Improving Non-Functional Properties of Operating Systems by Reconfigurable Hardware

Michael Gernoth*, Tiago Mück†, Alexander Würstlein*, Wolfgang Schröder-Preikschat*
*Friedrich–Alexander University Erlangen–Nuremberg, Germany
{gernoth, arw, wosch}@cs.fau.de
†Software/Hardware Integration Lab
Federal University of Santa Catarina, Florianopolis, Brazil
{tiago}@lisha.ufsc.br

I. Abstract

The increasing availability of cheap FPGAs gives rise to the question if and how operating systems can benefit from these means. So far, only applications in userspace (like video filters) or special dedicated algorithms in kernel space (like cryptography) make use of reconfigurable hardware.

Building hardware which operates on operating system internals presents the problem of getting access to the required data needed for its operation. The host system has to transform internal data structures into a format understood by the hardware and write the result to the configuration interface of the hardware. This induces overhead which can negate the benefits of using the hardware in the first place.

In this paper we propose a technique to build hardware which understands operating system internals and can adapt to changes in the system without adding overhead and complexity to the system software.

This allows us to implement services of an operating system as hardware blocks to gain additional, or improve on existing non-functional properties like performance, safety, security and determinism.

II. Introduction

With the availavility of reconfigurable hardware which can be easily integrated into common computer systems, the question arises if this hardware can be reasonably used by the operating system. If an operating system service can be easily migrateable to hardware, a deep integration of hardware into core operating system functionality becomes possible, providing additional non-functional properties[5] to the system.

This is, in contrast to current approaches using reconfigurable hardware, for improving system performance. These are usually used for specific userspace applications like video filters or algorithms from special domains (like cryptography) in kernel space. Even "reconfigurable operating systems" like ReconOS[6] focus on using reconfigurable hardware to improve userspace applications by providing access to hardware threads to users. These traditional approaches use specialized drivers which are tailored to the hardware and "know" how to interface the hardware with the system.

When trying to migrate a operating system service to hardware, this programming model no longer suffices and a more flexible interface to the hardware is needed. Looking at the Linux kernel development makes it is obvious that there are no static interfaces in the core kernel code. This results in every approach of using a fixed hardware interface as an interface to a kernel service to be immediately obsolete.

As a consequence of this, the only way of using reconfigurable hardware in the kernel is by adapting the hardware to the software and not the other way around.

This paper presents a novel way to aid the translation of a software feature to hardware and the automatic generation of interfaces for the resulting hardware which are dynamically adaptable to software changes.

A. Motivation

Integrating hardware into operating system kernels was not really feasible in the past, as building hardware meant designing and producing silicon chips with defined interfaces to the system software. The operating system needed to transform internal data structures into formats understood by the hardware, tell the hardware to work on a given problem and transform the result computed by the hardware back into internal data-structures. This overhead of transforming data structures needed to be done in the operating system, as fixed hardware can only be adaptive to small interface changes.

There is also the problem of different code bases for the software and hardware side of the interface, which could lead to subtle bugs in the implementation, not found when looking at either source base alone.

With the introduction of reconfigurable hardware, this problem of transforming data structures can be solved by adapting the hardware implementation to the operating system and not the other way around. This also allows the hardware to operate on its own without being instructed by the system. When hardware has an understanding of system internals, an efficient external manipulation of system state becomes possible.

This work was partly supported by the German Research Foundation (DFG) under grant no. SCHR 603/7-1 and SFB/TR 89.

III. Improving Non-Functional Properties of Operating Systems

An additional benefit of having reconfigurable hardware executing system services is the possibility to improve the non-functional properties of the system. Traditionally, desktop systems put their focus on drastically different properties than embedded systems which are used in safety critical applications. With the addition of hardware, some properties from one domain can be made available in another. A few examples of these non-functional properties and their respective focus are:

1) Performance: Perceived overall speed of the system. Desktop systems focus on this property e.g. to improve the user experience to provide a smooth desktop experience.
2) Safety: The system does not harm its environment. E.g. when an unexpected event occurs, the system enters a predefined safe state. Embedded systems in safety critical areas (like engine-control ECUs) focus on this property.
3) Security: Security policies like access-control lists or capabilities are enforced by the system providing data integrity, confidentiality and availability of the system in case of atttacks. This property is pronounced in server- and multiuser-systems.
4) Determinism: The system follows a predefined execution plan, so the cause for each action of the system can be determined, be it the scheduling of a task or the reaction to events from the surrounding environment. Real-Time Systems are an example of systems focusing on this property.

Current operating systems focus on optimally achieving only a subset of these properties, because the addition of more properties leads to conflicts with existing properties by inducing additional overhead or complexity.

If a current operating system allows the user to change its non-functional properties, this is done by techniques like preprocessor macros[8] or AOP[4].

IV. Integrating Hardware in the Operating System Core

The overhead of additional properties in an operating system (as described in section III) can be drastically reduced by having (reconfigurable) hardware implement an operating system service providing the property. This reconfigurable hardware should have direct access to the hosts memory and the ability to send interrupts to the CPU when interaction with the operating system becomes necessary. Due to these requirements, an FPGA connected to the PCI or PCI-Express bus seem like an ideal choice to provide the integration of reconfigurable hardware into an operating system.

A. Solving the interface-problem

Moving operating system code to hardware blocks has the disadvantage that the operating system still needs to convert data into a representation understood by the hardware implementation. This includes converting internal data structures into a format understood by the hardware by resolving all indirections and writing the result into dedicated configuration registers. When the overhead and added complexity induced by these operations is taken into account, this type of integration rapidly loses its appeal.

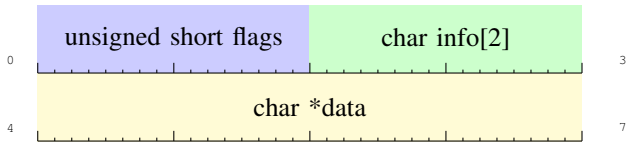

Fig. 1. Memory layout of **struct sample**

To help with this problem of interfacing hardware with software, we adapt the hardware to the operating system and not the operating system to the hardware. In order to achieve this goal, we extract the memory layout of internal operating system structures by looking at object-files including debugging informations produced during compilation of the operating system kernel. The *DWARF* debugging format used by *gcc* and other compilers contains enough information to produce HDL code for accessing the structures referenced in the object and can provide means to adapt the hardware to inner changes of the system.

B. The dwarf2vhdl Tool

We have implemented the idea from section IV-A in a tool we call *dwarf2vhdl*, which reads in an object file with *DWARF* debug symbols and generates *VHDL*-code to access the individual members of referenced data structures. This code can then be used to access the contents of these structures by utilizing *DMA*-transfers from the main memory to the FPGA card. Computed results can be transferred back to the appropriate memory location in the same way. This all happens without the host system needing to prepare the data and initiating the execution of the hardware service.

```
1 struct struct_sample {
    unsigned short flags;
3   char info[2];
    char *data;
5 };
```

Listing 1. Simple C struct

Listing 1 gives an example of a very simple structure with just three members, each being 16 or 32 bit long (on a 32 bit architecture). The memory layout of this simple structure is shown in Figure 1. When this structure is converted to VHDL with *dwarf2vhdl*, code as shown in Listing 2 is generated.

```
1 entity struct_sample is port (
    signal flags_i: in std_logic;
3   signal info_i: in std_logic;
    signal data_i: in std_logic;
5
    signal base_addr_i: in std_logic_vector(31
        downto 0);
7   signal addr_o: out std_logic_vector(31
        downto 0);
```

```
    signal clk_i: in std_logic
9   );
  end entity struct_sample;
11
  architecture struct_sample__arch of
      struct_sample is
13 begin
    process (clk_i)
15  begin
      if (clk_i'event and clk_i = '1') then
17      if (flags_i = '1') then
          addr_o <= base_addr_i + 0 + 0;
19      elsif (info_i = '1') then
          addr_o <= base_addr_i + 0 + 2;
21      elsif (data_i = '1') then
          addr_o <= base_addr_i + 0 + 4;
23      else
          addr_o <= base_addr_i;
25      end if;
      end if;
27  end process;
  end architecture struct_sample__arch;
```

Listing 2. Struct from Listing 1 converted to VHDL

This generated interface consists of a declaration of input- and output signals (lines *2 – 8*) and a hardware representation of the memory layout of the structure (lines *12 – 28*). The generated signals in this example are:

- `flags_i`, `info_i` and `data_i` (lines *2 – 4*): Inputs selecting which member of a structure should be accessed. By setting one of these inputs to *high*, the address of the corresponding member is calculated.
- `base_addr_i` (line *6*): 32 bit input for the base-address of the structure. This can either be a static address determined during compilation of the kernel or a dynamic address obtained by following pointers from other structures.
- `addr_o` (line *7*): 32 bit output providing the address of the reqeuested struct-member. This address can then be used to access the member in-memory by using DMA-transfers.
- `clk_i` (line *8*): Input providing a clock-signal. This eliminates undefined states of the output-signals by clocking on the rising edge.

```
  elsif (se_group_node_i = '1') then
2   addr_o <= base_addr_i + 44 + 20;
  elsif (se_group_node_next_i = '1') then
4   addr_o <= base_addr_i + 64 + 0;
  elsif (se_group_node_prev_i = '1') then
6   addr_o <= base_addr_i + 64 + 4;
  elsif (se_on_rq_i = '1') then
8   addr_o <= base_addr_i + 44 + 28;
  elsif (se_exec_start_i = '1') then
10  addr_o <= base_addr_i + 44 + 32;
```

Listing 3. conversion of nested structures in the Linux task_struct to VHDL

Applying our tool to a more complex structure containing nested structures and a nested linked list (`struct sched_entity` contains the list `struct list_head group_node`) is (partially) shown in Listing 3, which is a part of the Linux-kernel process-structure `struct task_struct`. This is a structure with 127 members (in Linux 2.6.32.15) which is correctly converted by *dwarf2vhdl* to VHDL code.

After generating the interfaces with *dwarf2vhdl*, they can be connected to other hardware blocks implementing the corresponding service and providing access to the main memory. Even if the layout of a structure changes, in most cases only the automatic regeneration of the interface becomes necessary and the hardware implementation can be left untouched. This is very useful when accessing structures which are in a constant flow of changes, where most changes are not affecting the functionality required by the hardware but change the memory-layout (like `struct task_struct` in Linux, which changes with most released kernel versions). With this ability it is possible to integrate hardware services into highly dynamic systems with ever-changing interfaces like the Linux kernel.

V. Prototypical Implementation

In our research we are using a PCI connected FPGA board[1] to show the practical applicability of our research to the Linux and EPOS[2] kernel. The FPGA is directly connected to the PCI bus of the host system, with no additional bridge-chips in between.

The choice of using the PCI bus in contrast to the PCI-Express bus is caused by the availability of a high quality open source PCI core[3] with bus-master DMA functionality, which can be configured to suit our requirements. Equivalent cores for PCI-Express are currently not available under those terms, so the decision for PCI was made, even though the decreased latency and increased bandwith of the PCI-Express bus would be beneficial. Nevertheless, our work should be easily adaptable to PCI-Express once an open-source core implementing the wishbone interface[7] becomes available.

This is currently all work in progress, so that's it for now. More to come...

A. Scheduler (EPOS/Linux)

B. Security (Linux)

C. Routing (Linux)

[3]

D. Current status

VI. Related work

[2] [1]
[6]

[1] Raggedstone-1 from Enterpoint, containing an Xilinx XC3S1500
[2] http://epos.lisha.ufsc.br/
[3] http://opencores.org/project,pci

VII. Future work

This paper describes the automatic generation of hardware adapting to operating systems. We are also working on automating the task of transforming operating system code to hardware blocks. For this we are extending the *ROCCC*[1] compiler to cope with operating system source code and generate hardware blocks from Linux kernel source code.

VIII. Conclusion

References

[1] Zhi Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized generation of data-path from c codes for fpgas. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Zhi Guo, A. Mitra, and W. Najjar. Automation of ip core interface generation for reconfigurable computing. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1 –6, August 2006.

[3] Huan Liu. Routing table compaction in ternary cam. *IEEE Micro*, 22:58–64, 2002.

[4] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX ATC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[5] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *4th AOSD W'shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, March 2005. Northeastern University, Boston (NU-CCIS-05-03).

[6] E. Lubbers and M. Planner. Reconos: An rtos supporting hard-and software threads. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 441 –446, 2007.

[7] OpenCores. *Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores*. OpenCores, 2010. http://cdn.opencores.org/downloads/wbspec_b4.pdf.

[8] Reinhard Tartler, Julio Sincero, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Configurability bugs in Linux: The 10000 feature challenge. 9th USENIX Symp. on OS Design and Implementation (OSDI '10), October 2010. Poster.