

Improving the Operating System with Reconfigurable Hardware (FGBS'11)

Michael Gernoth





System Software Group

Friedrich-Alexander University Erlangen-Nuremberg

November 11, 2011



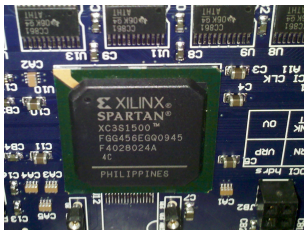
Challenges in Operating Systems

- Is there a way to tackle challenging properties of operating systems:
 - **Security:** Running software must not alter security policies 
 - **Safety:** Entering a safe state, even when the system crashes 
 - **Determinism:** Enforcing hard limits to reach real-time goals 
 - **Performance:** Increasing system throughput 



Reconfigurable Hardware

- Today, cheap reconfigurable hardware is readily available
 - Big low-cost FPGAs are common
- Reconfigurable hardware makes building adaptable hardware possible
 - Logic cells contain boolean lookup tables which can be reprogrammed
 - Flip-flops and RAM cells can be used for data storage
 - This makes it feasible to adapt hardware to the requirements of software



State of the FPGA

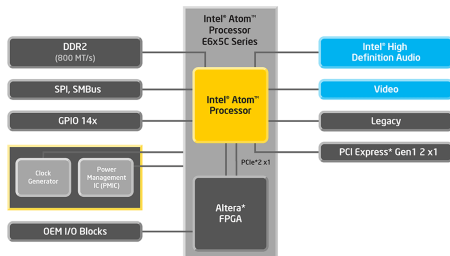
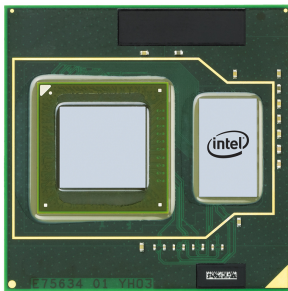
- Today FPGAs are mainly used in userspace applications
 - Applications: Video filtering, Signal pre-/postprocessing, SDR, ...
 - OS: cryptographic operations, manage FPGA resources for the user¹
- Reconfigurable hardware is designed independently from software
 - Static interfaces between hard- and software
 - Interface changes need to be made in both “worlds”
- As software is “easier” to develop, software usually follows hardware
 - Software becomes more complex to keep hardware simple

¹ Lübbers, Platzner: *ReconOS: An RTOS Supporting Hard- and Software Threads*, FPL 2007



Integrating FPGAs into commodity systems

- FPGAs are easily integrated into modern systems
 - *PCI*: Full PCI bridges can be synthesized into (cheap) low-end FPGAs
 - *PCIe*: Current FPGAs have integrated PCIe endpoints
- Processor manufacturers even add FPGAs to new processors

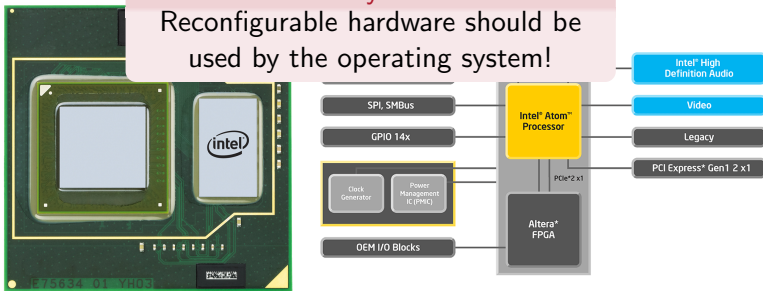


Integrating FPGAs into commodity systems

- FPGAs are easily integrated into modern systems
 - *PCI*: Full PCI bridges can be synthesized into (cheap) low-end FPGAs
 - *PCIe*: Current FPGAs have integrated PCIe endpoints
- Processor manufacturers even add FPGAs to new processors

FPGAs are broadly available

Reconfigurable hardware should be used by the operating system!



Outline

Motivation

Problem

- Interfacing an FPGA with the OS
- Accessing and Understanding OS Data
- Coping with Operating System Changes

Framework

- Automatic Interface Generation
- DMA Abstraction
- Hardware OS Component Implementation

Evaluation

- Malicious-Process Killer
- Simple Hardware Scheduler

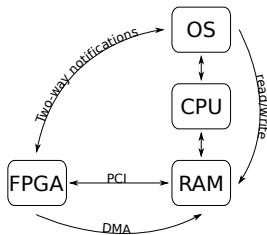
WIP and Conclusion

- Generating Hardware from OS code



Interfacing an FPGA with the OS

- To achieve good performance, the hardware needs to access data without asking the OS for it
 - PCI and PCIe both provide DMA capabilities
 - Usually used with OS-supplied DMA memory regions
 - But accessing the whole physical memory is possible
 - The operating system is not involved at all
- ⇒ Ideal interface for high-performance reconfigurable hardware

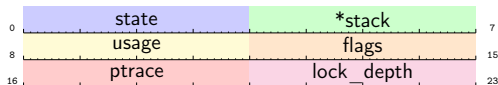


Accessing and Understanding OS Data

- Data accessible by DMA is only meaningful to the kernel

```
c14d75a0: c9ba 1fab 9d1c 622e 9735 ea5e 4c57 b730
c14d75b0: dc6c 3afe d38f 61a4 81fd 6eb6 b18e e8c6
c14d75c0: 1043 ce08 8232 2ab9 2529 1510 5f49 539f
c14d75d0: b5e5 ddb7 b378 bd88 3981 db6f 06bd d361
```

- But the OS should not be bothered to convert interesting bits of data
- Hardware design needs to know operating system data formats
 - Operating system structures have to be described in hardware
 - Member offsets in structures can be hard-wired into the implementation



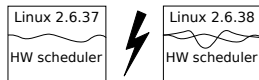
⇒ Hardware is able to access and understand OS data



Coping with Operating System Changes

- Operating system internals change with every version
- Best example: Linux

- No stable kernel API



(see `linux-2.6/Documentation/stable_api_nonsense.txt`)

- Memory layout even dependent on kernel configuration
- Transforming system state to a hardware-defined format is expensive
- Manually adapting hardware to changing software is fragile

Solution

Generate hardware interfaces
automatically



Outline

Motivation

Problem

Interfacing an FPGA with the OS

Accessing and Understanding OS Data

Coping with Operating System Changes

Framework

Automatic Interface Generation

DMA Abstraction

Hardware OS Component Implementation

Evaluation

Malicious-Process Killer

Simple Hardware Scheduler

WIP and Conclusion

Generating Hardware from OS code



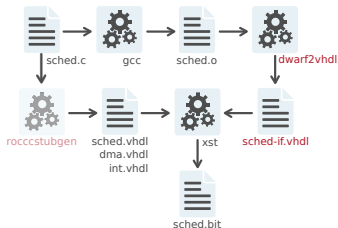
A Framework for Hardware OS Components

■ Mechanisms

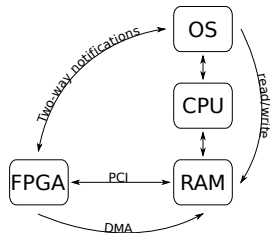
- Automatic interface generation
- Partial hardware component generation from kernel source code
- Integration into kernel build process

■ Features

- Abstraction of DMA transfers
- Two-way notifications between soft- and hardware



Mechanisms



Features

Automatic Interface Generation

- Use DWARF debug symbols in kernel objects to generate interfaces
 - Memory layout as created by the compiler
 - Used for VHDL interface generation
 - Stable interface for hardware implementation
 - Integrates in kernel build process



object file



```
entity struct_simple is port (  
  signal flags_i: in std_logic;  
  signal info_i: in std_logic;  
  signal data_i: in std_logic;  
  
  signal base_addr_i: in std_logic_vector(31 downto 0);  
  signal addr_o: out std_logic_vector(31 downto 0);  
  signal clk_i: in std_logic);  
end entity struct_simple;  
  
architecture struct_simple_arch of struct_simple is  
begin  
  process (clk_i)  
  begin  
    if (clk_i'event and clk_i = '1') then  
      if (flags_i = '1') then  
        addr_o <= base_addr_i + 0 + 0;  
      elsif (info_i = '1') then  
        addr_o <= base_addr_i + 0 + 2;  
      elsif (data_i = '1') then  
        addr_o <= base_addr_i + 0 + 4;  
      else  
        --  
      end if  
    end if  
  end process  
end architecture struct_simple_arch;
```


VHDL file



Automatic Interface Generation (2)

1. Reads an object file generated with DWARF debugging symbols
2. Determines data type, location, size and name for each Debugging Information Entry (DIE)
3. Generates VHDL entities for each structure
4. Generates VHDL accessor code for each member of a structure
5. Connects each VHDL entity to the DMA multiplex of the framework

```
struct struct_simple {  
    unsigned short flags;  
    char info [2];  
    char *data;  
};
```



```
architecture struct_simple__arch of struct_simple is  
begin  
    process (clk_i)  
    begin  
        if (clk_i 'event and clk_i = '1') then  
            if (flags_i = '1') then  
                addr_o <= base_addr_i + 0 + 0;  
            elsif (info_i = '1') then  
                addr_o <= base_addr_i + 0 + 2;  
            elsif (data_i = '1') then  
                addr_o <= base_addr_i + 0 + 4;  
            else  
                addr_o <= base_addr_i;  
            end if;  
        end if;  
    end process;  
end architecture struct_simple__arch;
```



- DMA transfers are handled by the framework
 - Hardware component accesses data with simple reads and write
 - All timing critical operations are handled by the framework
- Multiple components can be active at the same time
- Framework will also provide handling of coordination between operating system and hardware
 - No atomic read-modify-write possible
 - But shared data-structures have to be consistent
- Hardware components can be (de-)activated on-the-fly

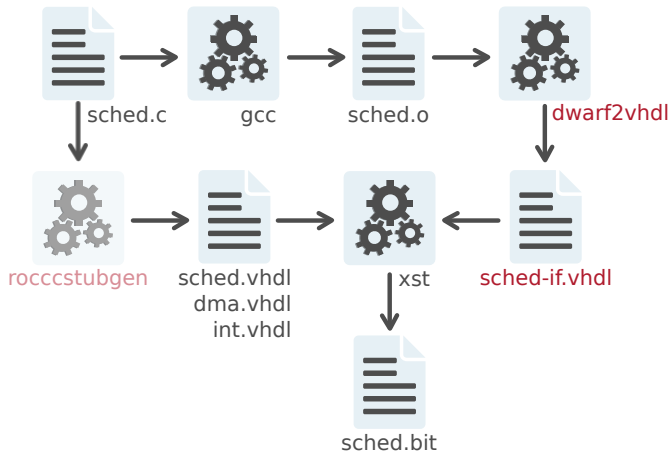


Hardware OS Component Implementation

- Components are currently manually implemented in VHDL
- Only core functionality needs to be implemented
- Two-way notifications are provided by the framework
 - Notification of the OS by interrupts is provided
 - Notification of the hardware by the OS is provided
- Visible and invisible components possible
 - **Visible:** The operating system knows and interacts with the component
 - **Invisible:** The component works independently from the operating system



Building Hardware OS Components



Outline

Motivation

Problem

Interfacing an FPGA with the OS

Accessing and Understanding OS Data

Coping with Operating System Changes

Framework

Automatic Interface Generation

DMA Abstraction

Hardware OS Component Implementation

Evaluation

Malicious-Process Killer

Simple Hardware Scheduler

WIP and Conclusion

Generating Hardware from OS code



- Two hardware OS components were built using the framework
 - Malicious-process killer (invisible component)
 - Simple scheduler (visible component)
- Hardware is synthesized for a XILINX Spartan 3 (XC3S1500) FPGA
- FPGA is connected to the PCI bus of the host system
- Host system is an Intel Pentium 4 with 2.4GHz
- Operating system on the host is Linux 2.6.32



Malicious-Process Killer

- Component terminates unwanted processes
 - Constantly iterates over the task structure
 - Executable name is compared to a blacklist
 - Process identification could also be more complex (e.g. memory checksums)
 - If the name is on the list, a KILL signal is added to the pending signals of the process

- The operating system does not know the hardware component exists
 - It can not be disabled or modified during run-time
 - Rebooting the system will not disable the hardware
 - A remote attacker has no way to disable the component



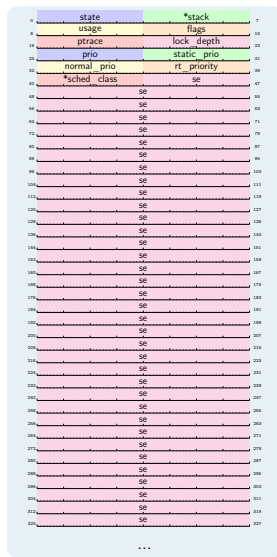
Simple Hardware Scheduler

- Simple uniprocessor round-robin scheduler
- Replaces `pick_next_task()` in `linux/kernel/sched.c`
- Works on the global process list `task_struct`
- Determines next process by iterating through list
- Writes the address of the next `task_struct` to be scheduled to a memory location
- Sends an interrupt after the allotted time slice for the process is reached
- Can be activated and de-activated at run-time



Access to the Process List

- `task_struct` is converted by `dwarf2vhdl`
 - Complex structure with multiple embedded sub-structures
 - 127 members in Linux 2.6.32
- Kernel linked list is traversed in hardware
- Conversion between virtual and physical addresses is done in hardware
- Address of first element (`init_task`) is determined during kernel compile-time, can be changed at run-time



Preliminary Results

- Scheduling jitter is only affected by variance in soft interrupt latency

Best	Worst	Average
6.3 μ s	13.8 μ s	7.8 μ s

(1000 measurements of soft interrupt latency on the host)

- These latencies also occur with the software-based scheduler (timer)
- Only small changes to existing kernel code necessary (≈ 10 lines)
- Decision process is independent from operating system
- Determining the next process only involves reading one memory location
- Task switching is initiated as soon as the OS receives the interrupt



Outline

Motivation

Problem

Interfacing an FPGA with the OS

Accessing and Understanding OS Data

Coping with Operating System Changes

Framework

Automatic Interface Generation

DMA Abstraction

Hardware OS Component Implementation

Evaluation

Malicious-Process Killer

Simple Hardware Scheduler

WIP and Conclusion

Generating Hardware from OS code



WIP: Generating Hardware from OS code

- Is it possible to translate an operating system component into hardware?
 - **SystemC** is suitable for writing hardware in C++, not for translating existing software into hardware, FPGA compilers are expensive and closed
 - The open-source **Riverside Optimizing Compiler for Configurable Computing (ROCCC)**² can only translate simple C sources to VHDL
- Work based on **ROCCC**, as it is available and extendable
 - Automatically simplify operating system code for ROCCC
 - Simplified code is used to generate a base VHDL component
 - Final component is built by manually extending the base component

² Guo, Buyukkurt, Najjar, Vissers: *Optimized Generation of Data-path from C Codes for FPGAs*, DATE 2005

- Framework for hardware operating system components

- Interface generation
- Memory access abstraction
- Hardware OS components for Linux
 - Simple scheduler
 - Malicious-process killer

⇒ Reconfigurable hardware should be used by the operating system

- Future work

- Base hardware component generation using ROCCC
- Implementation of more hardware OS components
 - **Security:** Hardware-based policy enforcement
 - **Determinism:** Real-time scheduling
 - **Performance:** Network routing

