RTLinux in a FPGA

1. OpenRTU Project
2. FPGAs and soft processors
3. uClinux
4. RTLinux in Microblaze

www.os3sl.com
1. OpenRTU Project

- Spanish Industrial Research Project (Nov, 2004 – May, 2006)
- Financed by Spanish Industrial Ministry (PROFIT)
- Consortium:
  - TELVENT: real time company, RTUs
  - ESI: European Software Institute, Product Line
  - CSIC: Scientific Research Institution, FPGA Design
  - OS3: embedded Linux company, RTLinux
RTLinux in a FPGA

1. OpenRTU: Project Goal

Building a new generation of RTUs (Remote Terminal Units) using FPGAs and Open Source.

- More Flexibility
- Faster Development
- Better Scalability
- Avoiding Obsolescence Challenge
1. OpenRTU Project: Results

- A prototype built and running uClinux and RTLinux in Microblaze
- HARD Real Time requirements achieved (Initially, 1ms)
2. FPGAs and soft processors

- FPGA: Field Programmable Gate Array
- Device containing programmable logic and programmable interconnects.
- Field programmable = it can be programmed after the manufacturing process in the field
- You can program the HW !!!

www.os3si.com
2. FPGAs: How it can be programmed

- FPGA design using VHDL (VHSIC & HDL)
- An Electronic Automation Tool obtains a netlist from the VHDL code
- Place & Route software fits the netlist to the FPGA
- Validation through timing analysis, simulation and verification tools
- A bitstream is generated to program the logic gate array
RTLinux in a FPGA

2. FPGAs: How it can be programmed

- You can add IP (Intellectual Property) Cores to your design: libraries of predefined complex functions and circuits
- IP Cores: buses, codecs, DSPs, interfaces, ... and processors
- Soft processors (FPGA logic): picoblaze (Xilinx), Microblaze (Xilinx), Nios (Altera), LatticeMico32

www.os3sl.com
2. FPGAs: Microblaze soft processor

- Xilinx Microblaze (4.0) 32 bits processor
- Three-stage pipeline
- RISC, Harvard architecture
- Configurable Code and Data Caches
- Hardware Debug Logic
- Non-MMU processor
RTLinux in a FPGA

3. uClinux

- Linux for non MMU processors

- Available ports: DragonBall, Coldfire, QUICC, ARM7DMI, Intel i960, Blackfin, Microblaze, NEC V850

- Commercial products based on uClinux: IP cameras, wireless routers, VoIP based telephones
3. uClinux: drawbacks

- No protection between tasks, and even worse: a user process can crash the system

- This is the most well-known issue, but it is not the only one

- Using Linux code is not automatic
3. uClinux: non MMU problems

• Processes are created using the vfork system call instead of fork
• User stack per process is static in size
• Memory management done by the OS is different
• Dynamic libraries are not available (at least as standard)
3. uClinux: Microblaze Architecture

- Port done in 2003 by John Williams, from Queensland University in Australia
- uCLinux 2.4 in Microblaze is being used in several commercial products
- Xilinx has recently released the uClinux 2.6 for Microblaze
4. RTLinux

- RTLinux is a hard real time microkernel
- Interrupts are virtualized for Linux
- Linux runs as the task with the lowest priority inside RTLinux: Linux is the idle task for RTLinux
RTLinux in a FPGA

Virtualization technology is in fashion

www.os3si.com
4. RTLinux in Microblaze

- RTLinux DOES NOT need an MMU (but can use it)

- RTLinux needs Linux or uClinux: uClinux port in Microblaze done by John Williams in 2003

- What HARD real time performance can be achieved with a 75Mhz soft processor running a GPOS?
4. RTLinux in Microblaze: Doing the PORT

- We had some initial doubts about the technology.
- These doubts were reinforced when we found a bug in the processor IP core implementation.
- This led us to put most of the blame on the technology when latencies were not as expected.
4. RTLinux in Microblaze: Doing the PORT

- We decided to divide the work clearly:
  1) Interrupts virtualization layer
  2) RTLinux microkernel

- In case of problems with the full RTLinux microkernel coexisting with uClinux kernel, we could just make use of the virtualization mechanism for a simple system executing critical code when an event raises an interrupt without uClinux interference.
4. RTLinux in Microblaze: Doing the PORT

- Once the Interrupt Virtualization layer was implemented, first tests showed latencies higher than expected.

- The measurements were done with the timer interrupt, and Linux could be interfering with the results.

- We decided to wait until we knew what latencies we had with the full RTLinux microkernel working.

- ... But we suspected OPB (On-Chip peripheral Bus) was introducing the delays.
4. RTLinux in Microblaze: Doing the PORT

- The second part of the work was done to implement the full RTLinux microkernel: threads creation and destruction, scheduling, timer programming (one shot and periodic) and threads synchronization and communication.

- The RTLinux microkernel code is composed of architecture specific part, and by independent architecture which will run without changes.
RTLinux in a FPGA

4. RTLinux in Microblaze: Doing the PORT

- Once the full RTLinux microkernel was implemented, tests showed peak latencies higher than expected
- We did some code inspection but in C language level
- Due to our initial doubts about the uncertainty of the technology, we suspected OPB was introducing the delays
- This took us to the longest route to solve the problems, but on the positive side, this was not a complete waste of time.
4. RTLinux in Microblaze: Doing the PORT

- The solution taken was to make use of a special Microblaze configuration avoiding the execution of real time code from DRAM or SRAM

- LMB and Microblaze caches have 1 cycle access

- The idea was to allocate the real time code into these special memories avoiding the peak latencies when the code had to go through the OPB
4. RTLinux in Microblaze: Doing the PORT

Drawbacks when using 1 cycle access memories

- LMB Ram Blocks are used by FPGA designers, so we can not take them for free.

- Microblaze cache is write-through, so if the problem is when RAM or SRAM is accessed, we are in the same point
4. RTLinux in Microblaze: Doing the PORT

- The first problem was maximum space available (Spartan3) using LMB was 16Kbytes (just 12Kbytes aligned)

- RTLinux modules:
  - rtl.o: 8192 bytes (only code)
  - rtl_time.o: 2264 bytes (only code)
  - rtl_sched.o: 13692 bytes (only code)
RTLinux in a FPGA

4. RTLinux in Microblaze: Doing the PORT

- RTLinux code distribution was modified, creating a new module where "real" real time code is allocated.
- In the old modules we left the code just used during initialization.
- The new module `rtl_previous_core` had a final size of 8K, so it could be allocated in the LMB.
4. RTLinux in Microblaze: Doing the PORT

- We did other changes in uClinux related with interrupts code: a new section was created in the elf kernel file for this code, and during the initialization it is copied to LMB

- Once we had all the code related with real time in 1 cycle memory access, we did new tests and ...

- Peaks latencies had survived the attack
4. RTLinux in Microblaze: Doing the PORT

Technology was absolved. We had the real guilty:

The RTLinux Port Implementation was buggy
RTLinux in a FPGA

4. RTLinux in Microblaze: Doing the PORT

Some problems hard to find:

- Microblaze has not lock instructions: changes in `set_bit`, `test_bit`, `clear_bit`, `test_and_set`, `test_and_clear`, which need to disable interrupts.

- Some `cli` uClinux operations were not being virtualized

- compilation flags: muls and divs by software introduce high latencies

- buggy gcc 2.95 with 64 bits operations (needed for timers)

www.os3sl.com
## RTLinux in a FPGA

### 4. RTLinux in Microblaze: Results

<table>
<thead>
<tr>
<th>Periodic Task</th>
<th>Microblaze Caches Enabled</th>
<th>Code at LMB?</th>
<th>System Load</th>
<th>task jitter</th>
<th>irq timer worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>500us</td>
<td>Yes</td>
<td>No</td>
<td>IDLE</td>
<td>90</td>
<td>21</td>
</tr>
<tr>
<td>500us</td>
<td>Yes</td>
<td>No</td>
<td>Stressed</td>
<td>135</td>
<td>36</td>
</tr>
<tr>
<td>500us</td>
<td>Yes</td>
<td>Yes</td>
<td>IDLE</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>500us</td>
<td>Yes</td>
<td>Yes</td>
<td>Stressed</td>
<td>38</td>
<td>23</td>
</tr>
<tr>
<td>500us</td>
<td>No</td>
<td>No</td>
<td>IDLE</td>
<td>54</td>
<td>35</td>
</tr>
<tr>
<td>500us</td>
<td>No</td>
<td>No</td>
<td>Stressed</td>
<td>104</td>
<td>40</td>
</tr>
<tr>
<td>500us</td>
<td>No</td>
<td>Yes</td>
<td>IDLE</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>500us</td>
<td>No</td>
<td>Yes</td>
<td>Stressed</td>
<td>37</td>
<td>24</td>
</tr>
<tr>
<td>500us</td>
<td>Yes SRAM</td>
<td>No SRAM</td>
<td>IDLE</td>
<td>74</td>
<td>24</td>
</tr>
<tr>
<td>500us</td>
<td>Yes SRAM</td>
<td>No SRAM</td>
<td>Stressed</td>
<td>141</td>
<td>39</td>
</tr>
</tbody>
</table>

www.os3sl.com
4. RTLinux in Microblaze: Doing the PORT

Conclusions

- we followed the longest route to achieve the hard real time performance

- The usual way would had been to suspect first that the implementation is buggy, but we had some preconceived ideas...

- On the positive side, we have now the best performance we can get using RTLinux and uClinux in Microblaze
RTLinux in a FPGA

Thank you

Alejandro Lucero
alucero@os3sl.com

www.os3sl.com