Porting Linux to a New Architecture

Marta Rybczyńska

October 15, 2014
Different Types of Porting

- New board
- New processor from existing family
- New architecture
New Architecture: What it Means?

- Processor instruction set
  - Compile
  - Write the assembly parts
- Memory map: different peripherals
  - Configure drivers
  - Write new drivers
- Optimizations
  - New opportunities
  - Write optimized code
Porting Linux: Basic Elements

- Build tools
  - Gcc, binutils...
- The kernel
  - Core code
  - Drivers
- Libraries
  - Libc, libm, pthread, ...
- User space
  - Busybox, applications
One day..
you have a new architecture
First MPPA®-256 Chips with TSMC 28nm CMOS
256 Processing Engine cores + 32 Resource Management cores

- 256 (+32) user-programmable, generic cores
- Architecture and software scalability
- High processing performance
- High energy efficiency
- Execution predictability
- PCIe Gen3, Ethernet 10G, NoCX
The MPPA-256 Processor

- **Compute cluster includes:**
  - 16+1 cores
  - Shared memory
  - Network-on-Chip Interfaces
  - Debug unit (DSU)

- **IO cluster includes:**
  - 4 cores
  - Shared memory
  - Peripherals
The MPPA-256 Processor Core ISA

- Same on IO and compute cluster
- 5-issue Very Long Instruction Word (VLIW)
- DSP instructions
- Advanced bitwise instructions
- Hardware loops
- MMU
- Idle modes
- 32/64-bit IEEE 754 floating point unit
mkdir linux/arch/k1
The Initial Files: Less Than You Expect

- Processor startup
  - Configure the core
- Memory map
  - Initialize the memory allocators
  - Configure memory zones
- Processor mode change
  - Interrupts and traps
  - Clock interrupt
  - Context switch
- Device tree and KConfig
- Console (printk)
How To Write It?

- Read documentation
- Copy & paste
- Understand & write
Assembly vs C code

- K1 core is a VLIW: multiple instructions (one bundle) per cycle
  - High performance gain
    - GCC handles it well
    - Manual bundling OK for short code, hard for longer ones

- Result
  - Preferring built-ins over asm inlines
  - Less assembly in the code
Failed to execute /init
Kernel panic - not syncing. No init found
Time to Bring User Space Up

- Port libc (if not done already)
  - Which one? It depends...
  - For K1, we've ported uClibc
- First init can be statically linked
  - If not, dynamic loader needed first
Interface User<->Kernel (ABI)

- Program startup
  - Which values in which registers?
  - What is on the stack?
- Syscalls
- Signals
Instruction Set Simulator: Boot Process Debugging
init started: BusyBox v1....
First Executables

- First static libraries
- Then dynamic loader
- And some drivers
Early Testing

- Unit tests for the kernel space
  - Complicated build
- Debugging ease is important
  - Best if run in simulator
- “Test” init
  - Basic tests of all main functionalities in an “init”
## Traces: Visualization

This view displays matching pairs of [*_in/*_out*] and [*_ENTER/*_EXIT*] tracepoints, as nested sections.

<table>
<thead>
<tr>
<th>Thread</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8028C000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB2C000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB2A000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB34000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB36000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB38000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB3A000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB3E000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB46000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB48000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB4C000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB4E000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB50000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB54000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB58000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB5A000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB5C000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB60000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB62000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB64000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB6D000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB8D000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB8D000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB8D000</td>
<td>Inactive thread</td>
</tr>
<tr>
<td>0xFFB8D000</td>
<td>Inactive thread</td>
</tr>
</tbody>
</table>

Active thread [4,554-1,851,510,457] duration=1,851,505,903
Later Testing

- “Do it yourself”
  - Too much work
  - What is the expected behaviour?
- Use existing testsuites
  - For K1, we use LTP (Linux Testing Project)
  - Active, big number of tests at different level
open("/lib/libm.so.0", O_RDONLY) = 3
Enabling New Functions

- **Examples**
  - Traces
  - New file system
  - New device type

- **New functionality requires**
  - New kernel options
  - Support in kernel headers
  - Support in libc

- **Try Test-Driven-Development**
New Functionality Example: Strace and Ptrace (1)

- **Strace**
  - See syscalls run by a program
  - Shows both parameters and results
  - Useful for debugging errors

- **Implementation**
  - Ptrace calls
  - Signals
New Functionality Example: Strace and Ptrace (2)

- Unit tests
  - Available in LTP
- Strace implementation
  - The code compiles but...
  - Defines in the code
Supporting your hardware well
Special Cases

- SMP
- MMU
- Network-on-Chip
- Multiple address spaces
  - Device-tree
Building a distribution
Distribution Choices

- Do-it-yourself
- Buildroot
- Yocto
Summary: Lessons Learned (1)

- Divide the port in stages
- Test early
Summary: Lessons Learned (2)

- Use generic functionality if possible
- Keep the coding style
Summary: Lessons Learned (3)

- Use panic() and exit()
- Prefer code that doesn't compile if architecture unknown
Summary: Lessons Learned (4)

- Use and develop advanced debugging techniques
- Read documentation
- Read other platforms code
Questions?

Marta Rybczynska
marta.rybczynska@kalray.eu

http://www.kalray.eu