A Hardware/Software Co–Execution Model Using Hardware Libraries for a SoPC Running Linux

Alejandro Gómez-Conde, José de Jesús Mata-Villanueva, Marco A. Ramírez-Salinas, and Luis A. Villa-Vargas

Center for Research in Computing,
National Polytechnic Institute, D.F., Mexico.
alegzc@yahoo.com.mx, gsusmata@gmail.com, mars@cic.ipn.mx, lvilla@cic.ipn.mx

Abstract. A SoPC (System on a Programmable Chip) is the combination of computer architecture, IP designs, and an embedded operating system implemented on a FPGA. A SoPC developer needs to possess skills in advanced hardware design and efficient software programming to make high performance solutions on this platforms type. These requirements have resulted in a complicated and error-prone development process, therefore the dissemination of this promising technology has been confined to a small group of developers. To address this problem, our proposal exemplifies a Hardware/Software co-execution model where software tasks are assisted by hardware libraries in a SoPC.

Keywords: System on a Programmable Chip, Linux FPGA system, Mixed execution mode, Function interception.

1 Introduction

A SoPC system requires a FPGA with a processor core that can be either *hard* or *soft*. This designation refers to the flexibility or configurability of the core. *Hard cores* have a custom VLSI layout that is added to the FPGA in the manufacturing process. They are less configurable and tend to have higher performance characteristics than *soft cores*. Although *soft cores* are highly configurable they use logic elements as well as routing resources within a FPGA that increases the complexity of hardware design and reconfigurable stages. The overall implementation of a SoPC system needs to meet one or more of these constraints:

- Performance
- Execution time
- Small footprint area
- Communication and connectivity
- Implementation cost
- Upgradability
- Low power

Advances in Computer Science and Engineering
Research in Computing Science 58, 2012 pp. 287–297
Paper Recived 01-10-2012 and Acepted 12-11-2012

The outline of this paper is as follows: Section 2 describes our proposal. Section 3 presents the experiments and results for the implementation process of a SoPC that uses a Hardware/Software co-execution model that includes the creation of a hardware platform, the integration of the Operating System and the implementation of the bitswapper hardware function. Section 4 discusses related work. Finally a conclusion is given.

2 Hardware/Software Co-Execution Model

In a traditional execution model, a user program often includes references to software libraries. Once the user–program starts execution it requests software functions; those functions are grouped together in software libraries that may be statically or dynamically attached to the user program.

A Purely software-based execution model does not always take advantage of the hardware within the platform nor is it able to delegate software tasks to hardware modules. As a direct consequence the execution model does not always meet the performance or execution time constraint for a specific task in a SoPC system. Our approach was to develop a Hardware/Software co-execution model for critical applications in a SoPC. Our proposal uses a library-based access model and extends its functionality to support the execution of hardware functions. Figure 1 shows the proposed execution model. The idea behind interception is to assist a software task by hardware modular functions in a seamlessly manner. If the hardware function is not available when the request is issued then the process that requests such functionality should either wait for the hardware function to be programed into the FPGA or follow its execution path until the requested function is provided. Our proposal waits for the hardware function to be configured and then continues its execution assisted by a hardware function. This capability adds upgradability to a SoPC system because there is no longer the need to modify the application source code to embedded hard-coded routines that interact with hardware functions. If a new SoPC system is needed, our project will provide a Hardware/Software partitioning model to get the best performance at the lowest implementation cost using Hardware/Software partitioning, co-design and co-synthesis techniques but even if there is not access to the source code, an application can be assisted to reduce its execution time.

Our proposal uses the Dynamic Linking Loader service but instead of *calling* the original software library function, it *calls* a hardware modular function that might be programed at run-time in the reconfigurable area of the SoPC. The communication protocol between the user program and the hardware function is implemented by a dynamic library and a character device driver. The main reason for using a character device driver is that it can be dynamically loaded at the system startup and all the functionality is in the kernel module; the dynamic library could be set up to capture function calls for specific programs while the rest of the systems work as usual. This behavior could be activated or deactivated at any time while the system is still running as long as the hardware function is not in use. Once the hardware function has been configured and the

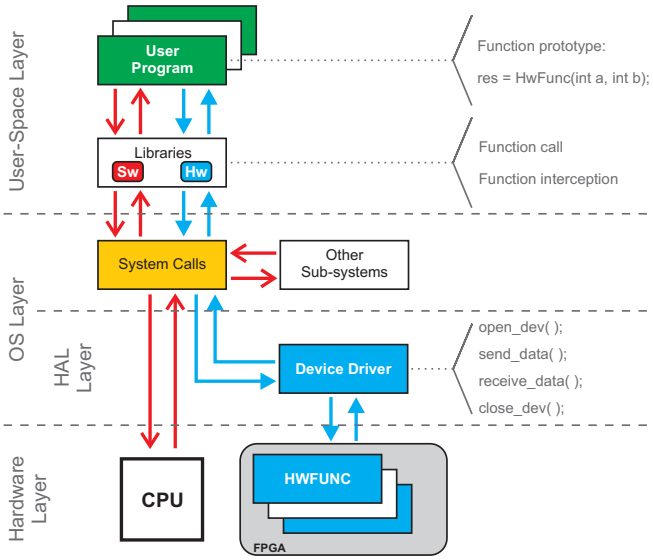

Fig. 1. Proposed execution model.

software access layer has been properly set up, the program can take advantage of the functionality provided by the hardware module. A performance penalty is expected to exist as a direct consequence of the reconfiguration process at run-time when a hardware function is called for the first time. When a large amount of data is processed by the function, the impact of the reconfiguration process becomes smaller.

3 Experiments and results

3.1 Hardware Platform

A Linux-capable SoPC hardware platform was built up using Xilinx's EDK tools. Figure 2 shows a brief description of the architecture `S1` implemented in the FPGA of the ML507 development board. It integrates the PowerPC 440 Processor, a DDR2 memory controller, interrupt controller, serial port, JTAG debug port, float point unit, System ACE, a timer, and some other peripherals. Although Xilinx's tools helped in the integration and configuration processes, it was necessary to incorporate a description of the elements within the hardware platform besides of the `xparameters.h` description file.

D. Gibson and B. Herrenschmidt in [1] propose a *device tree model* to generate a hierarchical description of a hardware platform. The most important properties of the *flattened tree* obtained from the *device tree model* are: relocatable properties that allow the bootloader or the kernel to move around the blob

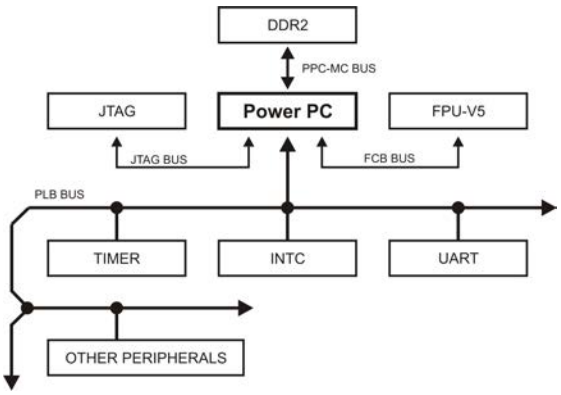

Fig. 2. Architecture of the hardware platform S1.

tree as a whole, without need to parse or adjust its internals. It is also possible to *insert* or *delete* a node or a whole *subtree*; this property will help us integrate hardware functions without having to recompile the kernel.

Xilinx in collaboration with the Open Source Community developed a BSP package that was integrated into the SDK to generate a *device tree* description for the `S1` hardware platform.[2]

The hardware platform `S1` was tested using `Push_Buttons_5Bit_GPIO` and `LEDs_8Bit_GPIO` hardware modules in stand alone mode using *Xilinx's SDK*.

3.2 Integration of the Operating System

An embedded operating system was generated for the ML507 Board using Poky framework tools[3] and `meta-xilinx`[4] specification layer. This framework provides a structured scheme to integrate architecture specifications in the kernel compilation processes and provides a cross–compile platform to create, test and debug user–space as well as kernel–space applications. Figure 3 shows the workflow used in Poky to generate the Operating System.

`U-boot` was chosen as the bootloader because Xilinx's modified version of U-Boot has been successfully adapted and tested to work with Virtex5 FPGAs.[5] The `meta-xilinx` configuration and specification layer was modified to successfully generate the kernel image and the bootloader.

The most important files generated using Poky framework are: The file *u-boot-ml507.ace* contains the FPGA configuration *bitstream* and the executable file for U-Boot in a System ACE file format. It will be loaded at startup to configure the FPGA, and start the system's boot process. The file *uImage-virtex5.bin* is a binary file of the kernel image. It is adapted to be *launched* by U-Boot. The file *uImage-virtex5.dtb* holds a *flattened tree blob* that represents the internal hardware architecture implemented within the FPGA. The file *poky-image-minimal-virtex5.tar.gz* is a compressed image of the SoPC file system (including

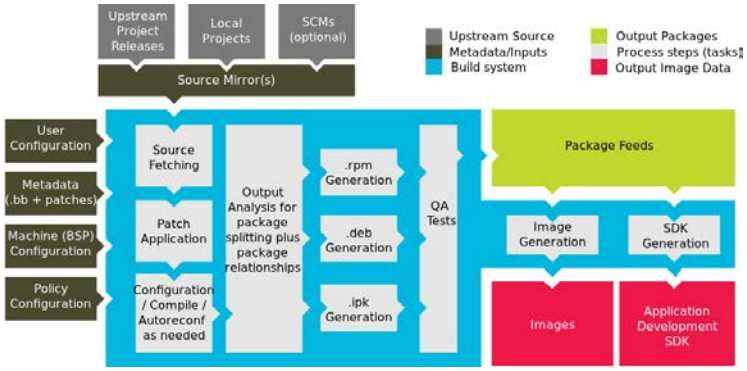

Fig. 3. Poky Architecture Workflow.

`/boot`, `/dev`, `/etc`, `/home`, `/lib` and others). The file *modules-2.6.37+-r12-virtex5.tgz* is another compressed file that contains kernel modules that are dynamically loaded at *run time*.

In order to run the Operating System on the ML507 board the user must set the configuration switch SW3 so that the FPGA is configured by the SysACE as described in [6]. The Compact Flash must be partitioned in tree sections; the first one must be formated as *FAT16*, the second partition should be *ext3* and the third partition should be used as the swap partition. The first partition must have the `u-boot-ml507.ace`, `uImage-virtex5.bin` and `uImage-virtex5.dtb` files in it. The second partition must be filled with the content of `poky-image--minimal-virtex5.tar.gz` and the `modules-2.6.37+-r12-virtex5.tgz`.

The serial communication program `kermit` was used to explore the boot process and the kernel startup. The last part of the output text shown in the console for the boot process is presented below. The last four lines of text are associated to the *device tree blob* and the *kernel* load respectively. The device tree specifies architecture details that the kernel will use to properly setup the system. The kernel image was generated using the *uimage* format that contains the compressed *vmlinux* plus a few extra bytes of *metadata* that describe the kernel load address and the image name.

```
CPU:   Xilinx PowerPC 440 UNKNOWN (PVR=7ff21912) at 400 MHz
DRAM:  256 MB
FLASH: 32 MB
In:    serial
Out:   serial
Err:   serial

reading uimage.dtb
25059 bytes read

reading uimage.bin
```

```
2096237 bytes read
```

At the end we obtained a fully functional Linux-based Operating System that runs on the ML507 board; the kernel was generated using the 2.6.37 Linux kernel source code.

3.3 The Hardware Functional Unit

The hardware functional unit *bitswapper* changes the endianness of a data word from big endian to little endian and vice versa. This function uses Xilinx's `IPIF` module as an interface between the functional unit and the system. This functional unit has been chosen for its simplicity and ease of implementation; it is not yet a real functional unit, but it lets us debug integration procedures and provided valuable information to select the best communication model to be implemented between a hardware function and the software layer. Our first approach was to attach it to the PLB bus. It uses two registers and a basic communication protocol to send and receive data to and from the system through the IPIF interface.

Xilinx's EDK tools were used to attach the *bitswapper* hardware module to the existing platform; the core functionality of the proposed function is presented below.

```
process(datain)
begin
   for i in datain'low to datain'high loop
      my_swap_sig(i) <= datain((C_SLV_DWIDTH-1)-i);
   end loop;
end process;
dataout <= my_swap_sig;
```

Figure 4 shows the synthesized modular hardware function and fig. 5 shows the simulation results for the core functionality of the proposed hardware modular function.

Our implementation used a character device driver to provide communication services between the hardware function and the user program. Therefore a special file entry in the `/dev` folder was created; it was named `hwfn1`. The kernel module implements `device_init`, `device_exit`, `send_data`, `receive_data`, `device_open`, `device_read`, `device_write`, `device_release`, and other functions that are commonly related to character device drivers. Our proposal generates a dynamic library as described in [7]; to prove the function interception process the software library function `bitswap` was created; in our embedded development environment this software function is provided by the dynamic library `libhw.so`. The Dynamic Linking Loader uses a special feature that is controlled by the system variable `LD_PRELOAD`; in order to intercept the `bitswap` call another dynamic library was created using the same function prototype but implementing the communication protocol through the file descriptor associated to `hwfn1`. The kernel uses the services provided by the character device driver to send data to and receive data from the hardware function.

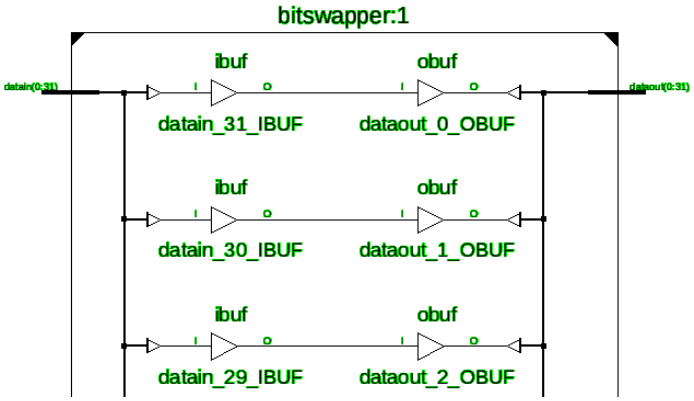

Fig. 4. RTL Schematic representation for Hardware Functional Unit.

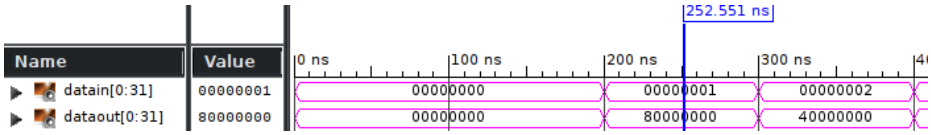

Fig. 5. Simulation results for the core functionality of the *bitswapper* function.

Table 1 presents a summary of the FPGA resources used in the *bitswapper* hardware modular function and for the SoPC system implementation.

Table 1. FPGA resource utilization.

Resource type	Bitswapper	SoPC System	Available
Slices	88	5668	11200
Slice Registers	109	8939	44800
Slice LUTs	86	9926	44800
IOBs	0	210	640
BlockRAMs	0	40	148
Memory used (kib)	0	1404	5328
ICAPs	0	1	2
DCM	0	1	6

4 Related work

Hayden Kwok-Hay So *et al.*[8] presented a very interesting Hardware/Software co-design methodology based on BORPH (*Berkeley Operating system for Re-*

Programmable Hardware). It included an Operating System designed for reconfigurable computers that implements hardware modules designed by the user; it also provides a native kernel support to implement processes in FPGAs using a homogeneous Unix interface for hardware and software processes. An interesting issue of this project was that a hardware process implemented into an FPGA inherits the same level of service from the kernel, as a software process does. BORPH includes file system support, interrupt support, and scheduling capabilities for hardware processes; it also provides common Unix APIs that interact with hardware processes in a seamlessly manner. The unified file interface allows hardware and software processes to communicate via standard Unix file pipes. BORPH was implemented on a development board with five FPGAs; one was used as a master unit while the others each held an implementation of a hardware process. They also developed a Hardware/Software interface that allows application development, migration of software procedures to hardware modules and a comparison among Hardware/Software implementations of the same application. Although they provided a revolutionary implementation of hardware processes, they only implement user hardware design into FPGAs and did not consider partial reconfiguration.

Qingxu Deng, *et al.*[9] propose a Hardware/Software unified architectural model for an FPGA implementing partially runtime reconfigurable services. Their proposal implements scheduling services and on-line placement of hardware modules within the same FPGA. Hardware and software tasks were implemented as processes but each hardware task uses a software communication layer to interact with the Operating System. They also implement a modified version of the system call *exec* to recognize HELF files (*Hardware Executable and Linkable Format*) and extract the information required to program a hardware IP-Core and its dedicated communication software layer from the HELF file; this new type of executable file format is an extension to ELF. A key feature of this proposal is that it implements a routing and placement algorithm to reduce the required time to schedule and execute hardware tasks in the reconfigurable area of the FPGA. Even though this proposal presented a new approach to create an abstract model of a hardware process that can be launched from user-space it is only fully functional for that specific platform since other platforms may not recognize the HELF executable file format; besides that, the functional elements of this proposal, at kernel level, cannot be easily migrated into other architectures due to the fact that they are deeply merged into a specific version of the Linux kernel.

Vaibhawa Mishra *et al.*[10] propose a dynamically reconfigurable SoPC that uses an Operating System based on the 2.6.34 Linux kernel. They implement a minimal hardware platform in the ML507 development board using the FPGA, the PowerPC 440 *hard core* processor and other peripherals. This proposal uses floating–point hardware modular functions to prove the reconfigurability of the system. They propose a functional model that is similar to a peripheral device

model with the only difference of implementing one *virtual device* that can be re-programed at runtime to implement any of the four floating–point functional units developed. Although this proposal implements reconfigurability using the *hwicap* IP-Core on a Virtex5 FPGA, it did not provide a major advantage over other proposals made in the past.

5 Conclusion and future work

SoPC systems are about to be a major improvement in the mobile devices field. Therefore our proposal is building up the required infrastructure to be able to generate, test, validate, and improve our knowledge of this type of systems.

Task execution will no longer use a purely software execution model; nevertheless, there cannot be a solid conclusion about which methodology shall prevail.

This work aims to provide an overview of the challenges that are faced while designing and implementing SoPC systems.

The *products* generated in this work are: a reconfigurable hardware platform for SoPCs and Poky-Linux framework tools and a configuration layer to build and integrate an Operating System to the ML507 board. The on going work implements another communication scheme between the hardware function and the software layer to compare and select a lower latency communication scheme for the SoPC system. Our team is developing a set of hardware modules to complement the actual hardware library.

Hardware functional unit *bitswapper* was developed and used to test the SoPC system for both correctness and performance. Although it was designed for test purposes, this modular hardware unit lets us explore the overall development process over the little-known path of mixed execution model.

Although the operating system is still in the development phase and, even when the hardware platform has not been tuned up to the maximum allowable performance, early results are promising. The Hardware/Software execution model reduces the CPU processing load when using hardware functional units.

This work is part of the MASA Project, the current development stage focuses on implementation specifically on upgradability.[11] Another on-going work of this project has developed a framework for custom routing and partitioning, that will enable effectively implementing relocatable hardware functions of variable size (in terms of reconfigurable partitions) and number of input/output ports.

Future work will design, implement and compare our model with a system-call based access model and a memory mapped access model that attaches a hardware function to the user–process virtual memory.

References

1. D. Gibson and B. Herrenschmidt. Device trees everywhere. OzLabs, IBM Linux Technology Center. February 13, 2006.
2. Xilinx Inc. Device Tree Generation. *Online resources.* `http://elinux.org/Device_Trees` and `http://wiki.xilinx.com/device-tree-generator` Visited on March 2011.
3. Poky Platform Builder. *Online resource.* `www.pokylinux.org/` Visited on January 2011.
4. Xilinx Inc. BSP Specification Layer in Poky to support ML50x development platforms. *Online resource.* `http://git.yoctoproject.org/cgit/cgit.cgi/meta-xilinx/` Visited on March 2011.
5. Xilinx Inc. Xilinx's modified versión of `U-Boot`. *Online resource.* `http://git.xilinx.com/?p=u-boot-xlnx.git;a=summary` Visited on April 2011.
6. Xilinx Inc. ML507 Evaluation Platform. *Online resource.* `http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf` Visited on Nov 2010.
7. U. Drepper. How To Write Shared Libraries. *Online resource.* `http://people.redhat.com/drepper/dsohowto.pdf` , 2011.
8. H. Kwok-Hay, A. Tkachenko and R. Brodersen. A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH. Department of Electrical Engineering and Computer Science University of California, Berkeley. CODES+ISSS'06, October, 2006.
9. Q. Deng, Y. Zhang, N. Guan and Z. Gu. A Unified HW/SW Operating System for Partially Runtime Reconfigurable FPGA based Computer Systems. 2008.
10. V. Mishra, K. Solomon Raju and P. Tanwar. Implantation of Dynamically Reconfigurable Systems on Chip with OS Support. International Journal of Computer Applications. 2012.
11. L. Villa , M. Ramírez, O. Espinosa, and C. Peredo. Modular Architecture for Synthesized Applications. Center for Research in Computing, IPN. Mexico, DF. `http://www.microse.cic.ipn.mx/masa-es` , 2010.