well.
- btrfs, f2fs: can achieve best read and write performance, taking advantage of the characteristics of flash-based block devices.
- SquashFS: best mount time and read performance, for read-only partitions. Great for root filesystems which can be read-only.

Block filesystem boot benchmarks

Measured on the Atmel SAMA5D3 Xplained board (ARM), Linux 3.10

	ext3	ext4	btrfs	f2fs
Start init	7.878 s	8.039s	7.907s	8.817s

Note that the `rootfstype` kernel command line option also helps. It saves `10 ms` for `ext3` on the same board and kernel (can be even worse if the static kernel supports more filesystems).

Flash filesystems

For raw flash storage

- JFFS2: bad read, write and mount performance. Needs `CONFIG_JFFS2_SUMMARY` to avoid huge mount time.
- YAFFS2: good read, write and mount performance, but no compression. Not in mainline.
- UBIFS: good read and write performance. Good mount performance, but requires *UBI Fastmap* (need Linux 3.7 or later).
- See our flash filesystem benchmarks: `http://elinux.org/Flash_Filesystem_Benchmarks`.

Using UBI Fastmap

- Compile your kernel with `CONFIG_UBI_FASTMAP`
- Boot your system at least once with the `ubi.fm_autoconvert=1` kernel parameter.
- Reboot your system in a clean way
- You can now remove `ubi.fm_autoconvert=1`

UBI Fastmap benchmark

- Measured on the Atmel SAMA5D3 Xplained board (ARM), Linux 3.10
- UBI space: 216 MB
- Root filesystem: 80 MB used (Yocto)
- Average results:

	Attach time	Diff	Total time
Without *UBI Fastmap*	968 ms		
With *UBI Fastmap*	238 ms	-731 ms	-665 ms

- Expect to save more with bigger UBI spaces!

Note: total boot time reduction a bit lower probably because of other kernel threads executing during the attach process.

ubiblock + SquashFS

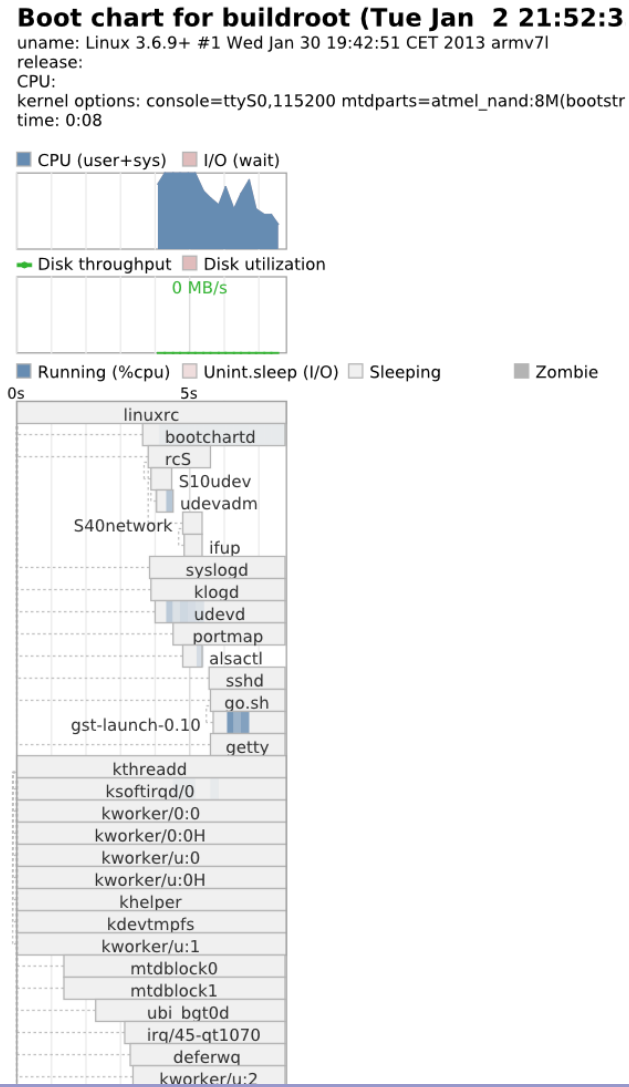
For raw flash storage

- *ubiblock*: read-only block device on top of UBI (`CONFIG_MTD_UBI_BLOCK`). Available in Linux 3.15 (developed on his spare time by Ezequiel Garcia, a Bootlin contractor).
- Allows to put SquashFS on a UBI volume.
- Expecting great boot time and read performance. Great for read-only root filesystems.
- Benchmarks not available yet.

Init scripts

Measuring - bootchart

If you want to have a more detailed look at the userland boot sequence than with `grabserial`, you can use `bootchart`.



Measuring - bootchart

- You can use `bootchartd` from `busybox` (`CONFIG_BOOTCHARTD=y`)
- Boot your board passing `init=/sbin/bootchartd` on your kernel command line
- Copy `/var/log/bootlog.tgz` from your target to your host
- Generate the timechart:

```
cd bootchart-<version>
java -jar bootchart.jar bootlog.tgz
```

`bootchart` is available at `http://www.bootchart.org`

Measuring - systemd

If you are using `systemd` as your `init` program, you can use `systemd-analyze`. See `http://www.freedesktop.org/software/systemd/man/systemd-analyze.html`.

```
$ systemd-analyze blame
  6207ms udev-settle.service
   735ms NetworkManager.service
   642ms avahi-daemon.service
   600ms abrtd.service
   517ms rtkit-daemon.service
   396ms dbus.service
   390ms rpcidmapd.service
   346ms systemd-tmpfiles-setup.service
   316ms cups.service
   310ms console-kit-log-system-start.service
   309ms libvirtd.service
   303ms rpcbind.service
   298ms ksmtuned.service
   281ms rpcgssd.service
   277ms sshd.service
...
```

Optimizing init scripts

- Start all your services directly from a single startup script (e.g. `/etc/init.d/rcS`). This eliminates multiple calls to `/bin/sh`.
- If you need `udev` to manage hotplug events, replace `udev` with BusyBox `mdev`. It is not running as a daemon. It will only be run when hotplug events happen.
- If you just need `udev` to create device files, remove it and use `devtmpfs` (`CONFIG_DEVTMPFS`) instead, automatically managed by the kernel, and cheaper.
- Results: Atmel SAMA5D3x evaluation kit, video player demo: `1.015 s` saved by replacing `udev` by `devtmpfs`.

Reduce forking (1)

- `fork`/`exec` system calls are very expensive. Because of this, calls to executables from shells are slow.
- Even an `echo` in a BusyBox shell results in a `fork` syscall!
- Select `Shells -> Standalone shell` in BusyBox configuration to make the 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`

Reduce forking (2)

Replaced:

```
if [ $(expr match "$(cat /proc/cmdline)" '.* debug.*')\
       -ne 0 -o -f /root/debug ]; then
DEBUG=1
```

By a much cheaper command running only one process:

```
res=`grep " debug" /proc/cmdline`
if [ "$res" -o -f /root/debug ]; then
DEBUG=1
```

This only optimization allowed to save 87 ms on an ARM AT91SAM9263 system (200 MHz)!

Do not compress your initramfs (1)

- If you ship your initramfs inside a compressed kernel image, don't compress it (enable `CONFIG_INITRAMFS_COMPRESSION_NONE`).
- Otherwise, your initramfs data will be compressed twice, and the kernel will be slightly bigger and will take a little more time to uncompress.

Do not compress your initramfs (2)

Tests on Linux 3.13-rc4, measuring the penalty of having a `gzip` compressed initramfs in a `gzip` compressed kernel.

Beagle Bone Black (ARM, TI AM3359, 1 GHz)

Mode	Size	Copy	Uncompress	Total	Diff
No initramfs compression	4308200	451 ms	945 ms	5.516 s	
Initramfs compression	4309112	455 ms	947 ms	5.527 s	+ 11 ms

CALAO USB-A9263 (ARM, Atmel AT91SAM9263, 200 MHz)

Mode	Size	Copy	Uncompress	Total	Diff
No initramfs compression	3016192	4.1047 s	1.737 s	8.795 s	
Initramfs compression	3016928	4.1050 s	1.760 s	8.813 s	+ 18 ms

Quick splashscreen display (1)

Often the first sign of life that you are showing!

- You could use the `fbv` program (`http://freecode.com/projects/fbv`) to display your splashscreen.
- On `armel`, you can just use our statically compiled binary:

  `http://git.bootlin.com/users/michael-opdenacker/static-binaries/tree/fbv`
- However, this is slow:
  878 ms on an Atmel AT91SAM9263 system!

Quick splashscreen display (2)

- To do it faster, you can dump the framebuffer contents:

```
fbv -d 1 /root/logo.bmp
cp /dev/fb0 /root/logo.fb
lzop -9 /root/logo.fb
```

- And then copy it back as early as possible in an initramfs:

```
lzopcat /root/logo.fb.lzo > /dev/fb0
```

Results on an Atmel AT91SAM9263 system:

	`fbv`	plain copy (`dd`)	`lzopcat`
Time	878 ms	54 ms	52.5 ms

`http://bootlin.com/blog/super-fast-linux-splashscreen/`

Applications

Tracing applications

You need ways of tracing your application,
and understand where time is spent:

- `strace`
- `oprofile`
- `perf`

See usage details on our slides:
`http://bootlin.com/doc/training/boot-time`

Kernel optimizations

Measure - Kernel initialization functions

To find out which kernel initialization functions are the longest to execute, add `initcall_debug` to the kernel command line. Here's what you get on the kernel log:

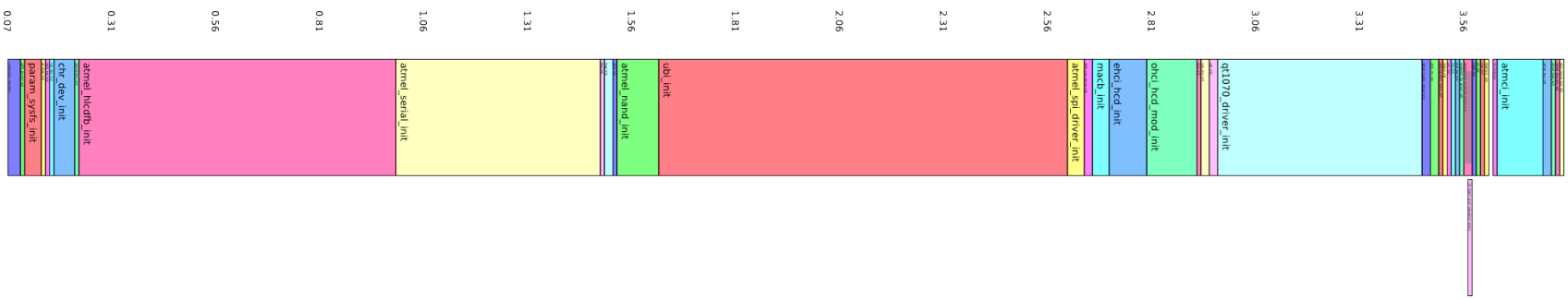
```
...
[    3.750000] calling  ov2640_i2c_driver_init+0x0/0x10 @ 1
[    3.760000] initcall ov2640_i2c_driver_init+0x0/0x10 returned 0 after 544 usecs
[    3.760000] calling  at91sam9x5_video_init+0x0/0x14 @ 1
[    3.760000] at91sam9x5-video f0030340.lcdheo1: video device registered @ 0xe0d3e340, irq = 24
[    3.770000] initcall at91sam9x5_video_init+0x0/0x14 returned 0 after 10388 usecs
[    3.770000] calling  gspca_init+0x0/0x18 @ 1
[    3.770000] gspca_main: v2.14.0 registered
[    3.770000] initcall gspca_init+0x0/0x18 returned 0 after 3966 usecs
...
```

It is probably a good idea to increase the log buffer size with `CONFIG_LOG_BUF_SHIFT` in your kernel configuration. You will also need `CONFIG_PRINTK_TIME` and `CONFIG_KALLSYMS`.

Kernel boot graph

With `initcall_debug`, you can generate a boot graph making it easy to see which kernel initialization functions take most time to execute.

- Copy and paste the console output or the output of the `dmesg` command to a file (let's call it `boot.log`)
- On your workstation, run the `scripts/bootgraph.pl` script in the kernel sources:
  `perl scripts/bootgraph.pl boot.log > boot.svg`
- You can now open the boot graph with a vector graphics editor such as `inkscape`:



Using the kernel boot graph (1)

Start working on the functions consuming most time first. For each function:

- Look for its definition in the kernel source code. You can use LXR (for example `http://lxr.bootlin.com`).
- Remove unnecessary functionality:
  - Look for kernel parameters in C sources and Makefiles, starting with `CONFIG_`. Some settings for such parameters could help to remove code complexity or remove unnecessary features.
  - Find which module (if any) it belongs to. Loading this module could be deferred.

Using the kernel boot graph (2)

- Postpone:
  - Find which module (if any) the function belongs to. Load this module later if possible.
- Optimize necessary functionality:
  - Look for parameters which could be used to reduce probe time, looking for the `module_param` macro.
  - Look for delay loops and calls to functions containing `delay` in their name, which could take more time than needed. You could reduce such delays, and see whether the code still works or not.

Reduce kernel size

To let the kernel load and initialize faster

- Compile everything that is not needed at boot time as a module
  - Results: Atmel SAMA5D3x evaluation kit, video player demo: `950 ms` saved by using modules.
- Remove features not needed in your system: features, drivers, and also debugging functionality.
- Kernel compression: will be done after bootloader optimizations.

Turning off console output

- Console output is actually taking a lot of time (very slow device). Probably not needed in production. Disable it by passing the `quiet` argument on the kernel command line.
- You will still be able to use `dmesg` to get the kernel messages.
- Time between starting the kernel and starting the `init` program, on Atmel SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
Without `quiet`	2.352 s	
With `quiet`	1.285 s	-1.067 s

- Less time will be saved on a reduced kernel, of course.

Preset loops per jiffy

- At each boot, the Linux kernel calibrates a delay loop (for the `udelay` function). This measures a number of loops per jiffy (*lpj*) value. You just need to measure this once! Find the `lpj` value in the kernel boot messages:

```
Calibrating delay loop... 262.96 BogoMIPS (lpj=1314816)
```

- Now, you can add `lpj=<value>` to the kernel command line:

```
Calibrating delay loop (skipped) preset value.. 262.96 BogoMIPS (lpj=1314816)
```

- Tests on Atmel SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
Without `lpj`	71 ms	
With `lpj`	8 ms	-63 ms

- This calculation was longer before 2.6.39 (about 200 ms).

Bootloader optimizations

U-Boot - Remove unnecessary functionality

Recompile U-Boot to remove features not needed in production

- Disable as many features as possible in `include/configs/<soc>-<board>.h`
- Examples: MMC, USB, Ethernet, dhcp, ping, command line edition, command completion...
- A smaller and simpler U-Boot is faster to load and faster to initialize.

U-Boot - Remove the boot delay

- Remove the boot delay:
  `setenv bootdelay 0`
- This usually saves several seconds!
- Before you do that, recompile U-Boot with `CONFIG_ZERO_BOOTDELAY_CHECK`, documented in `doc/README.autoboot`. It allows to stop the autoboot process by hitting a key even if the boot delay is set to `0`.

U-Boot - Simplify scripts

Some boards have over complicated scripts:

```
bootcmd=run bootf0
bootf0=run ${args0}; setenv bootargs ${bootargs} \
maximasp.kernel=maximasp_nand.0:kernel0; nboot 0x70007fc0 kernel0
```

Let's replace this by:

```
setenv bootargs 'mem=128M console=tty0 consoleblank=0 console=ttyS0,57600 \
mtdparts=maximasp_nand.0:2M(u-boot)ro,512k(env0)ro,512k(env1)ro,\
4M(kernel0),4M(kernel1),5M(kernel2),100M(root0),100M(root1),-(other)\
rw ubi.mtd=root0 root=ubi0:rootfs rootfstype=ubifs earlyprintk debug \
user_debug=28 maximasp.board=EEKv1.3.x \
maximasp.kernel=maximasp_nand.0:kernel0'
setenv bootcmd 'nboot 0x70007fc0 kernel0'
```

This saved 56 ms on this ARM9 system (400 MHz)!

Bootloader: copy the exact kernel size

- When copying the kernel from flash to RAM, we still see many systems that copy too many bytes, not taking the exact kernel size into account.
- In U-Boot, use the `nboot` command:
  `nboot ramaddr 0 nandoffset`
- U-Boot using the kernel size information stored in the `uImage` header to know how many bytes to copy.

U-Boot - Optimize kernel loading

- After copying the kernel `uImage` to RAM, U-Boot always moves it to the load address specified in the `uImage` header.
- A CRC check is also performed.

```
[16.590578 0.003404] ## Booting kernel from Legacy Image at 21000000 ...
[16.595204 0.004626]    Image Name:   Linux-3.10.0+
[16.597986 0.002782]    Image Type:   ARM Linux Kernel Image (uncompressed)
[16.602881 0.004895]    Data Size:    3464112 Bytes = 3.3 MiB
[16.606542 0.003661]    Load Address: 20008000
[16.608903 0.002361]    Entry Point:  20008000
[16.611256 0.002353]    Verifying Checksum ... OK
[17.134317 0.523061] ## Flattened Device Tree blob at 22000000
[17.137695 0.003378]    Booting using the fdt blob at 0x22000000
[17.141707 0.004012]    Loading Kernel Image ... OK
[18.005814 0.864107]    Loading Device Tree to 2bb12000, end 2bb1a0b6 ... OK
```

Kernel CRC check time

Kernel `memmove` time

U-Boot - Remove unnecessary memmove (1)

- You can make U-Boot skip the `memmove` operation by directly loading the `uImage` at the right address.
- Compute this address:

```
Addr = Load Address - uImage header size
Addr = Load Address - (size(uImage) - size(zImage))
Addr = 0x20008000 - 0x40 = 0x20007fc0
```

```
[16.590927 0.003407] ## Booting kernel from Legacy Image at 20007fc0 ...
[16.595547 0.004620]    Image Name:   Linux-3.10.0+
[16.598351 0.002804]    Image Type:   ARM Linux Kernel Image (uncompressed)
[16.603228 0.004877]    Data Size:    3464112 Bytes = 3.3 MiB
[16.606907 0.003679]    Load Address: 20008000
[16.609256 0.002349]    Entry Point:  20008000
[16.611619 0.002363]    Verifying Checksum ... OK
[17.135046 0.523427] ## Flattened Device Tree blob at 22000000
[17.138589 0.003543]    Booting using the fdt blob at 0x22000000
[17.142575 0.003986]    XIP Kernel Image ... OK
[17.156358 0.013783]    Loading Device Tree to 2bb12000, end 2bb1a0b6 ... OK
```

Kernel CRC check time

Kernel `memmove` time (skipped)

U-Boot - Remove unnecessary memmove (2)

Results on Atmel SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
Default	1.433 s	
Optimum load address	0.583 s	-0.85 s

Measured between `Booting kernel` and `Starting kernel ...`

U-Boot - Remove kernel CRC check

- Fine in production when you never have data corruption copying the kernel to RAM.
- Disable CRC checking with a U-boot environment variable: `setenv verify no`

Results on Atmel SAMA5D3 Xplained (ARM), Linux 3.10:

	Time	Diff
With CRC check	583 ms	
Without CRC check	60 ms	-523 ms

Measured between `Booting kernel` and `Starting kernel ...`

Further U-Boot optimizations

- Silence U-Boot console output. You will need to compile U-Boot with `CONFIG_SILENT_CONSOLE` and `setenv silent yes`.
  See `doc/README.silent` for details.
- Ultimate solution: use U-Boot's *Falcon* mode.
  U-Boot is split in two parts: the SPL (Secondary Program Loader) and the U-Boot image. U-Boot can then configure the SPL to load the Linux kernel directly, instead of the U-Boot image.
  See `doc/README.falcon` for details.

Kernel compression and size optimizations

After optimizing the time to load the kernel in the bootloader, we are ready to experiment with kernel options impacting size:

- Kernel compression options
- Optimizing kernel code for size

Kernel compression options

Results on TI AM335x (ARM), 1 GHz, Linux 3.13-rc4

Timestamp	gzip	lzma	xz	lzo	lz4	uncompressed
Size	4308200	3177528	**3021928**	4747560	5133224	8991104
Copy	0.451 s	0.332 s	**0.315 s**	0.499 s	0.526 s	0.914 s
Uncompress	0.945 s	2.329 s	2.056 s	0.861 s	**0.851 s**	**0.687 s**
Total	**5.516 s**	6.066 s	5.678 s	5.759 s	6.017 s	8.683 s

Results on Atmel AT91SAM9263 (ARM), 200 MHz, Linux 3.13-rc4

Timestamp	gzip	lzma	xz	lzo	lz4	uncompressed
Size	3016192	2270064	**2186056**	3292528	3541040	5775472
Copy	4.105 s	3.095 s	**2.981 s**	4.478 s	4.814	7.836 s
Uncompress	1.737 s	8.691 s	6.531 s	**1.073** s	1.225 s	N/A
Total	8.795 s	14.200 s	11.865 s	**8.700 s**	9.368 s	N/A

Results indeed depend on I/O and CPU performance!

Optimize kernel for size

- `CONFIG_CC_OPTIMIZE_FOR_SIZE`: possibility to compile the kernel with `gcc -Os` instead of `gcc -O2`.
- Such optimizations give priority to code size at the expense of code speed.
- Results: the initial boot time is better (smaller size), but the slower kernel code quickly offsets the benefits. Your system will run slower!

Results on Atmel SAMA5D3 Xplained (ARM), Linux 3.10, gzip compression:

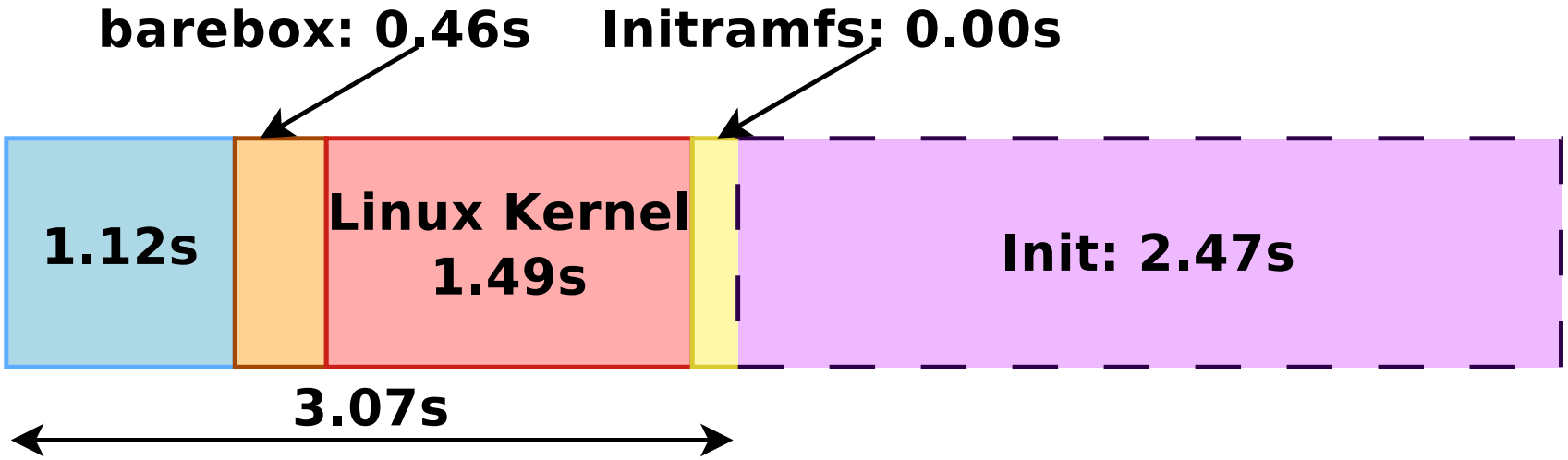
Timestamp	O2	Os	Diff
Starting kernel	4.307 s	4.213 s	-94 ms
Starting init	5.593 s	5.549 s	-44 ms
Login prompt	21.085 s	22.900 s	+ 1.815 s

Replacing U-Boot by Barebox

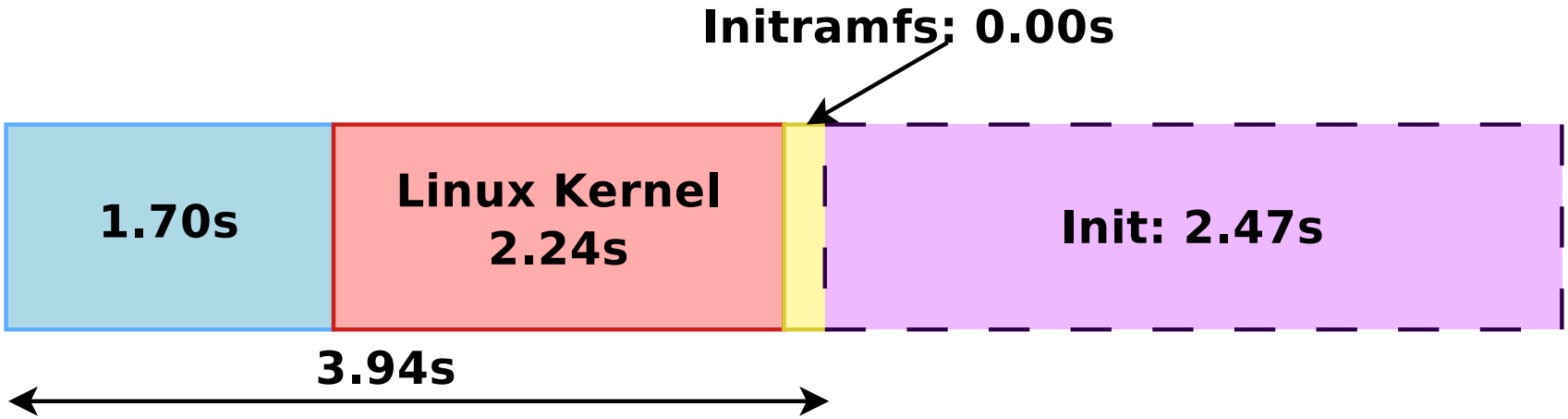
- We saved time by replacing U-Boot by Barebox.
- Barebox can be made very small too, and loads the Linux kernel with the CPU caches on. This significantly reduces kernel decompression time!
- At this stage, we can't share our benchmarks yet. They are not fair for U-Boot, as we did optimize Barebox further than we did with U-Boot.

Removing the bootloader (1)

Principle: instead of loading the bootloader and then the kernel, load the kernel right away!



Using `AT91bootstrap` to boot the Linux kernel:



Removing the bootloader (2)

- In our particular case, though, we can see that we are losing the main advantages of Barebox: it uses the CPU caches while loading the kernel.
- Skipping the bootloader is not always the best choice!

`http://bootlin.com/blog/starting-linux-directly-from-at91bootstrap3/`

Questions?

Michael Opdenacker

michael.opdenacker@bootlin.com

http://bootlin.com/pub/conferences/2014/elc/opdenacker-boot-time/