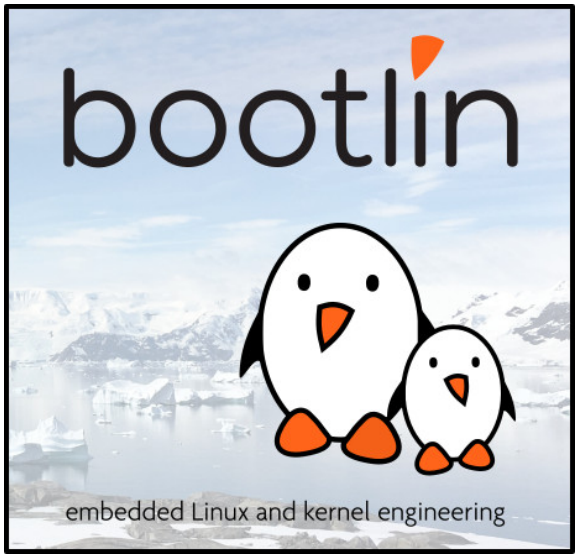
Update on boot time reduction techniques

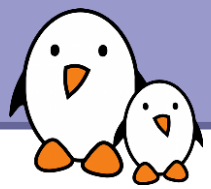
Michael Opdenacker
Bootlin



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

Our test hardware

TNY-A9260 board from CALAO Systems

- AT91SAM9260 CPU at 180 MHz
- 64 MB of RAM
- 256 MB of NAND flash storage
- Serial port
- USB device port (used for networking)
- Expansion port (Ethernet, SPI...)

Our test system

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

Boot time components

- Bootstrap (at91bootstrap)
- Bootloader (U-boot)
- Linux
- User space (mount jffs2, BusyBox http)

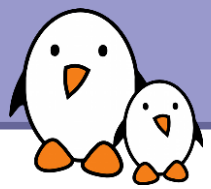
Measuring kernel boot time

CONFIG_PRINTK_TIME

- Configure it in the Kernel Hacking section.
- Adds timing information to kernel messages. Simple and robust.
- Not accurate enough on some platforms (1 jiffy = 10 ms on arm!)

See http://elinux.org/Printk_Times

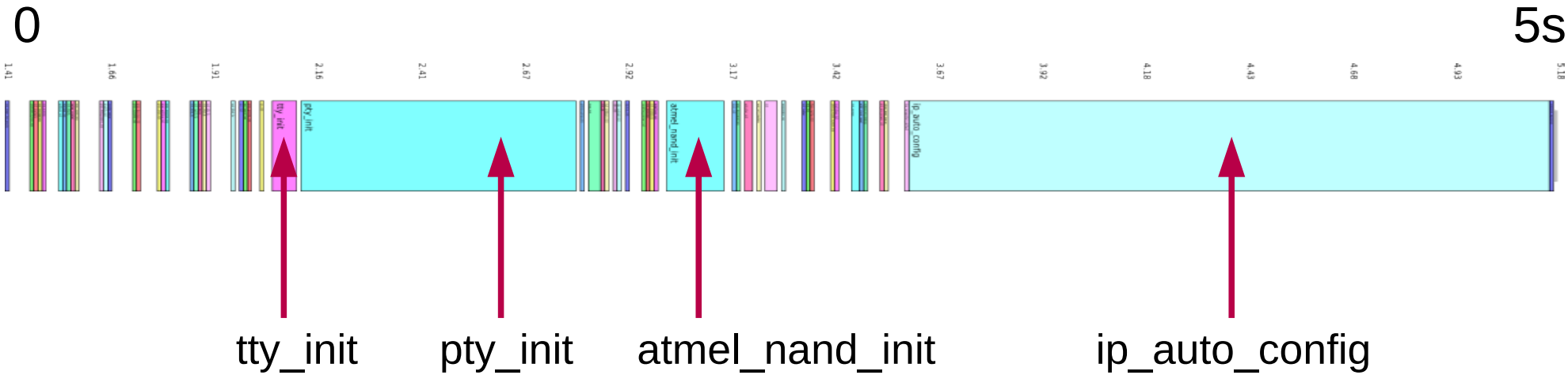
```
...
[42949372.970000] Memory: 64MB = 64MB total
[42949372.970000] Memory: 54784KB available (1404K code, 296K data, 72K init)
[42949373.180000] Mount-cache hash table entries: 512
[42949373.180000] CPU: Testing write buffer coherency: ok
[42949373.180000] checking if image is initramfs...it isn't (bad gzip magic numb
ers); looks like an initrd
[42949373.200000] Freeing initrd memory: 8192K
[42949373.210000] NET: Registered protocol family 16
...
```



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

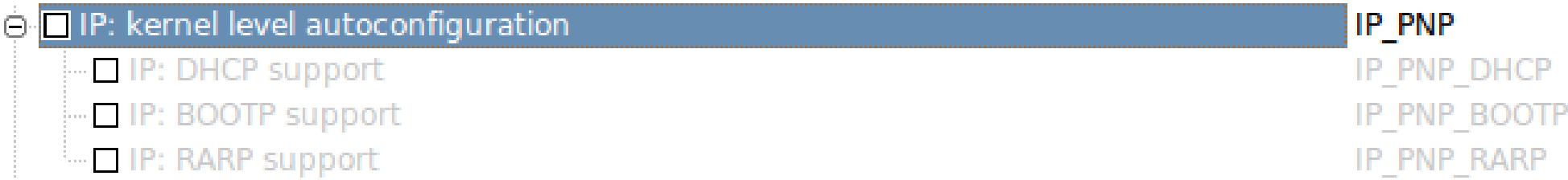


Grabserial

- 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 (bootloader), and doesn't just start when the kernel initializes.
- Another advantage: no overhead on the target, because run on the host machine.

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)



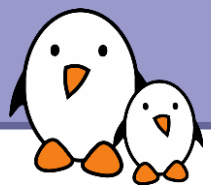
Reducing the number of PTYs

- 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.

New jffs2 features

- `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!



JFFS2 without compression

- Possible to disable compression.
  That's what we tried, as all our files (JPG photos) can't be compressed.
- Unfortunately, we just saved 0.03 s!
  JFFS2 probably gives up compressing when a file can't be compressed.

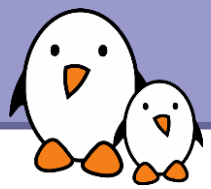
☑ Advanced compression options for JFFS2	JFFS2_COMPRESSION_OPTIONS
☐ JFFS2 ZLIB compression support	JFFS2_ZLIB
☐ JFFS2 LZO compression support (NEW)	JFFS2_LZO
☐ JFFS2 RTIME compression support	JFFS2_RTIME
☐ JFFS2 RUBIN compression support (NEW)	JFFS2_RUBIN
☑ JFFS2 default compression mode	
◉ no compression	JFFS2_CMODE_NONE
○ priority	JFFS2_CMODE_PRIORITY
○ size (EXPERIMENTAL)	JFFS2_CMODE_SIZE
○ Favour LZO	JFFS2_CMODE_FAVOURLZO

Preset loops_per_jiffy

- 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 and the result never changes! Find the `lpj` value in kernel boot messages (if you don't get it in the console, boot Linux with the `loglevel=8` parameter). Example:

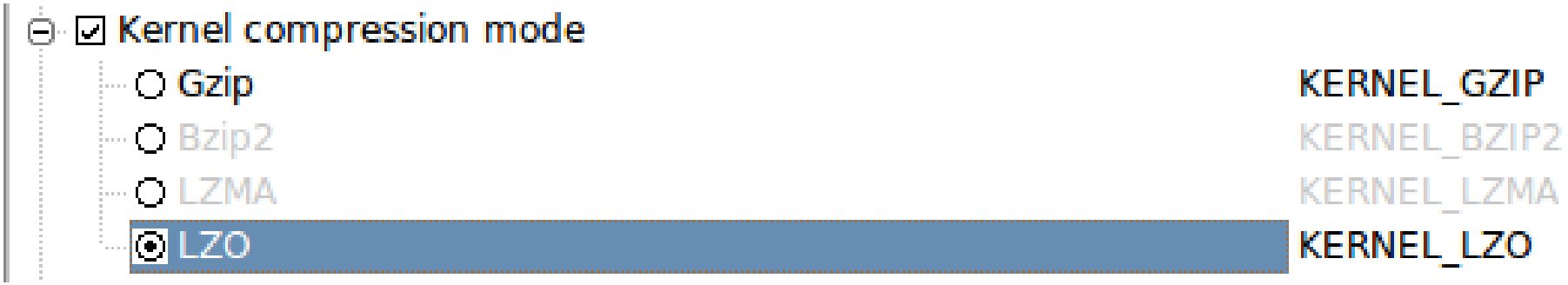
```
Calibrating delay loop... 99.73 BogoMIPS (lpj=498688)
```

- At the next boots, start Linux with the below option: `lpj=<value>`
- It saved us 0.18 s



LZO kernel decompression

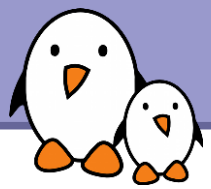
- LZO is a compression algorithm that is much faster than gzip, at the cost of a slightly degrade compression ratio (+10%).
- It was already in use in the kernel code (JFFS2, UBIFS...)
- Albin Tonnerre from Bootlin added support for LZO compressed kernels. His patches are waiting for inclusion in mainstream Linux. Get them from http://lwn.net/Articles/350985/



LZO decompression results

- Saves approximately 0.25 s of boot time
  See https://bootlin.com/blog/lzo-kernel-compression/
- Our patch also allows LZO to be used for initramfs decompression (`CONFIG_INITRAMFS_COMPRESSION_LZO=y`)
- Another solution is to use an uncompressed kernel (another patch will be sent), in which case kernel execution is just marginally faster than with LZO, at the expense of a double size.

	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**



Directly boot Linux from bootstrap code

- Idea: make a slight change to at91bootstrap 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 (SPI dataflash, for example).
  In this case, you can keep U-boot, but remove all the features not needed in production (USB, Ethernet, tftp...)
- Time savings: about 2 s

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

Disable 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

Results

- Initial boot time: 38 s
- Final boot time: approximately 4 s
- Other techniques can be used to reduce boot time even further!

Reduce the kernel size

Through the `CONFIG_EMBEDDED` option

- Remove things that are not needed in your dedicated system (features, debugging facilities and messages)
- Make sure you have no unused kernel drivers
- Disable support for loadable kernel modules and make all your drivers static (unless there are multiple drivers than can be loaded later).
- A smaller kernel is faster to load
- A simpler kernel executes faster

Optimize RC scripts

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

- Remove services you don't need (ssh), or start them later.
  Use static device files (no udev or mdev).
- Start your services directly from a single startup script.
  This eliminates multiple calls to `/bin/sh`.
- This saves tens of seconds with root filesystems generated with OpenEmbedded (for example).

Shells: reducing forking

- `fork` / `exec` system calls are very heavy.
  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

Use faster filesystems

Run faster by using the most appropriate filesystems!

- Compressed read-only filesystem (block device): use SquashFS (http://squashfs.sourceforge.net) instead of CramFS (much slower, getting obsolete).
- NAND flash storage: you should try UBIFS (http://www.linux-mtd.infradead.org/doc/ubifs.html), the successor of JFFS2. It is much faster. You could also use SquashFS. See our Choosing filesystems presentation (https://bootlin.com/docs/filesystems).

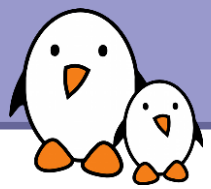
Boot from a hibernate image

The ultimate technique for instant boot!

- In development: start the system, required applications and the user interface. Hibernate the system to disk / flash in this state.
- In production: boot the kernel and restore the system state from with this predefined hibernation image.
- This way, you don't have to initialize the programs one by one. You just get the back to a valid state.
- Used in Sony cameras to achieve instant power on time.
- Unlike Suspend to RAM, still allows to remove batteries!

Use a profiler

- Using a profiler can help to identify unexpected behavior degrading application performance.
- For example, a profiler can tell you in which functions most of the time is spent.
- Possible to start with strace and ltrace
- Advanced profiling with Valgrind: http://valgrind.org/
  - Compile your application for x86 architecture
  - You can then profile it with the whole Valgrind toolsuite:
    Cachegrind: sources of cache misses and function statistics.
    Massif: sources of memory allocation.
- See Embedded Linux system development course for details:
  https://bootlin.com/training/embedded-linux/



Other ideas

- Copy kernel and initramfs from flash to RAM using DMA
  (Used by MontaVista in Dell Latitude ON)
- Compile drivers as modules for devices not used at boot time. This reduces time spent initializing drivers.
  A smaller kernel is also faster to copy to RAM.
- Fast boot, asynchronous initcalls:
  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.

See http://elinux.org/Boot_Time for more resources

Other ideas

- Bootchart Lite: a lightweight bootchart implementation
  http://code.google.com/p/bootchart-lite/
- Timechart from Arjan van de Ven:
  http://blog.fenrus.org/?p=5
  See http://elinux.org/Bootchart#Timechart
- Use statically linked applications
  (less CPU overhead, less libraries to load)
- Use deferred initcalls
  See http://elinux.org/Deferred_Initcalls
- NAND: just check for bad blocks once
  Atmel: see http://patchwork.ozlabs.org/patch/27652/

Useful resources

- See the “How we got a 3D application booting in 5 seconds” presentation from Grégory Clément and Simon Polette
  http://tree.celinuxforum.org/CelfPubWiki/ELCEurope2009Presentations