BORPH: Operating system support on the NetFPGA platform

Brandon Kyle Hamilton*
University of Cape Town

Hayden Kwok-Hay So
University of Hong Kong

ABSTRACT

This paper introduces the concepts behind BORPH, an operating system for reconfigurable computers. The porting and implementation of this operating system for the NetFPGA platform, as well as the the tool flow integration are described.

1. INTRODUCTION

The design of applications targeted for reconfigurable platforms, which include both standard CPU processors as well as reconfigurable FPGA fabric, is a complex and highly specific process, with a large amount of effort going into efficient hardware/software co-design. Due to the non-standard architecture of these platforms, there exists a lack of design reusability as interfaces between the hardware and software are typically highly specific to the target platform. Additionally, these applications tend to be limited by the master/slave accelerator model, whereby a software program has to be written to control the running and data communication requirements of a hardware design.

The adoption of an operating system that provides an abstraction layer to the reconfigurable platforms is needed to allow designers to shift their focus more towards their applications without needing to deal with these implementation issues, as well as promote portability and usability within the operating environment.

The BORPH operating system has been ported to the NetFPGA platform to provide a unified operating environment for the development and execution of hardware designs. This enables developers and users to easily access the NetFPGA through standard Unix binary utilities and file system access.

2. RELATED WORK

HASTE (Hybrid Architectures with a Single, Transformable Executable) [2] provides a single unified file containing hardware and software components of a reconfigurable application. The architecture to support this consists of a CPU, Reconfigurable Unit (RCU), and a Hardware Conversion Unit (HCU). The hardware conversion unit is responsible for creating the reconfigurable logic at run-time. An Instruction Set Architecture (ISA) is provided that allows the HCU to convert sequential code into parallel spatial code during application execution. The partitioning between hardware and software is determined at run-time.

*Supported by SKA, South Africa

A real-time embedded operating system, hThreads[1], was introduced by Andrews et. al. that extends the principle of a software thread to hardware. Their system allows threads to be run simultaneously on a CPU and the FPGA fabric. A thread scheduler runs on the FPGA fabric as well as the CPU simultaneously. An API is provided that exposes system calls for communication with registers on the FPGA fabric.

ReconOS[4] is another real-time operating system developed for reconfigurable platforms, based on the eCos Real-Time Operating System. This system provides a multi-threaded programming model for hardware and software design. An abstraction layer allows for the creation of portable and flexible multi-threaded applications.

3. BORPH

BORPH is an operating system designed for FPGA-based reconfigurable computers[5], implemented as an extension of the Linux kernel. Reconfigurable hardware, such as FPGAs, are treated as computational resources within the operating system. These resources are defined as hardware regions (HWRs), which can be implemented as an entire FPGA, or partially reconfigurable regions within a single FPGA. This abstraction allows the kernel to deal with the platform specific details and removes the need for the application designer to address these low level implementation details.

Through a UNIX process model, BORPH provides a unified hardware/software application runtime environment. Runtime support services, such as file system access, network access, and signal handling are made available to hardware designs by the kernel.

BORPH aims to provide increased usability for reconfigurable platforms, with a focus on simplifying hardware/-software co-design. An entire design, consisting in part of either hardware, software or both hardware and software components, is encapsulated in a BORPH executable file. A unified interface to execution and communication with the application is provided by the kernel. This allows designers to focus efforts on the application specific functionality of their designs, without the need to worry about low level implementation details such as writing device drivers for communication.

3.1 Hardware Processes

Analogous to a standard UNIX process (an executing instance of a software program), BORPH introduces the concept of a *hardware process*, which is an *executing instance of a gateware program.* As software processes are executed

on the system CPU, hardware processes are executed on the reconfigurable hardware regions (HWRs) within the system.

A key advantage following from this is that the running gateware is an active entity within the system, with no control transfer needed between hardware and software. This is in contrast to the typical accelerator (master/slave) model of a software program providing control to a passive gateware design via an API.

An additional advantage of this approach is that the allocation of reconfigurable hardware regions and resources (including the ability to run multiple designs at the same time if more than one HWR is present) is dealt with by kernel instead of the application designer.

A running hardware process behaves almost identically to any other software process in a Linux system, including standard functionality such as process management and hierarchies, and the handling of signals and interrupts. As an example: the status of a hardware processes can be checked by using a command such as *ps*, and can be controlled by sending signals such as SIGTERM using the *kill* command, or SIGINT by pressing *Ctrl-C*.

3.2 BORPH Object File

A *BORPH Object File (BOF)* is a binary executable that encapsulates a gateway design, as well as additional runtime information needed by the kernel. At compile-time, the system is partitioned into hardware (FPGA bitstream) and software (ELF file), and these elements are encapsulated in the BOF file. A BOF file is run in the same way as any other Linux executable file, such as ELF, and can be spawned with the standard *fork* and *exec* system calls. Upon execution, the kernel interprets the hardware configuration that is encapsulated within the BOF file, and automatically configures an appropriate hardware region (HWR).

3.3 IOREG system

Memory mapped registers are exposed via a virtual file system interface by the kernel during execution of a hardware process. This IOREG interface extends the proc file system (*procfs*), providing a systematic, language independent interface, allowing for easy communication with hardware processes. This removes the need for the hardware designer to redesign a driver interface with each design.

A new `hw` directory is created and populated for each running hardware process with information specific to the hardware design. This `/proc/<pid>/hw/` directory, corresponding to a hardware process running with process id `<pid>`, will contain three files:

`hardware_region` : A virtual file that contains information about physical location of the hardware process. On platforms that only contain a single hardware region, such as the NetFPGA, all processes will be run at the same physical location.

`ioreg_mode` : The contents of this file determines the operating mode of the `ioreg` directory. ASCII mode operation (the default) is indicated as 0, whereas a 1 indicates binary mode operation. Writing the desired value to this file will cause the kernel to transfer data in the corresponding mode when reading or writing to registers in the `ioreg` directory.

`ioreg` : A directory containing virtual files corresponding to each user defined register described by the symbol table in the BOF file. Reading or writing to files in this directory will cause the kernel to transfer data to and from the hardware region.

As an illustration of the usage of this interface, retrieving the contents of a 4-byte on-chip register called `MYREG` can be accomplished by a simple shell *cp* command:

```
1:bash$ cp /proc/1337/hw/ioreg/MYREG ~/
```

or similarly, in a C program:

```
memfile = fopen("/proc/1337/hw/ioreg/MYREG", "r");
fread(buf, 4, 1, memfile);
```

3.4 Hardware/Software interaction

A BOF file encapsulates both the hardware and software components of a reconfigurable computing application. A BOF file can be created (using the *mkbof* utility) with either a specified software component (ELF file) included, or with only the hardware component (FPGA bitstream). Including the software component of an application in the BOF file enables the use of the FPGA as an accelerator, where the running application consists of both software and hardware components as a single process. If this process is killed, both the software and hardware components will be stopped, and the BORPH kernel will remove the corresponding *IOREG* entries. Alternatively, the hardware is run as an independent entity, and communication with the hardware process can be achieved via the *IOREG* filesystem interface.

4. PORTING AND IMPLEMENTATION

The process model provides a consistent interface across different FPGA-accelerated systems. Based on the standard Linux kernel, BORPH is portable to any platform with a processor capable of running a Linux kernel. The low level platform specific functionality required to port BORPH to a new platform includes a few simple functions defining communication with the reconfigurable hardware on the target platform. These are:

- `configure` - This function handles the reconfiguration of a hardware region. It receives the configuration data file extracted from a BOF file and is responsible for transferring that to configure the FPGA.
- `unconfigure` - This function is called when a hardware process is terminated. It is responsible for unconfiguring the FPGA.
- `reserve_hwr` - This function is called to identify an available hardware region that will execute the current hardware process.
- `release_hwr` - This function is called to release a hardware region, making it available for future use.
- `get_buffer` - This function returns a pointer to a buffer that will be used for data transfer.
- `put_buffer` - This function will Deallocate the data buffer obtained in get_buffer.
- `send_buffer` - This function initiates a transfer of data to the FPGA.

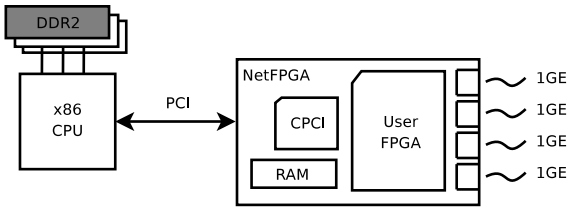

Figure 1: NetFPGA schematic

- `recv_buffer` - This function initiates a transfer of data from the FPGA.

We have ported the BORPH kernel to the NetFPGA[3] platform, based on the latest Linux 2.6 kernel. The kernel runs on the host system CPU and all communications between the kernel and the single user FPGA on the NetFPGA card are achieved over the PCI bus (Figure 1). The NetFPGA is viewed as a single HWR in the kernel, and executing a BOF file simply involves the automatic configuration of the NetFPGA hardware region by the kernel.

On system start-up, the PCI interface is initialized by the BORHP kernel, which maps the CPCI device into system memory and creates the network devices. The CPCI is then re-programmed with gateware that exposes the networking devices and FPGA via the PCI bus.

The platform-specific communication functions within the kernel are then able to read and write to the NetFPGA hardware region over the PCI bus via the memory-mapped locations using DMA. Configuration is performed (in the `configure` function) by setting the relevant CPCI registers and writing the configuration gateware to it, which it then uses to program the FPGA.

Communication through the IOREG interface is handled by accessing the memory-mapped FPGA locations in the `send_buffer` and `recv_buffer` functions.

5. INTEGRATION WITH THE NETFPGA ENVIRONMENT

The NetFPGA kernel driver code has been integrated into the corresponding BORPH kernel functions mentioned above. Access to the NetFPGA via DMA works as in the kernel device driver, and performance will be comparable. The NetFPGA projects have been converted to use the IOREG system through file I/O, removing the need for memory mapped register access.

6. TOOL FLOW INTEGRATION

6.1 NetFPGA register file system

The NetFPGA register file system uses an XML file to specify the registers provided by modules that are included in a project. The register generation tool (*nf2_register_gen.pl*), provided with the NetFPGA software distribution, reads the project description and relevant modules, and performs the required register and memory allocation. The output consists of a set of files with interfaces available for Verilog, C and Perl.

For use with BORPH, this tool has been modified to output an additional file, consisting of a symbol table describing the register memory locations and sizes. This file is then used, together with the FPGA bitstream, as input to a utility (*mkbof*) that will generate a corresponding BOF binary executable file.

6.2 Developing software for BORPH

BORPH enables developers and users to easily create software that communicates with the available reconfigurable regions. The hardware/software interface is abstracted, and exposed by the kernel via the file system. This allows software to be created in any language that supports file I/O, as well as make use of standard binary utilities (such as the Linux *cat* command).

As an example of this advantage, we take the GUI of the SCONE router included with the NetFPGA package, which is written in Java. Currently this implementation requires a native interface wrapper to a compiled C library in order to communicate with the NetFPGA via the memory-mapped registers. Within the BORPH environment, the Java application only has to read or write from the corresponding file to achieve this communication, eliminating the need for low level memory-mapped access via the compiled library module.

Software applications need to be aware of the *Process ID (PID)* of the running *hardware process* in order to access the correct directory in the */proc* filesystem for communication with the FPGA. If the software component is included as part of the BOF file, the software will simply use its own *PID*. Otherwise, standalone software can be passed the *PID*, or the full file path, via a command line argument.

7. EXAMPLE: REFERENCE NIC AND SCONE ROUTER IN BORPH

The Reference NIC project included with the NetFPGA software distribution is used to demonstrate the process of creating and running NetFPGA gateware with the BORPH operating system. The corresponding commands and output are illustrated in Figure 2, and Figure 3. This project makes use of a standalone hardware process model. The SCONE router project has also been modified for BORPH, with the software and hardware components unified in a single process.

7.1 Generating the BOF file

The first step is the generation of the register information from the project XML file using the modified *nf2_register_gen.pl* script. This script would be automatically executed during synthesis/simulation of the project, or can be directly executed to regenerate the register memory map, as in this case (Figure 2). The resulting symbol file (`reference_nic.symtab`) and gateware bitstream (`reference_nic.bit`) are passed to *mkbof*, to produce the binary executable (`reference_nic.bof`).

For the SCONE project, the *Makefile* has been modified to produce the BOF file, using the compiled SCONE software and the reference router bitstream. The SCONE software has been modified to use the IOREG interface via file IO instead of memory-mapped register access.

7.2 Executing the BOF file

Running the design in BORPH can be done via executing the binary file on the command line, which will create the hardware process. In this example of the Reference NIC, the hardware is run as a standalone background process, so we

can further read and write to the exposed device registers via the IOREG interface. The last command in Figure 3 shows a read from the register representing the number of packets received over the first network interface in the Reference NIC design, returning a value of 2.

For the SCONE example, the SCONE software is included in the BOF file. Running this file results in the configuration of the FPGA, and execution of the SCONE software component.

8. CONCLUSION

BORPH extends the familiar UNIX semantics to reconfigurable computers, providing a unified and well known interface to using and running hardware designs. This operating system has been ported to the NetFPGA platform, and the NetFPGA development tool flow has been extended to take advantage of this model, allowing users and application designers can to benefit from its advantages.

9. REFERENCES

[1] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving Programming Model Abstractions for Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):34–44, January 2008.

[2] B.a. Levine and H.H. Schmit. Efficient application representation for HASTE: Hybrid Architectures with a Single, Transformable Executable. *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pages 101–110, 2003.

[3] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 160–161, June 2007.

[4] E. Lübbers and M. Platzner. ReconOS: An RTOS supporting hard-and software threads. In *17th International Conference on Field Programmable Logic and Applications (FPL), Amsterdam, Netherlands*, 2007.

[5] Hayden Kwok-Hay So, Artem Tkachenko, and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis - CODES+ISSS '06*, page 259, 2006.

```
1:bash$ nf2_register_gen.pl --project reference_nic
2:bash$ mkbof -o reference_nic.bof -s reference_nic.symtab -t 4 reference_nic.bit
3:bash$ ls
  reference_nic.bit
  reference_nic.bof
  reference_nic.symtab
```

Figure 2: Generating the BOF file

```
1:bash$ ./reference_nic.bof &
[1] 9368
2:bash$ ps j
 PPID   PID  PGID   SID TTY   TPGID STAT  UID  TIME COMMAND
 1224  1226  1226  1226 pts/0  1399 Ss   1000  0:00 -bash
 1226  1399  1399  1226 pts/0  1399 R+   1000  0:00 ps j
 9367  9368  9368  9368 pts/0  1399 S    1000  0:00 reference_nic.bof
3:bash$ ls /proc/9368/hw/ioreg
 ...
 OQ_QUEUE_0_CTRL
 OQ_QUEUE_0_NUM_PKT_BYTES_STORED
 OQ_QUEUE_0_NUM_OVERHEAD_BYTES_STORED
 OQ_QUEUE_0_NUM_PKT_BYTES_REMOVED
 OQ_QUEUE_0_NUM_OVERHEAD_BYTES_REMOVED
 OQ_QUEUE_0_NUM_PKTS_STORED
 OQ_QUEUE_0_NUM_PKTS_DROPPED
 OQ_QUEUE_0_NUM_PKTS_REMOVED
 OQ_QUEUE_0_ADDR_LO
 OQ_QUEUE_0_ADDR_HI
 OQ_QUEUE_0_RD_ADDR
 OQ_QUEUE_0_WR_ADDR
 OQ_QUEUE_0_NUM_PKTS_IN_Q
 OQ_QUEUE_0_MAX_PKTS_IN_Q
 OQ_QUEUE_0_NUM_WORDS_IN_Q
 OQ_QUEUE_0_NUM_WORDS_LEFT
 OQ_QUEUE_0_FULL_THRESH
 OQ_QUEUE_1_CTRL
 OQ_QUEUE_1_NUM_PKT_BYTES_STORED
 OQ_QUEUE_1_NUM_OVERHEAD_BYTES_STORED
 OQ_QUEUE_1_NUM_PKT_BYTES_REMOVED
 OQ_QUEUE_1_NUM_OVERHEAD_BYTES_REMOVED
 OQ_QUEUE_1_NUM_PKTS_STORED
 OQ_QUEUE_1_NUM_PKTS_DROPPED
 ...
4:bash$ cat /proc/9368/hw/ioreg/OQ_QUEUE_0_NUM_PKTS_STORED
 2
```

Figure 3: Executing the BOF file