How we got a 3D application booting in 5 seconds under Linux
Agenda

Embedded Experts dedicated to the success of your design

- The Problem
- The context
- The boot process
- Time measurement
- Techniques for optimizations
- Results
Embedded Experts dedicated to the success of your design

• About Adeneo
  – French group
  – Foreign subsidiaries
    • USA, China, Africa
  – Developing and manufacturing electronics products
  – Developing embedded softwares
    • Application software
    • Drivers development, BSP (Linux, CE)
  – Works with embedded Linux for + 7 years
    • Involved in the community (ex: Xenomai, RTAI)
    • Interested in improving bootup time in our systems
Introduction

Embedded Experts dedicated to the success of your design

• About us
  – Grégory Clément
  – Simon Polette
The problem

- People don't want to wait
- Especially with embedded devices
Embedded Experts dedicated to the success of your design

Hardware

- AT92SAM9261-EK board:
  - ARM9-based µC at 200MHz
  - 64MB SDRAM
  - 256MB NAND Flash
  - DataFlash (SPI)
Embedded Experts dedicated to the success of your design

The boot process
The boot process

First stage bootloader

Second stage bootloader

Kernel

Init

Bootstrap

U-Boot

Linux

User-space
The boot process

Embedded Experts dedicated to the success of your design

- Bootstrap:
  - Basic hardware init. (GPIO, Clock, SDRAM, etc)
  - Load and start U-Boot

- U-Boot:
  - Hardware initializations
  - Load and start kernel

- Kernel:
  - Drivers initializations
  - Mount root filesystem

- Init:
  - First User-space processus
  - Launch RC scripts
  - Start graphic environment
Time measurement
Time measurement

• Find where the system loose time
• Be able to trace the boot process
• Identify the worst problem first
Time measurement

Embedded Experts dedicated to the success of your design

• Global measurement
  – serialchrono
• Kernel measurement
  – printk-times
  – initcall_debug
• User-space measurement
  – Bootchart-lite
Time measurement

Global measurement

Embedded Experts dedicated to the success of your design
Serialchrono

- Simple script
- Capture serial output, add time stamp at beginning of each line, then echo it.
- No need for any other module to run it, just /bin/sh
- Usage:

  ```
  ./serialchrono TTYS="/dev/ttyS0"
  ```
Serialchrono output example:

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.252419</td>
<td>Area 0: C0000000 to C00041FF (RO) Bootstrap</td>
</tr>
<tr>
<td>1.266907</td>
<td>Area 1: C0004200 to C00083FF Environment</td>
</tr>
<tr>
<td>1.274590</td>
<td>Area 2: C0008400 to C0041FFF (RO) U-Boot</td>
</tr>
<tr>
<td>1.281892</td>
<td>Area 3: C0042000 to C0251FFF Kernel</td>
</tr>
<tr>
<td>1.289026</td>
<td>Area 4: C0252000 to C083FFFF FS</td>
</tr>
<tr>
<td>1.296206</td>
<td>In: serial</td>
</tr>
<tr>
<td>1.303194</td>
<td>Out: serial</td>
</tr>
<tr>
<td>1.310185</td>
<td>Err: serial</td>
</tr>
<tr>
<td>1.317642</td>
<td>Hit any key to stop autoboot: 0</td>
</tr>
<tr>
<td>1.984084</td>
<td>## Booting kernel from Legacy Image at 22000000 ...</td>
</tr>
<tr>
<td>1.992148</td>
<td>Image Name: Linux-2.6.31-rc2</td>
</tr>
<tr>
<td>1.999412</td>
<td>Image Type: ARM Linux Kernel Image (uncompressed)</td>
</tr>
<tr>
<td>2.007201</td>
<td>Data Size: 911700 Bytes = 890.3 kB</td>
</tr>
<tr>
<td>2.013851</td>
<td>Load Address: 20008000</td>
</tr>
<tr>
<td>2.021067</td>
<td>Entry Point: 20008000</td>
</tr>
<tr>
<td>2.148271</td>
<td>Verifying Checksum ... OK</td>
</tr>
<tr>
<td>2.268042</td>
<td>Loading Kernel Image ... OK</td>
</tr>
<tr>
<td>2.275552</td>
<td>OK</td>
</tr>
<tr>
<td>2.282747</td>
<td></td>
</tr>
<tr>
<td>2.289796</td>
<td>Starting kernel ...</td>
</tr>
</tbody>
</table>
Serialchrono

Embedded Experts dedicated to the success of your design

• Advantages :
  – Very simple to use, no need installation
  – "Passive" instrumentation (just read what the target send)
  – Doesn't add any overhead

• Disadvantages :
  – First printks are queued up and printed only once serial console have been initialized
  – Timestamp only what is printed in console
Time measurement

Embedded Experts dedicated to the success of your design

Kernel measurement
Kernel measurement

- Printk-time
  - Add timestamp at the beginning of each console output line
  - To activate it, just select the option in kernel configuration, or just add `printk.time=1` in the kernel command line

```bash
# Kernel hacking
#
CONFIG_PRINTK_TIME=y
```
Kernel measurement

Embedded Experts dedicated to the success of your design

• Printk-time output example:

```plaintext
[ 0.000000] NR_IRQS:192
[ 0.000000] AT91: 96 gpio irqs in 3 banks
[ 0.000000] Console: colour dummy device 80x30
[ 0.000000] console [ttyS0] enabled
[ 0.120000] Calibrating delay loop (skipped) preset value.. 124.51 BogoMIPS (lpj=622592)
[ 0.130000] Mount-cache hash table entries: 512
[ 0.130000] CPU: Testing write buffer coherency: ok
[ 0.140000] NET: Registered protocol family 16
[ 0.180000] bio: create slab <bio-0> at 0
[ 0.190000] SCSI subsystem initialized
[ 0.200000] usbcore: registered new interface driver usbfs
[ 0.200000] usbcore: registered new interface driver hub
[ 0.210000] usbcore: registered new device driver usb
[ 0.230000] NET: Registered protocol family 2
[ 0.230000] IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
[ 0.240000] TCP established hash table entries: 2048 (order: 2, 16384 bytes)
```

• Note:
  – Timestamps are available only once clock is initialized
initcall_debug

- Measure time spend in each 'initcall' during boot
- Usage:
  - Simply add 'initcall_debug' to kernel command line
- Result will be available in dmesg
initcall_debug

Embedded Experts dedicated to the success of your design

• Exemple of output :
  – dmesg | egrep 'initcall|calling'

```plaintext
[ 1.020000] calling chr_dev_init+0x0/0xbc @ 1
[ 1.030000] initcall chr_dev_init+0x0/0xbc returned 0 after 5047 usecs
[ 1.030000] calling firmware_class_init+0x0/0x78 @ 1
[ 1.040000] initcall firmware_class_init+0x0/0x78 returned 0 after 495 usecs
[ 1.040000] calling sysctl1_core_init+0x0/0x44 @ 1
[ 1.050000] initcall sysctl1_core_init+0x0/0x44 returned 0 after 38 usecs
[ 1.060000] calling inet_init+0x0/0x1dc @ 1
[ 1.100000] initcall inet_init+0x0/0x1dc returned 0 after 35788 usecs
[ 1.100000] calling af_unix_init+0x0/0x5c @ 1
[ 1.110000] initcall af_unix_init+0x0/0x5c returned 0 after 4388 usecs
[ 1.120000] calling populate_rootfs+0x0/0x210 @ 1
[ 1.120000] initcall populate_rootfs+0x0/0x210 returned 0 after 495 usecs
[ 1.130000] calling timer_init_sysfs+0x0/0x40 @ 1
[ 1.140000] initcall timer_init_sysfs+0x0/0x40 returned 0 after 920 usecs
[ 1.140000] calling fpe_init+0x0/0x84 @ 1
[ 1.150000] initcall fpe_init+0x0/0x84 returned 0 after 6369 usecs
```
Possibility to change it into a pretty readable graph thanks to Arjan van de Ven's script:

- dmesg | perl scripts/bootgraph.pl > boot.svg
initcall_debug

- Notes about initcall_debug:
  - Increase significantly boot time
  - Log buffer could be too small
    - Increase it size by changing CONFIG_LOG_BUF_SHIFT value in the .config
- Printk-time must be enabled to get the graph
Time measurement

Embedded Experts dedicated to the success of your design

User-space measurement
Embedded Experts dedicated to the success of your design

Bootchart-lite

• http://code.google.com/p/bootchart-lite
• Similar to Bootchart
  – http://www.bootchart.org
• Collects datas from /proc
• Written in C
  – Less overhead than a shell script
• Use the same Java parser than Bootchart to change the logs into a nice graph
Bootchart-lite

Embedded Experts dedicated to the success of your design

- Usage:
  - Launch bootchart-lite instead of traditional /sbin/init
  - Make a gzipped tarball of logs
    - `tar -cvzf bootchart.tgz /etc/bootchart-lite/*`
  - Create the graph using bootchart's java parser:
    - `java -jar ./bootchart.jar bootchart.tgz`
Bootchart-lite

- Bootchart exemple:
Notes about bootchart-lite:

- The bootchart daemon could be stopped manually:
  - Send `SIGUSR1` to `bootchartd`
- Or automatically:
  - `#define EXIT_PROC "proc_name"
- Can easily modify the sample time
Techniques for optimization
Techniques for optimizations

Embedded Experts dedicated to the success of your design

- Our environment
  - Atmel AT91SAM9261-EK board
  - 2.6.29 Kernel
  - U-Boot v2009.06
  - at91sam9261ek_defconfig
  - JFFS2 filesystem (on 122MB partition)
  - Qt for embedded Linux graphic application
    - Start **30 seconds** after the board have been powered up
Techniques for optimizations

Embedded Experts dedicated to the success of your design

• Initial timing:

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Init</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>application</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>4.2</td>
<td>12.6</td>
<td>6.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.8</td>
</tr>
</tbody>
</table>

Time distribution

- Application
- Init
- Kernel
- U-Boot
- Bootstrap
Techniques for optimizations

Embedded Experts dedicated to the success of your design

User-Space optimizations
User-Space optimizations

- Use UBIFS instead JFFS2
  - Much faster for mounting, writing, reading
  - Will improve both kernel and user-space initialization time
  - User-space applications will be loaded faster
User-Space optimizations

*Results*
– Boot time: **14.5s**
– Saving: **15.5s**

![Time distribution graph]

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>0.1</td>
<td>4.2</td>
<td>3.8</td>
<td>4.5</td>
</tr>
<tr>
<td>application</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

JFFS2

UBIFS

UBIFS: 0 5 10 15 20 25 30 seconds

Application
Init
Kernel
U-Boot
Bootstrap
User-Space optimizations

- Remove unneeded rc scripts
  - For example: No need to start `udev` daemon (hardware configuration in embedded systems is often constant)
  - Keep only the scripts which are essential for your application
  - Still possible to start the other scripts later
Embedded Experts dedicated to the success of your design

User-Space optimizations

• Results
  – Boot time: **10.5s**
  – Saving: **4s**

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.1</td>
<td>4.2</td>
<td>3.8</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.9</td>
</tr>
</tbody>
</table>

Time distribution

- Application
- Init
- Kernel
- U-Boot
- Bootstrap
User-Space optimizations

**Embedded Experts dedicated to the success of your design**

- Dynamically linked applications take a long time to get loaded
- Use statically linked application

Note: The binary size can increase significantly

- In our example:
  - Dynamically linked: 152KB
  - Statically linked: 12MB
User-Space optimizations

Embedded Experts dedicated to the success of your design

• Results
  – Boot time: **9.1s**
  – Saving: **1.4s**

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.1</td>
<td>4.2</td>
<td>3.8</td>
<td>0.5</td>
</tr>
<tr>
<td>Init</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time distribution

- **Application**
- **Init**
- **Kernel**
- **U-Boot**
- **Bootstrap**
User-Space optimizations

- Other solutions for optimisation
  - /sbin/init alternatives
    - Initng
    - Upstart
      - Not suitable for small systems
      - Upstart needs additional libraries such as Dbus
  - Better to optimize init process "by hand"
Techniques for optimizations

Embedded Experts dedicated to the success of your design

Kernel optimizations
Kernel optimizations

- Identify what is essential for your system just after boot
- Remove unused features
- Use deferred modules
  - Initialize modules after the system completely booted, by doing

  \[
  \text{echo } 1 \text{ } > \text{/proc/deferred\_initcalls}
  \]
Kernel optimizations

- Best strategy to configure the kernel?
  - Don't use 'make yourboard_defconfig'
  - Do 'make allnoconfig'
  - Then select the minimum required by your system
Kernel optimizations

- Results
  - Boot time: **7.1s**
  - Saving: **2s**

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Init</td>
</tr>
<tr>
<td>before</td>
<td>0.1</td>
<td>4.1</td>
<td>1.9</td>
<td>0.5</td>
</tr>
<tr>
<td>after</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time distribution
Kernel optimizations

- Set 'CONFIG_EMBEDDED=y'
- [ * ] Configure standard kernel features
  - Allow disabling more (unused) features in the kernel
  - Reduce the kernel size
Kernel optimizations

Embedded Experts dedicated to the success of your design

• Results
  – Boot time : 6.6s
  – Saving : 500ms

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Init</td>
</tr>
<tr>
<td>before</td>
<td>0.1</td>
<td>3.9</td>
<td>1.6</td>
<td>0.5</td>
</tr>
<tr>
<td>after</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time distribution

- Application
- Init
- Kernel
- U-Boot
- Bootstrap
Kernel optimizations

• Make the kernel boot quietly
  – Disable the printk messages on the console during the kernel boot.

• Usage
  – Simply add 'quiet' in the kernel command line
    • Ex:
      `root=/dev/mtdblock1 console=ttyS0,115200 rootfstype=jffs2 quiet`

• Time saved will be proportional to number lines printed on the console during boot
• Still possible to consult the boot log in dmesg
Kernel optimizations

Results
- Boot time: **6.3s**
- Saving: **300ms**

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.1</td>
<td>3.9</td>
<td>1.3</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>application</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Time distribution**

Before:
- Application: 0
- Init: 1
- Kernel: 2
- U-Boot: 3
- Bootstrap: 4

After:
- Application: 0
- Init: 1
- Kernel: 2
- U-Boot: 3
- Bootstrap: 4
Kernel optimizations

- Preset 'loop per jiffies'
  - Constant value for the architecture, but calculated at each boot
  - Possibility to give this value to the kernel
- Usage:
  - Add 'lpj=<value>' in the kernel command line
Kernel optimizations

Embedded Experts dedicated to the success of your design

- Results
  - Boot time: 6.1s
  - Saving: 200ms

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>init</td>
<td>application</td>
<td></td>
<td></td>
</tr>
<tr>
<td>before</td>
<td>0.1</td>
<td>3.9</td>
<td>1.1</td>
<td>0.5</td>
</tr>
<tr>
<td>after</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time distribution
Kernel optimizations

- BBT is created at each boot for NAND devices, and stored in RAM
  - Use On Flash BBT
- We made a patch for Atmel nand driver
  - Now merged into the mainline (2.6.30 kernel)
- Usage
  - Add `atmel_nand.on_flash_bbt=1` in the kernel command line
Kernel optimizations

Embedded Experts dedicated to the success of your design

• Results
  – Boot time: 6s
  – Saving: 100ms

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Init</td>
</tr>
<tr>
<td>before</td>
<td>0.1</td>
<td>3.9</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>after</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kernel optimizations

• Other solutions for optimisation
  – Use new 'fastboot' technologie
    • In the mainline since 2.6.30
    • Allow asynchronous initcalls
    • Usefull if there are CPU and/or disk idle during the kernel boot
Techniques for optimizations

Embedded Experts dedicated to the success of your design

U-Boot optimizations
U-Boot optimizations

- Customize U-Boot for doing only what is necessary to launch the kernel
- For example in our system we don't need to use these devices in U-Boot
  - Ethernet device
  - Nand device
- We keep LCD display since it consume a very few time (~20ms)
U-Boot optimizations

Embedded Experts dedicated to the success of your design

- Results
  - Boot time: **4.8s**
  - Saving: **1.2s**

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Initial</td>
</tr>
<tr>
<td>Before</td>
<td>0.1</td>
<td>2.7</td>
<td>1.0</td>
<td>0.5</td>
</tr>
<tr>
<td>After</td>
<td></td>
<td></td>
<td></td>
<td>Initial</td>
</tr>
</tbody>
</table>

**Time distribution**

Before:
- Application: 0.1s
- Init: 2.7s
- Kernel: 1.0s
- U-Boot: 0.5s
- Bootstrap: 0.5s

After:
- Application: 0.1s
- Init: 2.7s
- Kernel: 1.0s
- U-Boot: 0.5s
- Bootstrap: 0.5s
Techniques for optimizations

Embedded Experts dedicated to the success of your design

Bootstrap optimizations
The bootstrap do some hardware initializations, then launch the kernel.

Official Atmel bootstrap offer two choice for initialize master clock: 100MHz and 125MHz.

- Upgrade to 125MHz will cause the CPU to run faster.
U-Boot optimizations

Embedded Experts dedicated to the success of your design

• Results
  – Boot time: **4.4s**
  – Saving: **400ms**

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Init</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>2.6</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>application</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.4</td>
</tr>
</tbody>
</table>

Time distribution

![Diagram showing time distribution](image)
Techniques for optimizations

Embedded Experts dedicated to the success of your design

Now, have a look at how much time we saved
### Techniques for optimizations

**Embedded Experts dedicated to the success of your design**

<table>
<thead>
<tr>
<th></th>
<th>Bootstrap</th>
<th>U-Boot</th>
<th>Kernel</th>
<th>User-space</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Init</td>
<td></td>
</tr>
<tr>
<td>Before optimization</td>
<td>0.1</td>
<td>4.2</td>
<td>12.6</td>
<td>6.3</td>
<td>30</td>
</tr>
<tr>
<td>After optimization</td>
<td>0.1</td>
<td>2.6</td>
<td>0.9</td>
<td>0.4</td>
<td>4.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>application</td>
<td></td>
</tr>
</tbody>
</table>

**Time distribution**

- **Before optimization**:
  - Application: 20 seconds
  - Init: 10 seconds
  - Kernel: 15 seconds
  - U-Boot: 5 seconds
  - Bootstrap: 0 seconds
- **After optimization**:
  - Application: 5 seconds
  - Init: 0 seconds
  - Kernel: 2 seconds
  - U-Boot: 2 seconds
  - Bootstrap: 0 seconds

---

**Adeneo**