Boot time optimization

free electrons

Embedded Linux
Experts

Latest update: June 2, 2017.

Document updates and sources:
`http://free-electrons.com/doc/training/boot-time`

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@free-electrons.com

Rights to copy

Document sources: http://git.free-electrons.com/training-materials/

Hyperlinks in the document

There are many hyperlinks in the document

- Regular hyperlinks:
  `http://kernel.org/`
- Kernel documentation links:
  `dev-tools/kmemcheck`
- Links to kernel source files and directories:
  `drivers/input/`
  `include/linux/fb.h`
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  `platform_get_irq()`
  `GFP_KERNEL`
  `struct file_operations`

Thanks

Special thanks to

- Atmel Corporation
  - For funding the development of the first version of these materials.
  - For providing the SAMA5 evaluation kit for the practical labs.

Free Electrons at a glance

- Engineering company created in 2004
(not a training company!)
- Locations: Orange, Toulouse, Lyon (France)
- Serving customers all around the world
- Head count: 12
Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel Free Software / Open Source for embedded and real-time systems.
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

Free Electrons on-line resources

- All our training materials:
  http://free-electrons.com/docs/
- Technical blog:
  http://free-electrons.com/blog/
- Quarterly newsletter:
  http://lists.free-electrons.com/mailman/listinfo/newsletter
- News and discussions (Google +):
  https://plus.google.com/+FreeElectronsDevelopers
- News and discussions (LinkedIn):
  http://linkedin.com/groups/Free-Electrons-4501089
- Quick news (Twitter):
  http://twitter.com/free_electrons
- Elixir - browse Linux kernel sources on-line:
  http://elixir.free-electrons.com

Generic course information

free electrons

Embedded Linux
Experts

Hardware used in this training session

Using SAMA5D31-EK boards in the practical labs

- Atmel SAMA5D3x embedded MPU (cortex A5 at 536MHz)
- 4 Gb DDR2, 2 Gb NAND flash, 128 Mb NOR, 32 Mb SPI Serial DataFlash
- Ethernet 10/100/1Gb
- Two USB 2.0 Hosts, one USB 2.0 Host/Device
- 5.0" WVGA resistive TFT LCD module with four QTouch keys
- MicroSD card slot, JTAG, MMC/MMC+/SD/SDIO/CE-ATA, CAN, HDMI
- Priced at $595

Course outline - Morning

Generic optimizations

- Principles
- Measuring
- Filesystems
- Userland
- Kernel
- Bootloader

Labs: flashing the demo image, measuring the boot time, optimizing the boot scripts, optimizing the kernel

Course outline - Afternoon

Generic optimizations

- Bootloader

Advanced optimizations

- Kernel
- Userland

Labs: changing the bootloader, reducing the kernel size and features

Principles

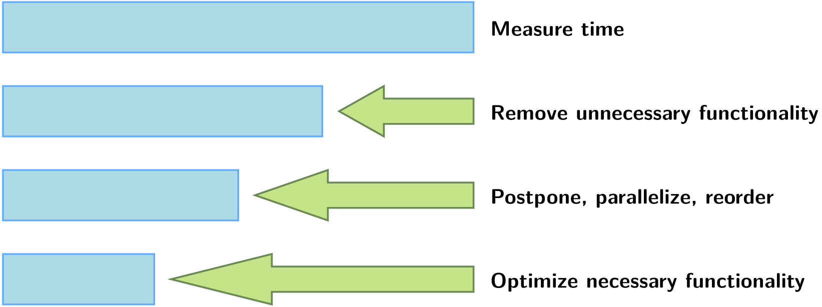
free electrons

Embedded Linux Experts

Set your goals

- Reducing boot time implies measuring boot time!
- You will have to choose reference events at which you start and stop counting time.
- What you choose will depend on the goal you want to achieve. Here are typical cases:
  - Showing a splash screen or an animation, playing a sound to indicate the board is booting
  - Starting a listening service to handle a particular message
  - Being fully functional as fast as possible

Boot time reduction methodology



Boot time components

Generic boot sequence

Power up sequence	**Rom code**	**1st stage bootloader (bootstrap)**	**2nd stage bootloader**	**Linux kernel**	**init scripts**	**Critical application**

Example: typical boot sequence on Atmel AT91

Power Up Sequence	**RomBOOT**	**AT91Bootstrap**	**U-Boot**	**Linux Kernel**	**Init scripts**	**Critical Application**

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 user space 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.

We will follow this order during the practical labs.

Worst things first!

Premature optimization is the root of all evil.
Donald Knuth

- Taking the time to measure time carefully is important.
- Find the worst consumers of time and address them first.
- You can waste a lot of time if you start optimizing minor spots first.

Build automation

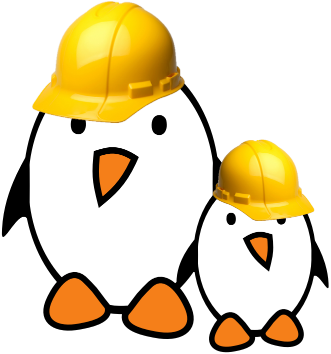
- Very important to automate the way the root filesystem is built, if not done yet. That's always the first thing we do in boot time reduction projects, and it's worth investing 1 or 2 days doing this.
- Otherwise, you may lose existing optimizations or introduce new bugs when making further optimizations. Without a build system, you will waste a lot of time too.
- Can be done through build systems such as Buildroot or Yocto, or using the original build automation of the project.
- Can also be done for kernel and bootloader optimizations, though the need is less critical.

Generic ideas

Some ideas to keep in mind while trying to reduce the boot time:

- The fastest code is code that is not executed
- A big part of booting is actually loading code and data from the storage to RAM. Reading less means booting faster. I/O are expensive!
- The root filesystem may take longer to mount if it is bigger.
- So, even code that is not executed can make your boot time longer.
- Also, try to benchmark different types of storage. It has happened that booting from SD card was actually faster than booting from NAND.

Practical lab - Getting started



- Learn how to interact with the board
- Learn how to flash it

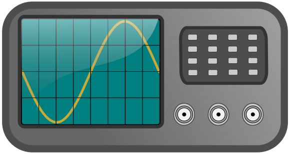
Measuring

free electrons

Embedded Linux
Experts

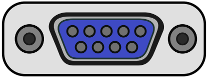
Time measurement equipment: hardware

- The best equipment is an oscilloscope.
- Allows to time the "Power on" event (connected to a power rail), or any event (connected to a GPIO pin, for example), all this in a very accurate way.
- Easy to write to a GPIO at all the stages of system booting.
- Some oscilloscopes are getting affordable. Example: Bitscope Pocket Analyzer (295 AUD, supported on Linux, `http://www.bitscope.com/product/BS10/`)



Time measurement equipment: serial port

- Useful when you don't have an oscilloscope, or don't want to make take any risk connecting wires to the hardware.
- Usually relies on software which times messages received from the board's serial port (serial port absolutely required). Such software runs on a PC connected to the serial port.
- Need a real serial port (directly connected to the CPU), immediately usable from the earliest parts of the boot process.
- Limitation: won't be able to time the "Power on" event in an accurate way. But acceptable as you can assume that the time to run the first stage bootloader is constant.



Time measurement equipment: USB to serial

- Serial ports over USB **device** are fine if there's an on-board serial-to-USB chip directly connected to the CPU serial port (very frequent).
- Attaching a USB-to-serial dongle to a USB **host** port on the device won't do: USB is available much later and messages go through more complex software stacks (loss of time accuracy).
- All development boards have a standard or USB serial port.

grabserial

- From Tim Bird: `http://elinux.org/Grabserial`
- 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

Serial device

Reset the time to 0 when this string is found

Print time for each received line

End recording time in seconds

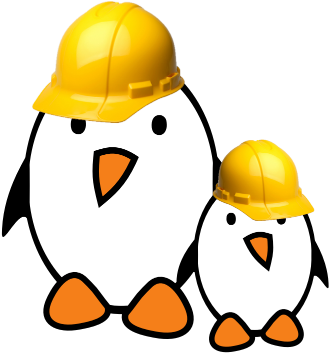
```
$ ~/bin/grabserial -d /dev/ttyACM0 -m RomBOOT  -t -e 30
[0.000002 0.000002] RomBOOT
[0.054440 0.054440]
[0.054594 0.000154]
[0.054719 0.000125] AT91Bootstrap 3.5.2 (Wed Jan 30 18:42:28 CET 2013)
[0.059009 0.004290]
[0.059185 0.000176] 1-Wire: Loading 1-Wire information ...
[0.062645 0.003460] 1-Wire: ROM Searching ... Done, 0x3 1-Wire chips found
[0.110066 0.047421]
[0.110175 0.000109] 1-Wire: BoardName | [Revid] | VendorName
[0.162743 0.052568]   #0x0  SAMA5D34-CM [C9]      EMBEST
[0.214729 0.051986]   #0x1  SAMA5D3x-MB [B9]      FLEX
[0.266321 0.051592]   #0x2  SAMA5D3x-DM [A9]      FLEX
```

Arrival time of the first character

Elapsed time since the previous time stamp

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.

Practical lab - Measuring



- Time the various components of boot time

Filesystem optimizations

free electrons

Embedded Linux Experts

Corrections, suggestions, contributions and translations are welcome!

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.

Different filesystem for different storage types

- Block storage (including memory cards, eMMC)
  - ext2, ext3, ext4
  - xfs, jfs, reiserfs
  - btrfs
  - f2fs
  - SquashFS
- Raw flash storage
  - JFFS2
  - YAFFS2
  - UBIFS
  - ubiblock + SquashFS

See our embedded Linux training materials for full details:
`http://free-electrons.com/doc/training/embedded-linux/`
See also our flash filesystem benchmarks:
`http://elinux.org/Flash_Filesystem_Benchmarks`.

Block filesystems

For block storage

- 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.

JFFS2

For raw flash storage

- Mount time depending on filesystem size: the kernel has to scan the whole filesystem at mount time, to read which block belongs to each file.
- Need to use the `CONFIG_JFFS2_SUMMARY` kernel option to store such information in flash. This dramatically reduces mount time.
- Benchmark on ARM:
  from 16 s to 0.8 s for a 128 MB partition.
- Rather poor read and write performance,
  compared to YAFFS2 and UBIFS.

YAFFS2

For raw flash storage

- Good mount time
- Good read and write performance
- Drawbacks: no compression, not in the mainline Linux kernel

UBIFS

For raw flash storage

- Advantages:
  - Good read and write performance (similar to YAFFS2)
  - Other advantages: better for wear leveling (can operate on the whole UBI space, not only within a single partition).
- Drawbacks:
  - Not appropriate for small partitions (too much metadata overhead). Use JFFS2 or JAFFS2 instead.
  - Not so good mount time, because of the time needed to initialize UBI (*UBI Attach*: at boot time or running `ubi_attach` in user space).
  - Addressed by *UBI Fastmap*, introduced in Linux 3.7. See next slides.

How UBI Fastmap works

- *UBI Attach*: needs to read UBI metadata by scanning all erase blocks. Time proportional to the storage size.
- *UBI Fastmap* stores such information in a few flash blocks (typically at UBI detach time during system shutdown) and finds it there at boot time.
- This makes *UBI Attach* time constant.
- If *Fastmap* information is invalid (unclean system shutdown, for example), it falls back to scanning (slower, but correct, and *Fastmap* will work again during the next boot).
- Details: ELCE 2012 presentation from Thomas Gleixner: `http://elinux.org/images/a/ab/UBI_Fastmap.pdf`

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 Free Electrons 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.

Finding the best filesystem

- Raw flash storage: UBIFS with CONFIG_UBI_FASTMAP is probably the best solution.
- Block storage: SquashFS best solution for root filesystems which can be read-only. Btrfs and f2fs probably the best solutions for read/write filesystems.
- 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!
- Do not focus only on boot time.
  For systems in which read and write performance matters, we recommend to use separate root filesystem (for quick boot time) and data partitions (for good runtime performance).

Init scripts

free electrons

Embedded Linux Experts

Corrections, suggestions, contributions and translations are welcome!

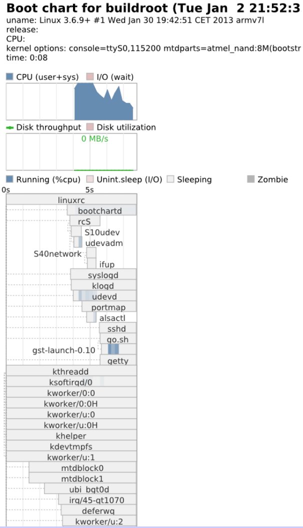
Methodology

There are multiple ways to reduce the time spent in init scripts before starting the application:

- Start the application as soon as possible after only the strictly necessary dependencies.
- Simplify shell scripts
- Start the application even before `init`

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
...
```

Initial measures

In the initial demo, we measure the time taken to get the login prompt (RomBOOT is excluded).

Total: 11.70s.

Init

Starting as soon as possible after all the dependencies are started:

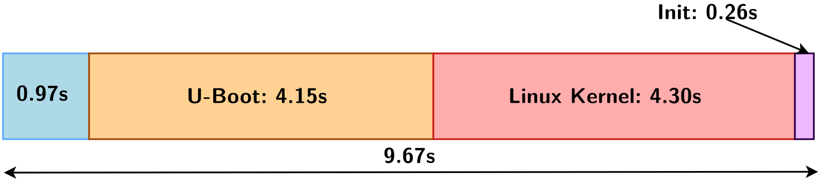
- Depends on your `init` program. Here we are assuming sysV `init` scripts.
- `init` scripts run in alphanumeric order and start with a letter (K for stop (**k**ill) and S for **s**tart).
- You want to use the lowest number you can for your application.
- You can even replace `init` with your application!

How fast would we be if we could be the first started application?

Results

Before: 11.70s

After:



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`.
- Replace `udev` with `mdev`. `mdev` is part of BusyBox. It is not running as a daemon and you can either run it only once or have it handling hotplug events.
- Remove `udev` (or `mdev`) if you just need it to create device files. Use `devtmpfs` (`CONFIG_DEVTMPFS`) instead, automatically managed by the kernel, and cheaper.

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)!

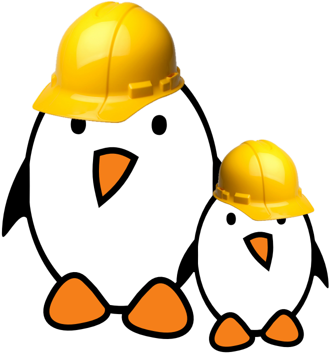
Reduce size (1)

- Strip your executables and libraries, removing ELF sections only needed for development and debugging. The `strip` command is provided by your cross-compiling toolchain. `BR2_STRIP_strip` in Buildroot.
- `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. `BR2_STRIP_sstrip` in Buildroot.

Reduce size (2)

- use mklibs, available at http://packages.debian.org/sid/mklibs.
  - mklibs produces cut-down shared libraries that contain only the routines required by a particular set of executables. Really useful with big libraries like OpenGL and QT. It even works without having the source code.
  - Available in Yocto, but not in Buildroot (at least in 2013.11).
  - Limitation: mklibs could remove dlopened libraries (loaded "manually" by applications) because it doesn't see them.

Practical lab - Reducing time in init-scripts



- Regenerate the root filesystem with Buildroot
- Use bootchart to measure boot time

C Libraries

free electrons

Embedded Linux Experts

Corrections, suggestions, contributions and translations are welcome!

glibc

- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Of course, actively maintained
- By default, quite big for small embedded systems: approx 2.5 MB on ARM (version 2.9 - `libc`: 1.5 MB, `libm`: 750 KB)
- But some features not needed in embedded systems can be configured out (merged from the old *eglibc* project).
- `http://www.gnu.org/software/libc/`

uClibc-ng (1)

- http://uclibc-ng.org/
- A continuation of the old uClibc project
- License: LGPL
- Lightweight C library for small embedded systems
  - High configurability: many features can be enabled or disabled through a menuconfig interface
  - Supports most embedded architectures
  - Supports no-MMU architectures (ARM Cortex-M, Blackfin, etc.)
  - No guaranteed binary compatibility. May need to recompile applications when the library configuration changes.
  - Focus on size rather than performance
  - Small compile time

uClibc-ng (2)

- Most of the applications compile with uClibc-ng. This applies to all applications used in embedded systems.
- Size (arm): 3.5 times smaller than glibc!
  - uClibc-ng 1.0.14: approx. 716kB (libuClibc: 282kB, libm: 73kB)
  - glibc 2.22: approx 2.5 MB
- Some features not available or limited: priority-inheritance mutexes, fixed Name Service Switch functionality, etc.
- Used on a large number of production embedded products, including consumer electronic devices

Honey, I shrunk the programs!

- Executable size comparison on ARM, tested with *glibc* 2.22 and *uClibc-ng* 1.0.14
- Plain ``hello world'' program (stripped):

helloworld	static	dynamic
uClibc	33.4kB	2.5kB
uClibc with Thumb-2	25.4kB	2.5kB
eglibc with Thumb-2	479kB	2.7kB

- Busybox (stripped):

busybox	static	dynamic
uClibc	818kB	664kB
uClibc with Thumb-2	602kB	504kB
eglibc with Thumb-2	1206kB	503kB

musl C library

http://www.musl-libc.org/

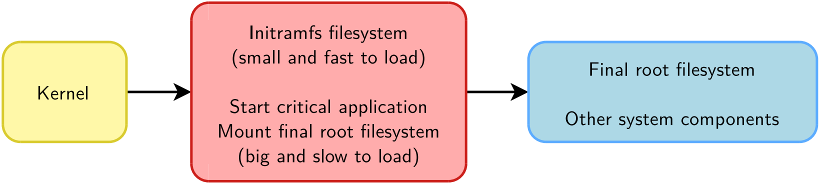
- A lightweight, fast and simple library for embedded systems
- Created while uClibc's development was stalled
- In particular, great at making small static executables
- Permissive license (MIT)
- Compare features with other C libraries:
  http://www.etalabs.net/compare_libcs.html
- Supported by build systems such as Buildroot

Other smaller C libraries

- Several other smaller C libraries have been developed, but none of them have the goal of allowing the compilation of large existing applications
- They can run only relatively simple programs, typically to make very small static executables and run in very small root filesystems.
- Choices:
  - Dietlibc, `http://fefe.de/dietlibc/`. Approximately 70 KB.
  - Newlib, `http://sourceware.org/newlib/`
  - Klibc, `http://www.kernel.org/pub/linux/libs/klibc/`, designed for use in an *initramfs* or *initrd* at boot time.

Initramfs

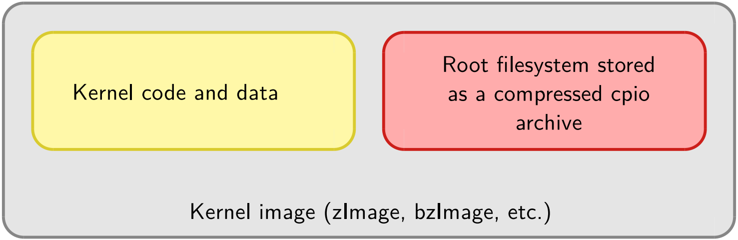
An idea is to use a very small initramfs, just enough to start the critical application and then switch to the final root filesystem.



rootfs in memory: initramfs (1)

- It is also possible to have the root filesystem integrated into the kernel image
- It is therefore loaded into memory together with the kernel
- This mechanism is called **initramfs**
  - It integrates a compressed archive of the filesystem into the kernel image
  - Variant: the compressed archive can also be loaded separately by the bootloader.
- It is useful for two cases
  - Fast booting of very small root filesystems. As the filesystem is completely loaded at boot time, application startup is very fast.
  - As an intermediate step before switching to a real root filesystem, located on devices for which drivers not part of the kernel image are needed (storage drivers, filesystem drivers, network drivers). This is always used on the kernel of desktop/server distributions to keep the kernel image size reasonable.

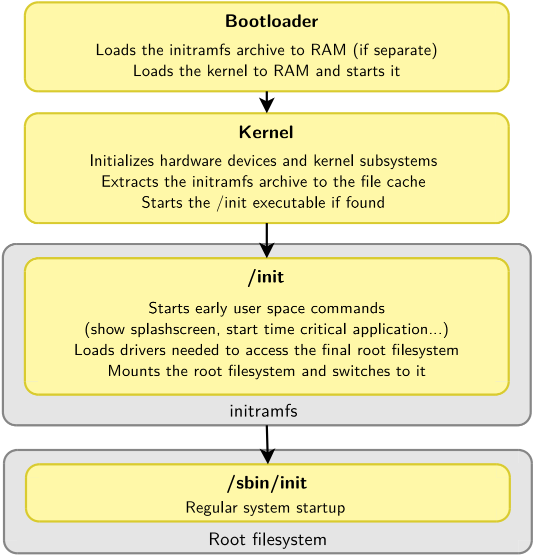
rootfs in memory: initramfs (2)



rootfs in memory: initramfs (3)

- The contents of an initramfs are defined at the kernel configuration level, with the `CONFIG_INITRAMFS_SOURCE` option
  - Can be the path to a directory containing the root filesystem contents
  - Can be the path to a cpio archive
  - Can be a text file describing the contents of the initramfs (see documentation for details)
- The kernel build process will automatically take the contents of the `CONFIG_INITRAMFS_SOURCE` option and integrate the root filesystem into the kernel image
- Details (in kernel sources):
  `Documentation/filesystems/ramfs-rootfs-initramfs.txt`
  `Documentation/early-userspace/README`

Overall booting process with initramfs



Initramfs for boot time reduction

Create the smallest initramfs possible, just enough to start the critical application and then switch to the final root filesystem with `switch_root`:

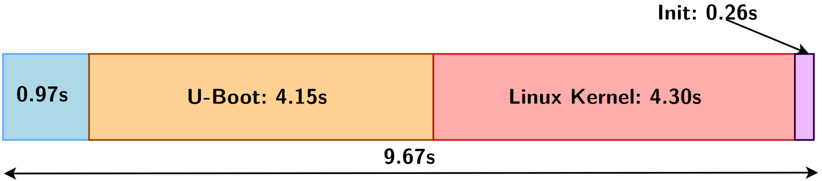
- Use a light C library reduced to the minimum, *uClibc* if you are not yet using it for your root filesystem
- Reduce BusyBox to the strict minimum. You could even do without it and implement `/init` in C.
- Use statically linked applications (less CPU overhead, less libraries to load, smaller initramfs if no libraries at all). `BR2_PREFER_STATIC_LIB` in Buildroot.

Statically linked executables: licensing constraints

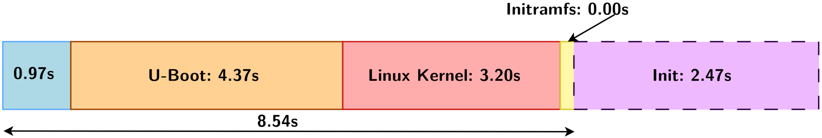
- Statically linked executables are very useful to reduce size (especially in small initramfs), and require less work to start.
- However, the LGPL license in libraries (uClibc, glibc), require to leave the user the ability to relink the executable with a modified version of the library.
- Solution to keep static binaries:
  - Either provide the executable source code (even proprietary), allowing to recompile it with a modified version of the library. That's what you do when you ship a static BusyBox.
  - Or also provide a dynamically linked version of the executable (in a separate way), allowing to use another library version.
- References:
  `http://gnu.org/licenses/gpl-faq.html#LGPLStaticVsDynamic`
  `http://gnu.org/copyleft/lesser.html#section4`

Results: after switch to initramfs

Before:



After:



- U-Boot takes more time because the kernel is bigger
- `init` is started earlier because UBI is not initialized yet (slow, UBIFS root filesystem)

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.free-electrons.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

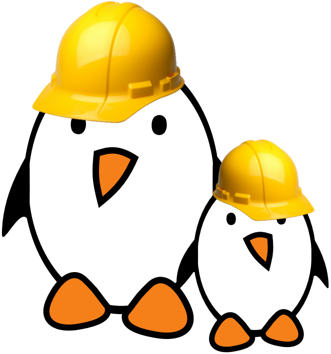
http://free-electrons.com/blog/super-fast-linux-splashscreen/

Animated splashscreen

Still slow to read and write entire screens. Just draw useful pixels and even create an animation!

- Create a simple C program that just animates pixels and simple geometric shapes on the framebuffer!
- Example: `http://free-electrons.com/pub/code/fb/anim.c`. On a 400 MHz ARM9 system: starts drawing in 10 ms size: 24 KB, compiled statically).

Practical lab - Reducing time in init-scripts



- Simplify user space scripts

Applications

free electrons

Embedded Linux
Experts

Corrections, suggestions, contributions and translations are welcome!

strace

- Allows to trace all the system calls made by an application and its children.
- Useful to:
  - Understand how time is spent in user space
  - For example, easy to find file open attempts (`open()`), file access (`read()`, `write()`), and memory allocations (`mmap2()`). Can be done without any access to source code!
  - Find the biggest time consumers (low hanging fruit)
  - Find unnecessary work done in applications and scripts. Example: opening the same file(s) multiple times, or trying to open files that do not exist.
- Limitation: you can't trace the `init` process!

Using strace

- `strace` can be compiled by your build system
- Even easier: drop a ready-made static binary for your architecture, just when you need it. See http://git.free-electrons.com/users/michael-opdenacker/static-binaries/tree/strace
- Recommended usage:
  `strace -f -tt -o strace.log <program> <arguments>`
  - `-f`: follow child processes
  - `-tt`: display timestamps with microsecond accuracy

strace output

```
0.000625 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
0.000877 open("/lib/libgstaudio-0.10.so.0", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
0.335815 open("/usr/lib/libgstaudio-0.10.so.0", O_RDONLY|O_CLOEXEC) = 12
0.000778 read(12, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0(q\0\0004\0\0\0"..., 512) = 512
0.000699 lseek(12, 171824, SEEK_SET) = 171824
0.000541 read(12, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 1040) = 1040
0.000775 lseek(12, 171536, SEEK_SET) = 171536
0.000535 read(12, "A8\0\0\0aeabi\0\1.\0\0\0\0057-A\0\6\n\7A\10\1\t\2\n\6\22"..., 57) = 57
0.000678 fstat64(12, {st_mode=S_IFREG|0755, st_size=172864, ...}) = 0
0.000714 mmap2(NULL, 204460, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 12, 0) = 0xb4d82000
0.000684 mprotect(0xb4dab000, 28672, PROT_NONE) = 0
0.000618 mmap2(0xb4db2000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 12, 0x28) = 0xb4db2
0.000875 close(12)                  = 0
0.005504 munmap(0xb4ea7000, 64)     = 0
0.004886 open("/proc/self/cmdline", O_RDONLY|O_LARGEFILE) = 12
0.001027 fstat64(12, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
0.000780 fcntl64(12, F_GETFL)       = 0x20000 (flags O_RDONLY|O_LARGEFILE)
0.000520 fstat64(12, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
0.000645 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb4ea7000
0.000673 _llseek(12, 0, [0], SEEK_CUR) = 0
0.000813 read(12, "./HomeAutomation\0-qws\0", 4096) = 22
0.000776 read(12, "", 3072)         = 0
0.000848 close(12)                  = 0
0.000438 munmap(0xb4ea7000, 4096)   = 0
0.006391 clock_gettime(CLOCK_MONOTONIC, {251, 636907995}) = 0
0.001903 uname({sys="Linux", node="buildroot", ...}) = 0
0.049023 socket(PF_FILE, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 12
0.000664 connect(12, {sa_family=AF_FILE, sun_path="/var/run/nscd/socket"}, 110) = -1 ENOENT (No such file or dire
0.000815 close(12)                  = 0
```

oprofile

- Two ways of working: *legacy* mode and *perf_events* mode
- *legacy* mode:
  - Low accuracy, use a kernel driver to profile
  - `CONFIG_OPROFILE`
  - User space tools: `opcontrol` and `oprofiled`
- *perf_events* mode:
  - Uses hardware performance counters
  - `CONFIG_PERF_EVENTS` and `CONFIG_HW_PERF_EVENTS`
  - User space tools: `operf`

oprofile: usage

- *legacy* mode:

```
opcontrol --vmlinux=/path/to/vmlinux # optional step
opcontrol --start
/my/command
opcontrol --stop
```

- *perf_events* mode:

```
operf --vmlinux=/path/to/vmlinux /my/command
```

- Get the results with:

```
opreport
```

oprofile

```
# opreport
Using /var/lib/oprofile/samples/ for samples directory.
CPU: CPU with timer interrupt, speed 393 MHz (estimated)
Profiling through timer interrupt
          TIMER:0|
  samples|      %|
------------------
     1540 78.3715 no-vmlinux
      105  5.3435 libQtGui.so.4.8.4
       66  3.3588 libc-2.15.so
       64  3.2570 libQtCore.so.4.8.4
       58  2.9517 ld-2.15.so
       45  2.2901 libgobject-2.0.so.0.3000.3
       37  1.8830 libgstreamer-0.10.so.0.30.0
       13  0.6616 libglib-2.0.so.0.3000.3
        9  0.4580 libQtScript.so.4.8.4
        6  0.3053 libgcc_s.so.1
        4  0.2036 libQtDeclarative.so.4.8.4
        4  0.2036 libstdc++.so.6.0.17
        3  0.1527 libpthread-2.15.so
        2  0.1018 busybox
        2  0.1018 libQtSvg.so.4.8.4
        2  0.1018 libQtWebKit.so.4.9.3
        2  0.1018 libgthread-2.0.so.0.3000.3
        1  0.0509 HomeAutomation
        1  0.0509 libQtNetwork.so.4.8.4
        1  0.0509 libphonon_gstreamer.so
```

perf

- Uses hardware performance counters
- `CONFIG_PERF_EVENTS` and `CONFIG_HW_PERF_EVENTS`
- User space tool: `perf`. It is part of the kernel sources so it is always in sync with your kernel.
- Usage:

```
perf record /my/command
```

- Get the results with:

```
perf report
```

perf

```
20.91%  gst-launch-0.10  libavcodec.so.53.35.0        [.] 0x00000000003bdaa1
15.45%  gst-launch-0.10  libgstflump3dec.so           [.] 0x0000000000014b42
 3.16%  gst-launch-0.10  libglib-2.0.so.0.3600.2      [.] 0x00000000000882c9
 2.99%  gst-launch-0.10  libc-2.17.so                 [.] __memcpy_ssse3_back
 2.37%  gst-launch-0.10  liboil-0.3.so.0.3.0          [.] 0x000000000004417e
 2.24%  gst-launch-0.10  libgobject-2.0.so.0.3600.2   [.] g_type_value_table_peek
 1.53%  gst-launch-0.10  libc-2.17.so                 [.] vfprintf
 1.37%  gst-launch-0.10  libgstreamer-0.10.so.0.30.0  [.] 0x0000000000026fd8
 1.29%  gst-launch-0.10  ld-2.17.so                   [.] do_lookup_x
 0.99%  gst-launch-0.10  libpthread-2.17.so           [.] pthread_mutex_lock
 0.98%  gst-launch-0.10  libgobject-2.0.so.0.3600.2   [.] g_type_check_value
 0.93%  gst-launch-0.10  libgstavi.so                 [.] 0x00000000000119f9
 0.88%  gst-launch-0.10  libgstreamer-0.10.so.0.30.0  [.] gst_value_list_get_type
 0.85%  gst-launch-0.10  libc-2.17.so                 [.] __random
 0.66%  gst-launch-0.10  [kernel.kallsyms]            [k] clear_page_c_e
 0.62%  gst-launch-0.10  [kernel.kallsyms]            [k] try_to_wake_up
 0.61%  gst-launch-0.10  [kernel.kallsyms]            [k] page_fault
 0.58%  gst-launch-0.10  libgobject-2.0.so.0.3600.2   [.] g_type_is_a
 0.57%  gst-launch-0.10  libc-2.17.so                 [.] __strcmp_sse42
 0.57%  gst-launch-0.10  [kernel.kallsyms]            [k] radix_tree_lookup_element
 0.57%  gst-launch-0.10  libc-2.17.so                 [.] malloc
 0.57%  gst-launch-0.10  libc-2.17.so                 [.] _int_malloc
 0.55%  gst-launch-0.10  libgobject-2.0.so.0.3600.2   [.] g_type_check_instance_is_a
 0.53%  gst-launch-0.10  [kernel.kallsyms]            [k] __ticket_spin_lock
 0.53%  gst-launch-0.10  libgobject-2.0.so.0.3600.2   [.] g_type_check_value_holds
 0.53%  gst-launch-0.10  libgstffmpeg.so              [.] 0x000000000001e40c
 0.51%  gst-launch-0.10  libgstreamer-0.10.so.0.30.0  [.] gst_structure_id_get_value
 0.50%  gst-launch-0.10  libc-2.17.so                 [.] _IO_default_xsputn
 0.50%  gst-launch-0.10  [kernel.kallsyms]            [k] tg_load_down
```

Toolchains

Try to use an optimized toolchain for your platform. That is something you'll have to benchmark near the end of your project.

- gcc version 4.6.3 (Sourcery CodeBench Lite 2012.03-57: 9.63s
- gcc version 4.7.3 20130226 (prerelease) (crosstool-NG linaro-1.13.1-4.7-2013.03-20130313 - Linaro GCC 2013.03): 9.26s

Results may vary depending on the size/features of your root filesystem. Here, we mainly benchmark the kernel performance as the root filesystem doesn't take much time. Note: Don't compare with a Buildroot toolchain as these are using *glibc* and not *uClibc* and then the initramfs has a quite different size. Compiling the kernel with a Linaro toolchain and using a Buildroot generated root filesystem yields the same results as compiling everything with Buildroot.

Linker optimizations (1)

Group application code used at startup

- Find the functions called during startup, for example using the `-finstrument-functions` gcc option.
- Create a custom linker script to reorder these functions in the call order. You can achieve that by putting each function in their own 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.

Details:

http://blogs.linux.ie/caolan/2007/04/24/controlling-symbol-ordering/

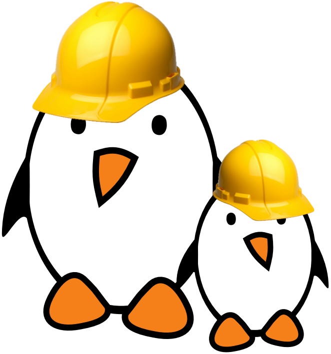
Linker optimizations (2)

- Here's a very simple way to find the maximum savings you can expect from this technique:
  - Start the application once and measure its startup time.
  - Start the application and measure its startup time again. Its code should still be in the Linux file cache, and the code loading time will be zero.
- You now know how much time it took to load the application code (and its libraries) the first time. Linker optimizations will save less than this upper limit.
- You can then decide whether this could be worth the effort. Such optimizations are costly, as the way you compile your applications has to be modified.

Prelink

- Prelinking reduces the time needed to start an executable
- It is extensively used on Android
- It has to be configured to know which libraries needs to be prelinked and will assign a fixed address for each available symbol thus removing the need to relocate symbols when starting an executable.
- Be careful of security implications, as executable code is always loaded at the same address.
- Code and paper at `http://people.redhat.com/jakub/prelink/`

Practical lab - Reducing time starting applications



- Trace and profile the main application with `strace`

Kernel optimizations

free electrons

Embedded Linux
Experts

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
...
```

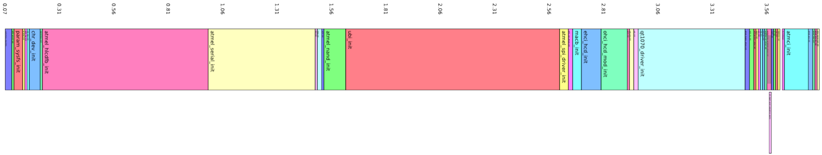
It is probably a good idea to increase the log buffer size with `CONFIG_LOG_BUF_SHIFT` in your kernel configuration. You will also need `CONFIG_PRINTK_TIME` and `CONFIG_KALLSYMS`.

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 Elixir (for example `http://elixir.free-electrons.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.

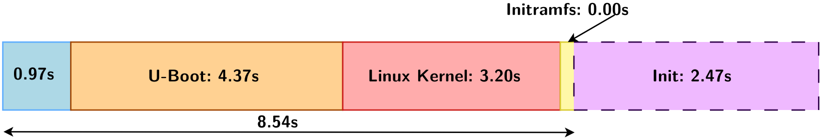
Reduce kernel size

First, we focus on reducing the size without removing features

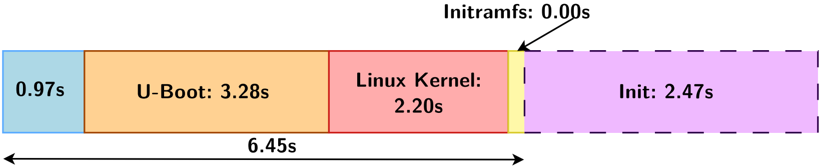
- The main mechanism is to use kernel modules
- Compile everything that is not needed at boot time as a module
- Two benefits: the kernel will be smaller and load faster, and less initialization code will get executed
- Remove features that are not used by userland: `CONFIG_KALLSYMS`, `CONFIG_DEBUG_FS`, `CONFIG_BUG`
- Use features designed for embedded systems: `CONFIG_SLOB`, `CONFIG_EMBEDDED`

Results

Before:



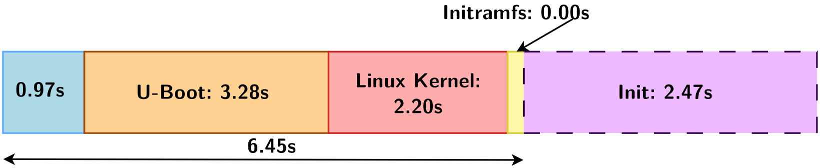
After:



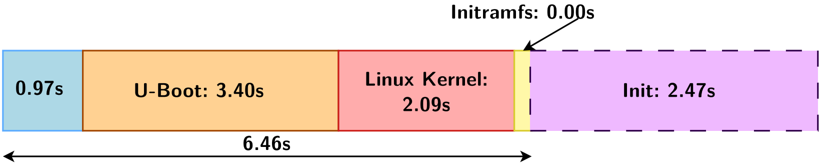
Kernel Compression

Depending on the balance between your storage reading speed and your CPU power to decompress the kernel, you will need to benchmark different compression algorithms.

Before (gzip):



After (LZO):



Conclusion: don't use LZO for now.

Deferred initcalls

- If you can't compile a feature as a module (e.g. networking or block subsystem), try `deferred_initcalls`.
- Your kernel will not shrink but some initializations will be postponed. Once your critical application is ready, you can execute the remaining initcalls.
- See http://elinux.org/Deferred_Initcalls

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:

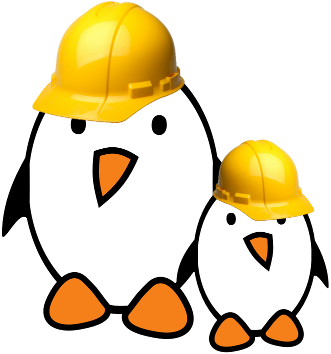
	Time	Diff
Without `lpj`	71 ms	
With `lpj`	8 ms	-63 ms

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

Multiprocessor support (SMP)

- SMP is quite slow to initialize
- UP systems may be faster to boot
- What you can try is to hotplug the other cores after your critical application has started

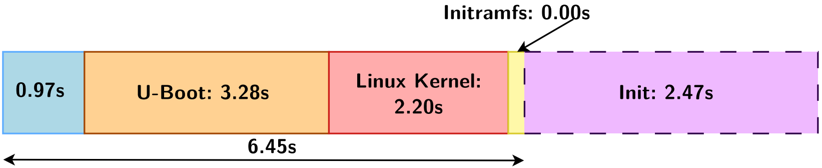
Practical lab - Reduce kernel boot time



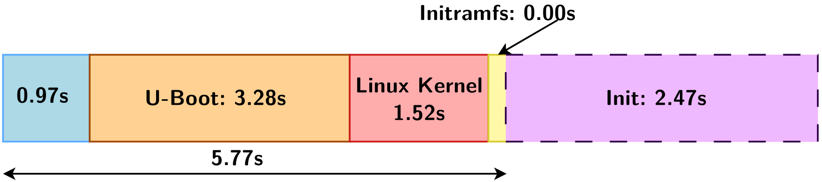
- Recompile the kernel, switching to an initramfs
- Use `initcall_debug` to find the biggest time consumers
- Reduce the number of modules
- Tune kernel command line parameters

Kernel Optimization results

Before (gzip)



After:



Without losing any functionality!

Kernel: last milliseconds (1)

To shave off the last milliseconds, you will probably want to remove unnecessary features:

- `CONFIG_PRINTK=n` will have the same effect as the `quiet` command line argument but you won't have any access to kernel messages. You will have a significantly smaller kernel though.
- Try `CONFIG_CC_OPTIMIZE_FOR_SIZE=y`. This will have an impact on performance, you will have to benchmark.

Kernel last milliseconds (2)

More features you could remove:

- Module loading/unloading
- Block layer
- Network stack
- USB stack
- Power management features
- `CONFIG_SYSFS_DEPRECATED`
- Input: keyboards / mice / touchscreens
- `CONFIG_LEGACY_PTY_COUNT` or the `pty.legacy_count` kernel parameter

Bootloader optimizations

free electrons

Embedded Linux
Experts

Corrections, suggestions, contributions and translations are welcome!

Bootloader

- Remove unnecessary functionality.
  Usually, bootloaders include many features needed only for development. Compile your bootloader with less features.
- Optimize required functionality.
  Tune your bootloader for fastest performance.
  Switch to another bootloader (if available)
  Skip the bootloader and load the kernel right away.

U-Boot 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 ...`

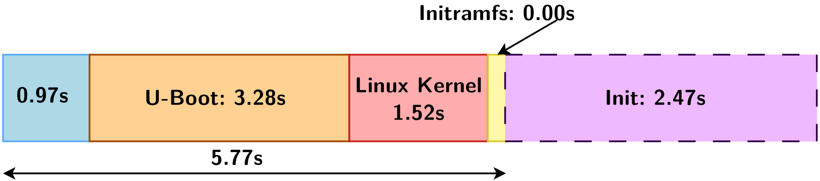
Further U-Boot optimizations

- 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.

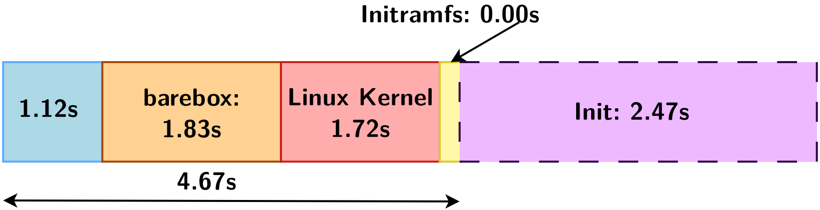
Switching to another bootloader

Switching from U-Boot to Barebox

Results with the SAMA5D3x-EK board



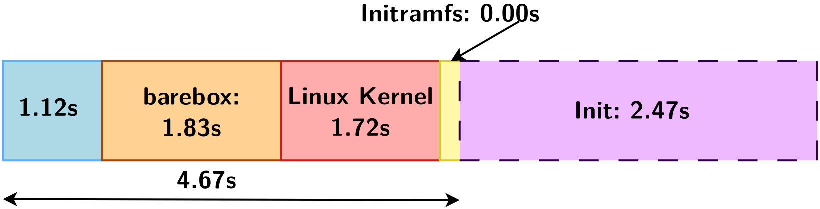
After:



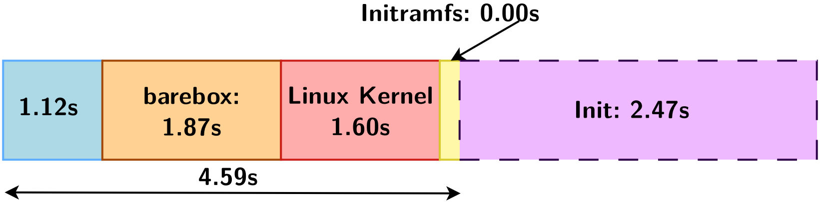
Compression

Let's try LZO compression for the kernel again:

Before (gzip)



After (LZO):



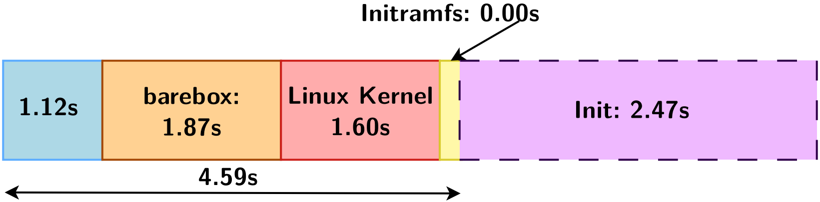
Features

Now, let's limit the features of the bootloader.
We still keep a way to interact with it when a GPIO has a given value. For example, using the `gpio_direction_input` and `gpio_get_value` commands in a script that would then start an upgrade or boot a rescue kernel.

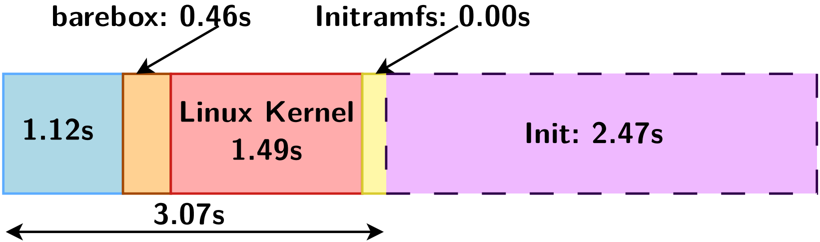
```
gpio_get_value 42
if [ $? = 0 ]; then
    kdev="/dev/nand0.krescue.bb"
fi
```

Results

Before:



After:



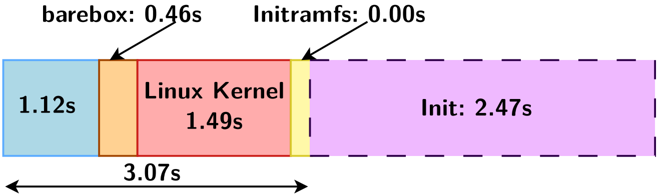
Results

Note: the kernel didn't actually change but we don't get a message on the serial line exactly at the time we switch from the bootloader to the kernel.

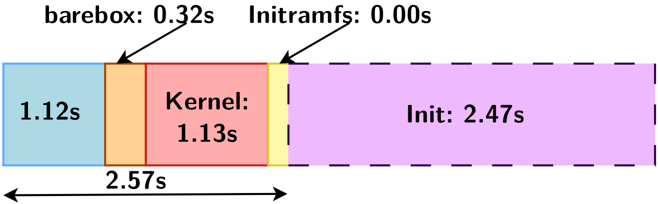
Warning: Sometimes, the kernel is relying on the bootloader to initialize the hardware (pinmuxing, clocks, ...) so be careful when removing features.

Results - Simplifying bootloader and kernel

Before:



After:



Skipping the bootloader

Removing the bootloader

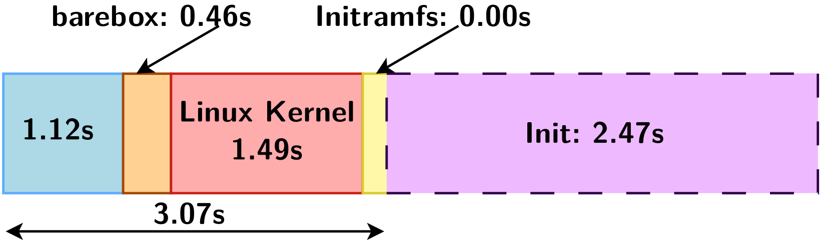
- Principle: instead of loading the bootloader and then the kernel, load the kernel right away!
- For example, on Atmel AT91, is is easy to implement with `at91bootstrap v3`. You just need to configure it with one of the `linux` or `linux_dt` configurations:

```
make at91sama5d3xeknf_linux_dt_defconfig
make
```

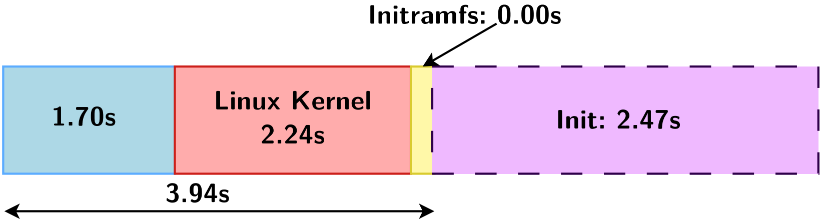
  Full details on
  http://free-electrons.com/blog/starting-linux-directly-from-at91bootstrap3/
- In our particular case, though, you will lose the main advantages of using Barebox. It uses the CPU caches while loading the kernel.

Removing the bootloader

Before:



Using `AT91bootstrap` to boot the Linux kernel:



Kernel compression options

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!

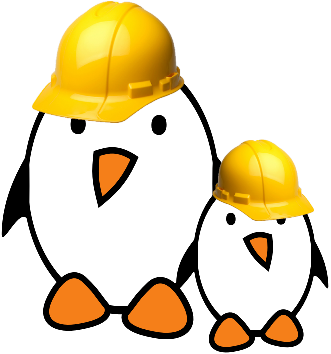
Optimize kernel for size

- `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

Practical lab - Reduce bootloader time



- Reduce boot time by using the Barebox bootloader
- Optimize Barebox

Hardware initialization

free electrons

Embedded Linux
Experts

Corrections, suggestions, contributions and translations are welcome!

Hardware initialization

The hardware needs time to initialize

- Voltage regulation, crystal stabilization
- Can be up to 200 ms
- As a software developer, you can't do anything about this part.
- All you can do is measure this time with an oscilloscope and ask the hardware board designers whether the can do anything about this. However, there are delays in the CPU which may not be possible to reduce (see the CPU datasheet).