Embedded Linux Optimization Techniques: How Not To Be Slow?

Benjamin Zores
ELCE 2010 – 26th October 2011 – Prague, Czech Republic
Embedded Linux Optimizations Techniques: How Not To Be Slow?

About Me ...

**ALCATEL LUCENT**

**SOFTWARE ARCHITECT**

- Expert and Evangelist on Open Source Software.
- 8y experience on various multimedia/network embedded devices design.
- From low-level BSP integration to global applicative software architecture.

**OPEN SOURCE**

**PROJECT FOUNDER, LEADER AND/OR CONTRIBUTOR FOR:**

- OpenBricks: Embedded Linux cross-build framework.
- GeeXboX: Embedded multimedia HTPC distribution.
- Enna: EFL Media Center.
- MPlayer: Linux media player application.

**EMBEDDED LINUX CONFERENCE**

**FORMER EDITIONS SPEAKER**

- ELC 2010: GeeXboX Enna: Embedded Media Center.
Embedded Linux Optimizations Techniques: How Not To Be Slow?

About My Job ...

From our “IP Touch” IP phone ...

- MIPS32 @ 275 MHz.
- 8/16 MB RAM, 4/8/16 MB NOR.
- Physical keys input.
- Basic 2D framebuffer display.
- Powered by VxWorks OS.

... to next-generation enterprise IP phones.

- Brainstorming exercise from our R&D Labs.
- Introduced as a proof-of-concept feasibility study, allowing us to explore modern Linux technologies.

- Early Requirements:
  - Powered by GNU/Linux OS, not Android.
  - Open to HTML/JS-based WebApps.
  - Remaining parts are open to imagination.
Embedded Linux Optimizations Techniques: How Not To Be Slow?
What You May Expect ...

• **Return of experience from feasibility study:**
  - You may want to see this presentation as one big exercise.
  - It won’t help you boost your system (sorry folks 😞).
  - But hopefully it’ll prevent you from facing some common troubles.

• **Share a few tips and tricks for:**
  - Correctly choosing your hardware.
  - Wisely selecting your software architecture and components.
  - Measuring and profiling your system.
  - Isolating the performances bottlenecks.
  - Optimizing your Linux embedded system.

• **Ultimately, avoid your software to be slow by design.**
Embedded Linux Optimizations Techniques: How Not To Be Slow?

Preamble

In 20 years (from my i286 Desktop to my Core i5 laptop):

• My CPU got 10000x faster.
• My RAM got 12800x bigger (and faster).
• My HDD got 8192x times bigger (and faster).

And yet my PC takes ages to boot
and I need more time to open up my text editor ...

Seriously, What Went Wrong ???
Rule #1: Know Your Hardware!
Embedded Linux Optimizations Techniques: How Not To Be Slow?
Common Considerations ...

• CPU SIMD Optimizations and Execution Modes:
  - Thumb-1/2: Tradeoff between code size and efficiency ...
  - Jazelle: Don’t do JAVA on ARM without it !
  - VFP / NEON: Impressive performance boost on all FPU operations;
                 Use integer-based routines otherwise.

  => Tradeoff between performances and portability (generic builds are meant for portability).

• Audio Management:
  - Choice #1: Legacy hardware DSP audio decoding (with complex shmem architecture) ?
  - Choice #2: Software Cortex-A9 audio decoding (within 50 MHz or so) ?

• Display / Input Optimizations:
  - GPU Capabilities: 2D blitting, 3D, post-processing ?
                 Ensure you’ll never fail into software fallback !
                 Don’t bother rendering more frames than your LCD can display.
  - TouchScreen: Calibrate your driver not to read more often than your max display FPS rate.
                 Reading on I2C consumes resources that you may never be able to interpret.
## Embedded Linux Optimizations Techniques: How Not To Be Slow?

### Embedded SoC Comparison...

<table>
<thead>
<tr>
<th>Our Test Case SoC</th>
<th>Apple iPhone 3GS</th>
<th>Apple iPhone 4</th>
<th>Samsung Galaxy S2</th>
<th>Today PC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction Date</strong></td>
<td>2009</td>
<td>2009</td>
<td>2010</td>
<td>2011</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>ARM1176</td>
<td>ARM Cortex-A8</td>
<td>ARM Cortex-A8</td>
<td>ARM Cortex-A9 MP</td>
</tr>
<tr>
<td><strong>Frequency (MHZ)</strong></td>
<td>500</td>
<td>600</td>
<td>1000</td>
<td>2 x 1200</td>
</tr>
<tr>
<td><strong>Memory Size (MB)</strong></td>
<td>256</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td><strong>L2 Cache Size (kB)</strong></td>
<td>None</td>
<td>256</td>
<td>640</td>
<td>1024</td>
</tr>
<tr>
<td><strong>FPU</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Specialized Instructions</strong></td>
<td>Thumb-1, Jazelle</td>
<td>Thumb-2, Jazelle, VFPv3, NEON</td>
<td>Thumb-2, Jazelle, VFPv3, NEON</td>
<td>Thumb-2, Jazelle, VFPv3, NEON</td>
</tr>
<tr>
<td><strong>Hardware GFX</strong></td>
<td>Limited 2D Blitter</td>
<td>Full 3D</td>
<td>Full 3D</td>
<td>Full 3D</td>
</tr>
<tr>
<td><strong>Hardware Video Engine</strong></td>
<td>Limited SD</td>
<td>Limited SD</td>
<td>Limited HD</td>
<td>Full HD</td>
</tr>
<tr>
<td><strong>Memory Bandwidth (GB/s)</strong></td>
<td>1.33</td>
<td>1.6</td>
<td>3.2</td>
<td>6.4</td>
</tr>
<tr>
<td><strong>Performances (DMIPS)</strong></td>
<td>625 (1.25 DMIPS/MHz)</td>
<td>1200 (2.00 DMIPS/MHz)</td>
<td>2000 (2.00 DMIPS/MHz)</td>
<td>6000 (2.5 DMIPS/MHz/Core)</td>
</tr>
<tr>
<td><strong>CPU PC Equivalency</strong></td>
<td>Pentium Pro @ 233 MHz (1996)</td>
<td>Pentium II @ 400 MHz (1998)</td>
<td>Pentium III @ 600 MHz (2000)</td>
<td>2x ATOM @ 1.3 GHz (2008)</td>
</tr>
</tbody>
</table>
Rule #2: Embedded is NOT Desktop!
Embedded Linux Optimizations Techniques: How Not To Be Slow?
Embedded is NOT Desktop ...

• **Brutal Facts:**
  - Embedded devices get more and more powerful each year.
  - But not everybody uses high-end ARM SoCs.
  - Still resources limited: CPU, memory bandwidth, run on batteries, slow I/Os ...

So why would you use the same kind of software than on a PC?
Android somehow came out and diverged from GNU/Linux for some reason ...

• **Good Hints on some desktop-oriented performances eating software/technologies:**
  - Abstraction Framework,
  - Messaging Bus,
  - Garbage Collector,
  - Virtual Machine,
  - Interpreted Language,
  - XML,
  - Data Parsing and Serialization.

Use these with care!
Badly used, they are sources of terrible difficulties.
Rule #3: Isolate Your System’s Bottlenecks!
Optimization requires accurate measurement.

Measure must:
- Be deterministic and repeatable.
- Not impact system’s behavior.
- Be the less intrusive as possible.

Try to cover as much usability scenarios as possible; don’t limit yourself to average Joe use cases.
Need for global feature/solution benchmark (*requires end-to-end implementation*)

- **At Input Level:**
  - **Record scenario:** At *tslib* level, we retrieve X/Y coordinates, pressure level and timestamp.
  - **Replay scenario:** We inject raw data to `/dev/input/eventX` and let the software handle events.
    - => Least intrusive input (mimics final human behavior).
  - Can also be fully automated through simple client/server approach.
Embedded Linux Optimizations Techniques: How Not To Be Slow?
Benchmarking: An External Approach (2/2) ...

• At Output Level:

- External video camera recording.
- Need to define scenario start and end conditions (e.g. some widgets appearance / disappearance).
- On a remote PC, play back the recorded video to measure delta between start/stop conditions using OpenCV libraries.
  - Measure is the least intrusive (no impact on target).
  - Can be used for non-regression tests on a given global feature.
  - But you still have no clue which exact part of your code is slow.
  - Accuracy depends on camera's capability (usually 30fps, so 33ms minimum threshold).
**Embedded Linux Optimizations Techniques: How Not To Be Slow?**

**Benchmarking: An Internal Approach ...**

- **Modern Linux kernel introduced support for hardware counters**
  - Introduced as **Performance Counters** (see [http://goo.gl/LldPv](http://goo.gl/LldPv)) in 2.6.31.
  - Renamed as **Performance Events** (see [http://goo.gl/KWIfo](http://goo.gl/KWIfo)) in 2.6.32+
  - Successor of **Oprofile**.
  - See tools/perf/ directory in kernel.

- **Example of usage (on OMAP 4430 Pandaboard):**
  - **Requirements:** You need debugging symbols to accurately trace your system.
  - **User-space Profiling:** `perf top -U`
  - **Kernel-space Profiling:** `perf top -K`

<table>
<thead>
<tr>
<th>PerfTop: 911 i9s/sec kernel:99.2% exact:</th>
<th>PerfTop: 2934 i9s/sec kernel:85.0% exact:</th>
<th>[kernel.kallsyms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>samples</td>
<td>pcnt function</td>
<td>DSO</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>147.00</td>
<td>32.9% _muldf3</td>
<td>/root/mac</td>
</tr>
<tr>
<td>69.00</td>
<td>15.4% main</td>
<td>/root/mac</td>
</tr>
<tr>
<td>54.00</td>
<td>12.1% __aeabi_fadd</td>
<td>/root/mac</td>
</tr>
<tr>
<td>50.00</td>
<td>11.2% __subdf3</td>
<td>/root/mac</td>
</tr>
<tr>
<td>47.00</td>
<td>10.5% __aeabi_fmul</td>
<td>/root/mac</td>
</tr>
<tr>
<td>32.00</td>
<td>7.2% __floatsdiv</td>
<td>/root/mac</td>
</tr>
<tr>
<td>25.00</td>
<td>5.6% __aeabi_d2f</td>
<td>/root/mac</td>
</tr>
<tr>
<td>23.00</td>
<td>5.1% __vmac_c</td>
<td>/root/mac</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**PerfTools** also can be used for global system profiling by generating a time chart:

- **On target:**
  - `perf timechart record` (will generate your `perf.data` samples).

- **On host:**
  - `perf timechart –i perf.data –o output.svg`

D-Bus events messaging can be generated using **dbus-monitor**, or better, **bustle**.

- Though very intrusive (impacts on performances).
- Can be extended to include `tcpdump` network messages into workflow.
- See [http://willthompson.co.uk/bustle/](http://willthompson.co.uk/bustle/) for more details.
Rule #4: Kill the Message Bus!

“Don’t Shoot The Messenger”, Shakespeare, 1598
Embedded Linux Optimizations Techniques: How Not To Be Slow?
RPC Frameworks Comparison ...

- **Study about different RPC architectures:**
  - Basic RPC function call between client and server.
  - Measure consists of 10000 calls on an AMD Athlon XP 2800+, 1 GB RAM.

- **Interesting results,**
  *CORBA* is known to be slow but:
  - DCOP is 3x slower.
  - DBUS is 18x slower.

- **Full analysis details are available at:**

<table>
<thead>
<tr>
<th></th>
<th>CORBA (ms)</th>
<th>DCOP (ms)</th>
<th>D-Bus (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOID Call</td>
<td>626</td>
<td>1769</td>
<td>9783</td>
</tr>
<tr>
<td>IN Integer Call</td>
<td>629</td>
<td>1859</td>
<td>10469</td>
</tr>
<tr>
<td>OUT Integer Call</td>
<td>660</td>
<td>1824</td>
<td>10399</td>
</tr>
<tr>
<td>IN/OUT Integer Call</td>
<td>686</td>
<td>1903</td>
<td>11162</td>
</tr>
<tr>
<td>IN String Call</td>
<td>650</td>
<td>1902</td>
<td>10510</td>
</tr>
<tr>
<td>OUT String Call</td>
<td>730</td>
<td>1870</td>
<td>10455</td>
</tr>
<tr>
<td>IN/OUT String Call</td>
<td>682</td>
<td>1952</td>
<td>11239</td>
</tr>
</tbody>
</table>
Embedded Linux Optimizations Techniques: How Not To Be Slow?

Messaging Benchmarks ...

- **Some IPC benchmark figures:**
  - Performed on TI Pandaboard (TI OMAP 4430 @ 2x 1GHz).
  - Reading rows from a SQLite database (75k rows chunks).
- **Different use cases:**
  - Native SQLite direct library function call.
  - Client/Server approach with UNIX sockets messaging channel.
  - Client/Server approach with D-Bus messaging channel.
  - Client/Server approach with D-Bus messaging channel with file descriptor support.

- See "IPC Performance" utility ([http://goo.gl/5ygSU](http://goo.gl/5ygSU)).
Embedded Linux Optimizations Techniques: How Not To Be Slow?
D-Bus Messaging: Be Careful ...

- D-Bus really is meant only for eventing/broadcasting; avoid passing data on it.
- There are more efficient and straightforward alternatives between 2 applications.
- Avoid passing large data: use D-Bus with UNIX file descriptor support instead.

- Remove paranoid message header/body checks/assertions:

```
diff -Naur dbus-1.5.0.orig/dbus/dbus-message.c dbus-1.5.0/dbus/dbus-message.c
--- dbus-1.5.0.orig/dbus/dbus-message.c 2011-08-06 12:31:50.624248071 +0200
+++ dbus-1.5.0/dbus/dbus-message.c 2011-08-06 12:32:49.264248103 +0200
@@ -3955,7 +3955,7 @@
    DBusValidationMode mode;
    dbus_uint32_t n_unix_fds = 0;

-    mode = DBUS_VALIDATION_MODE_DATA_IS_UNTRUSTED;
+    mode = DBUS_VALIDATION_MODE_WE_TRUST_THIS_DATA.Absolutely;

    oom = FALSE;
```
Embedded Linux Optimizations Techniques: How Not To Be Slow?

Rule #5: Go Native !!!
Embedded Linux Optimizations Techniques: How Not To Be Slow? Desktop Software Architecture Comparison ...

- Desktop Legacy Applicative Architecture Sample:

  - C/C++ code.
  - Graphical applications using native function calls to libraries.
  - Eventing through signals.
  - IPC through SysV IPC or UNIX /TCPIP Sockets.
  - Mastered memory usage.
  - Easily debuggable (using gdb or valgrind).
  - Easily profilable (using gcov, Oprofile, or Linux PerfTools).

Application’s portability, skin-ability and easiness of deployment really depends on how you write your code ☹️
Embedded Linux Optimizations Techniques: How Not To Be Slow?
Desktop Software Architecture Comparison ...

- **Desktop Web Applicative Architecture Sample:**
  - JS/HTML/CSS code.
  - Graphical user application using interpreted JavaScript functions with bindings to native middleware apps/libs.
  - WebServices usage and JSON data (de)serialization to exchange with middleware apps.
  - **JavaScript-based Apps:**
    - Easy and fast to write.
    - Even easier to skin, customize and deploy.
    - But interpreted and compiled in time, making them really hard to impossible to properly debug and/or profile.
    - Slower than any native equivalent.

  **Tradeoff needs to be made.**
Embedded Linux Optimizations Techniques: How Not To Be Slow?  
Browser Architecture Perspective: A Virtualized OS...

- **Browser Architecture:**
  - Makes JS portable to your legacy OS.
  - Specific bindings for OS and architectures.
  - Specifically designed modules to access the hardware beneath (audio, video, graphics, WebGL ...).

- **OS Concepts:**
  - Scheduler and Memory Allocator.
  - Applications Security / Sandboxing ...

- **Bindings for OS native services:**
  - HTML5 Local Storage
  - HTML5 Audio/Video tags ...

Modern browsers are to JavaScript what POSIX used to be for C.
Embedded Linux Optimizations Techniques: How Not To Be Slow?

Conclusion
Embedded Linux Optimizations Techniques: How Not To Be Slow?
The Embedded Linux Rules Set ...

Know Your hardware.

Embedded is NOT Desktop.

Isolate Your System’s Bottlenecks.

Kill the Message Bus.

Go Native!
Embedded Linux Optimizations Techniques: How Not To Be Slow?

Conclusion ...

• Your SoC never has been that powerful ...
  - ... ain't a reason for wasting it though.

• Don’t mimic software development trend!
  - Embedded Systems aren’t desktop PCs.
  - They can’t be programmed the same way.
  - Guess why Google’s Android differs from GNU/Linux?

• Back to the Basics!
  - It's not that more difficult to code in C/C++ than in JS or other "high-level language".
  - It's been proven to work; guess how's been coded your high-level language.
  - Go straight to the point: avoid as many indirection layers as you can.
Embedded Linux Optimizations Techniques: How Not To Be Slow?

Backup Slides:
Miscellaneous Tips & Tricks
• **Data (De)Serialization:**
  - Consumes a lot of CPU time: avoid at ALL cost whenever possible.
  - Only serialize elements you really need to, not the whole class content.
  - When possible, use shared memory instead.
  - Check serializer routines from the FOSS you include:
    - e.g. qjson adds extra white spaces that make it nice on Wireshark.
    - Our serialized 'contact' object (40 kB) contained 4 kB of white spaces.

• **Logs (seen in so many programs …):**
  - Check log macro level THEN compute log string, and not the opposite:

```c
#define LOG(lvl, format, arg...) do { \
    snprintf(fmt, sizeof(fmt), "%s: %s\n", format); \
    va_start(va, format); \
    if (lvl < DEBUG_LEVEL) { \
        snprintf(fmt, sizeof(fmt), "%s: %s\n", format); \
        va_start(va, format); \
        vfprintf(stderr, fmt, va); \
        va_end(va); \
    } \
} while (0);
```

```c
#define LOG(lvl, format, arg...) do { \
    if (lvl < DEBUG_LEVEL) { \
        snprintf(fmt, sizeof(fmt), "%s: %s\n", format); \
        va_start(va, format); \
        vfprintf(stderr, fmt, va); \
        va_end(va); \
    } \
} while (0);
```
Embedded Linux Optimizations Techniques: How Not To Be Slow? Miscellaneous Tips & Tricks (2/2) …

• **Memory Allocation:**
  - **Avoid Memory Fragmentation:** you’d better keep some objects in memory than continuously (de)allocating them.
  - **Real-Time Memory Management:** for performance critical apps, you’d better use a pre-allocated memory pool, that will never go in page fault (sloooooow).

• **Compiler Optimizations:**
  - GCC can do wonders by adding various optimizations flags (usually –march=…, -Ox, and –mfpu=neon when using floating point on ARM), but it’s a tradeoff with portability.
  - Isolate your critical sections code into dedicated C file and use Acovea (see [http://goo.gl/KdLqK](http://goo.gl/KdLqK)) for determining the best compiler options through evolutionary algorithms.
  - Rewrite your critical sections code using GCC inline ASM (very useful on codec routines).
  - See some FPU calculation on Pandaboard: Go [http://goo.gl/hT9Q7](http://goo.gl/hT9Q7) for benchmark sources.

<table>
<thead>
<tr>
<th>Measured Execution Time (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
</tr>
<tr>
<td>C with GCC Optimizations -O3 -fomit-frame-pointer -mcpu=cortex-a9 -ftree-vectorize -ffast-math</td>
</tr>
<tr>
<td>C with GCC Optimizations and NEON SIMD -mfloat-abi=softfp -mfpu=neon</td>
</tr>
<tr>
<td>Inline NEON ASM -mfloat-abi=softfp -mfpu=neon</td>
</tr>
</tbody>
</table>
**Embedded Linux Optimizations Techniques: How Not To Be Slow?**

**Web Technologies Optimizations** (if you _really_ wanna go this way) …

- **Web Rendering Engine Optimizations:**
  - Tune your rendering engine to use JIT.
  - Tune your rendering engine not to render invisible widgets (off-screen or hidden layers).
  - Tune your rendering engine to have a limited object cache (otherwise you'll quickly get low on free memory, which will induce more page faults and slow down your whole system until OOM gets its job done).

- **Simplify your CSS:**
  - Use regular images instead of slow CSS transformations.
  - Use solid pattern instead of gradients.
  - Use correct images size instead of software rescaling them each time.
  - **E.g:** Scroll lists with CSS gradient pattern took 90% CPU while using CSS solid pattern only took 3% in tests.
Embedded Linux Optimizations Techniques: How Not To Be Slow?
Web Technologies Optimizations (if you _really_ wanna go this way) …

• Parsing HTML is a CPU hog:
  Remove complexity by lowering DOM’s depth as much as possible.

• When designing WebServices, you’d better return a lot of information
  in one call than to proceed with multiple WS calls (anyway, you're asynchronous, right?)

• Don't refresh your MMI as much as possible, this is a terribly slow operation:
  You’d better wait for all of your data to be ready.

• If you’re lucky enough to have a recent engine, try delegating some graphics to GPU
  through OpenGL/WebGL to provide hardware acceleration.

• Additional JavaScript tips were provided at Oreilly’s conference
  “How to Make JavaScript Fast” (see http://goo.gl/K7VYd).

Process as much logic code as possible in C/C++ (i.e. go Native !!)

=> See Google’s Chrome NativeClient approach ( http://code.google.com/p/nativeclient/ ).
AT THE SPEED OF IDEAS™