Boot time optimizations

Alexandre Belloni
Adeneo Embedded

Latest update: November 6, 2012.

Yet another boot time optimizations talk !

Yet another boot time optimizations talk !

Who is Adeneo Embedded?

- BSP and driver development
- Hardware Design and design reviews
- Systems optimization
- Embedded application development
- Support contract
- Training and Workshop
- Consulting and engineering services

Use case

- Automotive customer wants to boot fast
- Platform is a Freescale i.mx53
- Requirement is to reply to a CAN message in less than 500ms
- Actually, we have 500ms from **power on** to the reply
  - The board is powered on only after a first CAN message is received.
  - This is important, the SoC is taking about 120ms before being able to execute the first instruction.
- Customer wants to keep all the functionnalities of the kernel

Another use case

- Booting an OpenGL application as fast as possible
- Platform is a Freescale i.mx6q
- For added difficulty, we can't modify the application

Where do we stand ?

- From the original Freescale rootfs, we take:
  - About 20s to start a custom application on the i.mx53
  - Between 15s and 53s to start an OpenGL application on the i.mx6q

What did we try ?

What did we try ?

Usual optimizations

- lpj
  - If your products are similar, you can set lpj
  - You will gain about 250 ms by skipping the calibration loop
- stripping init
  - Start your critical application as soon as possible: runlevel 1, rcS or even better from inittab.
  - An idea may be to try to use your critical application as init

Usual optimizations, continued

- flash storage:
  - NAND, NOR, SD
  - We usually get really good performance booting from SD cards, class 4 or class 6, but you will still have to benchmark
- toolchains
  - Not all toolchains are created equal. Changing toolchains, will usually make you gain the last hundred of ms

Usual optimizations, kernel

- Remove as many features as you can from the kernel. It as two consequences: the smaller, the faster to copy from storage to RAM and less features means less initializations
- Not always what you want, in particular, our customer wanted to still have a fully featured kernel.
- In particular, pay attention to:
  - console port and serial output, who needs that ?
  - `printk` and DEBUGFS
  - try `CONFIG_CC_OPTIMIZE_FOR_SIZE=y`
  - SLOB memory allocator
  - KALLSYMS

What about functionnalities ?

- Stripping a lot of features from the kernel is not always possible
- You can use modules !
  - Load them when your critical application is ready
  - but it is not always possible to compile as module (example: networking)
- So we also used `deferred_initcalls`
  - Your kernel will still grow
  - but it won't execute some initializations until you tell it to from userland
  - Once your critical application is started, "continue" booting to a fully featured kernel
  - see `http://elinux.org/Deferred_Initcalls`

Kernel compression

- None
- Gzip
- LZO

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

last kernel optimizations

- try to play with `mem=` on the cmdline
- the less RAM you need to initialize, the faster you will boot

Custom bootloader

- we completely got rid of u-boot
- we started of with arm-kernel-shim
- on Freescale platforms, it is just a matter of configuring a few register then passing atags

Custom bootloader: a warning

- the kernel may not initialize every device, a lot is still left to the bootloader (quite often u-boot)
- should we migrate everything to the kernel ?
- Join us at 16:15 in Zafir for a BoF !

Root filesystem

- There are multiple challenges when building your root filesystem
  - You may not be able to optimize the customer's application (it was the case on the i.mx53)
  - Even worse, you may not have the sources (it is the case on i.mx6, you can't get the sources of the HW acceleration librairies)
  - you may need a lot of applications and dependencies so, you may not be able to reduce the size of your rootfs

Root filesystem

- Use an initramfs, but use it right !
  - it only does what we need to do quickly
  - we used uClibc
  - we used `mklibs` to further strip the libs
  - then, we `switch_root` to the final filesystem

Our final solutions

i.mx53

- boot from SDcard
- with a custom bootloader
- stripped down kernel
- start a custom init:
  - launches the critical application (in that case, a CAN daemon)
  - calls `deferred_initcalls`
  - `switch_root` to the final filesystem and `exec` the final init

i.mx6q

- boot from SDcard
- with a custom bootloader
- stripped down kernel
- start a custom init that launches the OpenGL application
- rootfs has been stripped using `mklibs`
- final image including bootloader, kernel, rootfs is 5.2MB, including 3MB only for HW acceleration support. No compression

Final results

- the CAN message is ready to be received and replied to in about 360ms
- the OpenGL application is started in 720ms from power on, 590ms from reset

What we didn't try... yet

What we didn't try... yet

EDO

- patch sent on sept 11th
- improves reading on NAND by a factor of 6

DMA

- an interesting idea is to load the kernel from storage to RAM using DMA
- the kernel can then start to decompress itself while being loaded to RAM

initcalls reordering

- it is possible to reorder initcalls by changing makefiles
- on SMP, we may also try to run initcalls in parallel

Demo time !

Demo time !

Demonstration

- I have a live demonstration

Demonstration

- I have a live demonstration
- Just in case, I have a video.

What's next ?

- Let's try to capture, decode and display a video in less than 1s

Show me the code !

- Code for the bootloader is at `https://github.com/alexandrebelloni/whoosh`
- Some patches to the kernel are not yet upstreamed
- our main source was `http://www.elinux.org/Boot_Time` but we believe it needs some updates

We want more

- We really feel there is a need for that kind of bootloaders
- We are focusing on being as small and as fast as possible
- We'd like to support TI
- but we don't care about being generic

What about the kernel ?

- As said, a lot of initializations are still done in the bootloader
- Do we want to keep everything there
- or should the kernel be able to do everythng on its own ?

Suggestions ?

- Can we go faster ?
- Do you have any suggestion ?

Questions ?

- Any questions ?