

IMPROVING USABILITY OF FPGA-BASED RECONFIGURABLE COMPUTERS THROUGH OPERATING SYSTEM SUPPORT

Hayden Kwok-Hay So, Robert W. Brodersen

Electrical Engineering and Computer Science,
University of California, Berkeley
email: {skhay, rb}@eecs.berkeley.edu

ABSTRACT

Advances in FPGA-based reconfigurable computers have made them a viable computing platform for a vast variety of computation demanding areas such as bioinformatics, speech recognition, and high-end digital signal processing. The lack of common, intuitive operating system support, however, hinders their wide deployment.

This paper presents BORPH, an operating system framework for FPGA-based reconfigurable computers with a goal to ease and accelerate development of high-level applications to run on these computers. It provides kernel support for FPGA resources by extending a standard Linux operating system. Users therefore compile and execute hardware processes on FPGA resources the same way they run software processes on conventional processor-based systems. The operating system offers run-time general file system support to hardware processes as if they were software. Furthermore, a virtual file system is built to allow access to memories and registers defined in the FPGA, which provides communication links with running hardware processes. Increased productivities have been observed for high-level application developers, who have few previous experiences in hardware design, to implement complex mixed software/hardware designs on a FPGA-based reconfigurable computer running BORPH.

1. INTRODUCTION

Advances in FPGA technologies have made FPGA-based reconfigurable computers (RC) a promising architecture to meet the increasing computational demand in research areas such as bioinformatics[1], speech recognition[2], and high-end digital signal processing[3]. However, despite the promising computing power, many high level application developers remain disinclined to develop their applications on these machines because of the substantial difficulties involved.

This work was funded in part by C2S2, the MARCO Focus Center for Circuit & System Solutions, under MARCO contract 2003-CT-888.

Many high-level application developers are not experienced FPGA hardware designers themselves. In addition to the inherent different computational model between software and hardware, we have observed the following three categories of difficulties commonly faced by them while developing applications on reconfigurable computers:

1. The lack of a high level design language and development environment that is both easy to learn and use, yet is able to retain high performance and high controllability.
2. The lack of a *common* and *intuitive* communication interface to and from FPGA fabrics.
3. The lack of a *central management scheme* that manages and coordinates among concurrent users and applications on a running RC system.

The need for a high level design language and methodology for RC is a well acknowledged issue that have drawn numerous research interests[4, 5, 6]. In this paper, we present BORPH, the Berkeley Operating system for ReProgrammable Hardware. The primary goal of this work is to improve usability of high performance FPGA-based reconfigurable computers by systematically addressing the remaining two issues through a common and intuitive OS framework.

The issue with data I/O capability includes a wide range of problems, from simply sending a single byte to a running FPGA, to transfer and synchronize data between software-hardware components, to capture and transfer gigabytes of sampled data for offline analyses. Most RC's have their own ad hoc mechanisms for transferring data in and out of FPGA, such as through a shared memory, or serial terminal connection. However, because of this ad hoc nature, each newly built RC must reengineer these I/O interfaces, wasting precious engineering resources. BORPH abstracts data I/O in standardized OS interfaces that are portable across multiple platforms. Consequently, application designers and RC users need not to readjust to different interfaces when experimenting with different RC's.

The lack of a central management unit in RC is a subtle but significant problem that deserves attention. As simple as it might seem, tasks such as querying the availability of FPGA resources, reserving FPGA resources *atomically*, and relocating designs to any available FPGA resource, are critical for mass deployment of RC in a concurrent multi-user environment. BORPH addresses these issues through centralized OS management of all FPGA resources of a system.

Most early FPGA systems use FPGA as a coprocessor to a host processing system[7]. In these systems, “operating system” took the role of a FPGA configuration loader. None of the run-time I/O support proposed in this paper was present. While recent systems use the term operating system to describe a load-time configuration modifier[8], or a run-time dynamic resource scheduler[9, 10], BORPH differentiates itself by focusing on improving usability of RC’s. We believe a *common* and *intuitive* OS framework is essential to warrant wide deployment of RC and foster collaborative community involvements in RC researches.

We will give an overview of BORPH in Sect. 2. In Sect. 3, we present key features of BORPH, with emphasis on how BORPH increases usability of our RC system. The implementation detail of BORPH is presented in Sect. 4. We report our experiences in using BORPH to facilitate a software-hardware codesign process in Sect. 5. Finally, we conclude this paper in Sect. 6.

2. BORPH OVERVIEW

BORPH is an operating system that extends a standard Linux system with integrated kernel support for FPGA resources. It treats FPGA resources as first-class computing resource in the same way a traditional OS treats a CPU. As a result, a user may spawn a process either as a software program running on a CPU, or as a hardware design running on a FPGA, or both. BORPH provides services to FPGA processes as if they were software, such as access to the general file system, `stdin` and `stdout` support. With the help of `stdout`, for example, a FPGA process can easily be debugged by performing “`printf`” to the console.

BORPH uses regions of FPGA fabric as computation unit to spawn hardware processes, similar to the way software processes are spawned to a processor. Each reconfigurable region is defined as a *hardware region* (`hwr`). Logically, it is the smallest unit of a RC that is managed by BORPH. Physically, it can be implemented as an entire FPGA in a multi-FPGA system, or a partially reconfigurable region within a FPGA.

Because of its close tie to the Linux OS, all Linux software can run in a BORPH system unmodified. Furthermore, since BORPH treats software and hardware processes equally, the two can communicate with each other naturally under BORPH via standard Linux services such as file pipe,

socket, or signals with helps of standard library functions. Users can interact with FPGA fabric the same way they interact with software processes, which has shown to have made our FPGA-based reconfigurable computer more accessible to novel users.

Some of the system service semantics are changed slightly from standard Linux implementations to accommodate the added FPGA support. For example, in standard Linux, the command “`./prog1 | ./prog2`” creates a software pipe that connects `stdout` of `prog1` to `stdin` of `prog2`, passing byte stream between them through system memory. In BORPH, if one of the two programs is a hardware process, the byte stream is buffered by the kernel and sent to the correct FPGA through a predefined packet network. If both programs are hardware processes, BORPH will attempt to create a direct connection between the two in hardware, such that no further OS intervention is needed for data transfer, thus eliminating the slowness of a processor.

In general, all OS services that cross user-kernel boundary for hardware processes are implemented by passing messages on this predefined packet network. As part of BORPH OS, this packet network is abstracted entirely from the user. As long as this message format is preserved, it is possible to extend BORPH, or port BORPH to another platform. This message passing system can be thought as a hardware system call interface.

3. KEY FEATURES OF BORPH

In this section, we will describe key features of BORPH from a user perspective to illustrate how BORPH improves usability of FPGA-based RC’s.

3.1. High-Level Design Flow

As previously reported[5], we have developed a high level design capturing system based on Simulink [11] and Xilinx System Generator [12]. Together with a rich in-house library and wrapper scripts, most detail about the hardware platform is abstracted from the user. For example, we have created a register block, `sw_register`, that users can incorporate in their Simulink designs. This block allows simple single word transfer to and from a user design the same way traditional memory mapped I/O registers behave on a processor system.

We have further enhanced this design flow to integrate with BORPH. We have extended our wrapper scripts to produce a BORPH Object File (BOF) instead of the usual FPGA configuration `.bit` file as final output. BOF files can be executed by users directly in BORPH. Besides FPGA configuration information, a BOF file encapsulates high level information about a user design such as locations and names of user defined registers or memory blocks. This information is used by the kernel when the BOF file is executed.

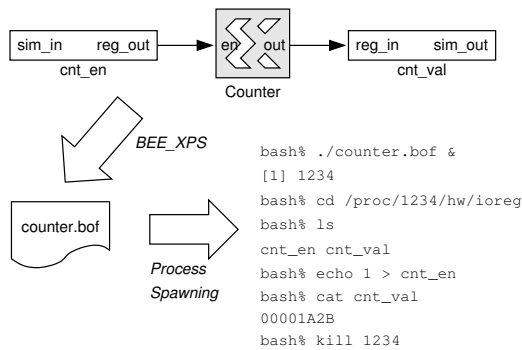


Fig. 1. From high-level Simulink design to a running BORPH process.

Through OS support, such as the `ioreg` interface described later, users can access these blocks at run time with minimum effort.

This integration between high-level design flow and the run time OS provides an environment that hides most hardware implementation details, leaving users focus in their high-level hardware/software designs. Fig. 1 shows the three major steps of getting a simple hardware design to run on a RC.

3.2. FPGA Design as a User Hardware Process

One of the key concept introduced by BORPH is its treatment of running FPGA designs as normal Linux processes.

A user starts a *hardware process* by executing a BOF file as if it is any other Linux executable, such as ELF. When a BOF file is run, the kernel examines hardware configurations encapsulated in that file. Based on this information, the kernel chooses and configures one or more suitable hwr's in the system for this BOF file. Updating necessary bookkeeping information, such as the kernel process table, and populating the `ioreg` file system is then performed. Finally, the user FPGA system is allowed to start running.

A running hardware process behaves almost identically to any other software process in a Linux system. Users can, for example, check the status of hardware processes using command such as `ps`. A hardware process can be terminated by command like `kill` or by simply pressing Ctrl-C. Fig. 2 shows a simple session of executing a BOF file, checking the value of an `ioreg`, and terminating the process.

With kernel support, hardware processes can be started by standard `fork` and `exec` system calls by software processes. Similarly, hardware processes can spawn software processes by passing messages to the kernel. To implement conventional software-centric designs, hardware accelerators can thus be started as needed by software. Moreover, BORPH allows hardware-centric designs, which launch software processes to handle exceptional conditions as needed.

```

1: bash% ./counter.bof &
[1] 2458
2: bash% ps
  PID TTY          TIME CMD
 2456 pts/4    00:00:00 bash
 2458 pts/4    00:00:00 counter.bof
 2507 pts/4    00:00:00 ps
3: bash% cat /proc/2458/hw/ioreg/cntval
A3B498E0
4: bash% cat /proc/2458/hw/ioreg/cntval
B289E906
5: bash% kill -9 2458
[1]+  Killed                  counter.bof
6: bash%

```

Fig. 2. Executing a BOF file containing a free running counter in BORPH. FPGA hardware is configured at prompt 1 and is unconfigured at prompt 5.

Alternatively, software and hardware can communicate in a client-server mode: A hardware process may process real-time signal continuously while software queries its result occasionally as needed.

In our experience, this simple hardware process abstraction has significantly lower the entry barrier for novel RC users.

3.3. Communicating with a Running FPGA Process

BORPH extends the `/proc` directory of Linux to include information about hardware processes. A directory called `/proc/<pid>/hw` is populated for each running hardware process with pid `<pid>`. There are currently two files and one subdirectory in this directory:

- `hw_region` — a file containing information about physical hwr locations a hardware process uses.
- `ioreg_mode` — a file containing the operating mode for `ioreg`, which can be binary or ASCII.
- `ioreg` — a subdirectory containing one virtual file for each I/O register (`sw_register` in Simulink), memory block (BRAM), off-chip memory or FIFO defined in a user design.

Reads and writes to virtual files in the `ioreg` directory are translated by the OS into actual readings and writings to the corresponding components in user designs. Each `ioreg` is defined with 4 properties: name, access mode, size and a physical location identifier. Access mode is a flag that specifies if it is readable, writable and/or seekable.

The design of this virtual file system shares a similar concept of [13] as a way for a user to interact with the running hardware design. For example, a simple command

```
echo 0 > /proc/1234/hw/ioreg/cnt_en
```

Type	Read/Write	Seekable	Size
Register	rw	no	4 bytes
On/Off Chip Memory	rw	yes	any
FIFO (from user)	read only	no	width×depth
FIFO (to user)	write only	no	width×depth

Table 1. Different types of `ioreg`

is adequate to disable the counter in Fig. 1. Besides single word registers, memories, both on-chip (BRAM) and off-chip (DRAM), are also supported through this interface. Virtual `ioreg` files map the entire content of their corresponding memories. It serves as a mechanism to share memory between software and hardware. User defined FIFO's are represented by virtual files that map all possible FIFO locations. They are not seekable, forcing each access to be from the beginning of the file, which corresponds to the head of FIFO's. Table 1 shows a summary of supported `ioreg`.

3.4. Access to Standard Input and Output

Conventional software programmer enjoys the ease of interacting with a computer user or other running processes through the system standard input (`stdin`) and output (`stdout`) file streams. BORPH extends this feature to allow hardware processes to have the same access to `stdin` and `stdout` by defining a standard message passing system between a user FPGA design and the main kernel. A hardware process can therefore perform simple, yet effective, debugging by printing messages to the screen. Also, user can effectively interact with the FPGA process through typing in the shell or piping from a file.

This support for standard I/O is our first step towards providing full general file system support for hardware processes, such that hardware processes can use standard Linux file system for general data input and storage.

Currently, we utilize this feature to implement a simple shell interface for all our hardware designs generated through our design flow. It complements the generic `ioreg` interface described in previous subsection by offering direct and application specific access to a running hardware process from within the FPGA.

4. IMPLEMENTATION

Currently, the BORPH system is implemented on a BEE2 hardware platform[14]. Briefly, each BEE2 board consists of 5 Xilinx Virtex-II pro xc2vp70 FPGA's. The center *control FPGA*, is connected to the 4 *user FPGA's* via both a shared 8-bit SelectMap bus and individually through a 50-bits direct connection. The 4 user FPGA's are connected in a ring with a 120-bits direct connection to its neighbor.

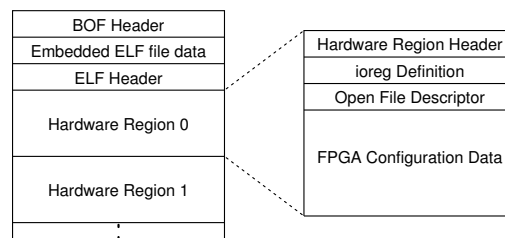


Fig. 3. BOF file format

4.1. The BORPH software kernel

The software kernel of BORPH is a modified version of Linux 2.4.30 kernel running on the left PowerPC core in the control FPGA. A standard Debian PowerPC root file system provides familiar Linux applications to the user. To provide kernel support for FPGA fabric, there are currently 4 major modifications made to the standard Linux kernel:

BOF file support. In order to encapsulate hardware configuration information in an executable file, we have developed a new Linux binary file format kernel module, `binfmt_bof`, to support BORPH Object Files (BOF). When a BOF file is executed, information in the file is parsed by this kernel module. This information is used for task such as configuring FPGA and populating `ioreg` interface. Fig. 3 shows basic definition of a BOF file.

Hardware Region (hwr). We have defined a set of kernel API to allow different hardware region types be loaded to BORPH as kernel modules. For example, each `hwr` kernel modules must implement a `configure` function that handles detail about configuring a particular `hwr` type. The rest of the kernel will then call the corresponding `configure` function based on `hwr` requirements embedded in a BOF file. This abstract `hwr` definition and the extensible kernel API allows BORPH to be ported to different RC relatively easily. In our first implementation, we have defined a `hwr` type, `b2fpga`, that corresponds to a user FPGA on BEE2. We are currently developing another type, `b2prmod`, that allows partial reconfiguration of a user FPGA.

Hardware Configuration and Resource Allocation. A new kernel thread `bkexecd` is defined to handle all `hwr` configurations. It is responsible for performing simple resource allocation by bookkeeping `hwr` usages. When a new BOF file is executed by the user, as long as the BOF file is relocatable, as in the case for `b2fpga`, it will only be spawned to a free FPGA. A user can override the behavior by preplacing a BOF to only be run on a specific FPGA. In case the requested FPGA is not available, a device busy error is returned to the user with standard Linux semantics.

Software/Hardware Communication. All communications between software and hardware are handled by the BORPH

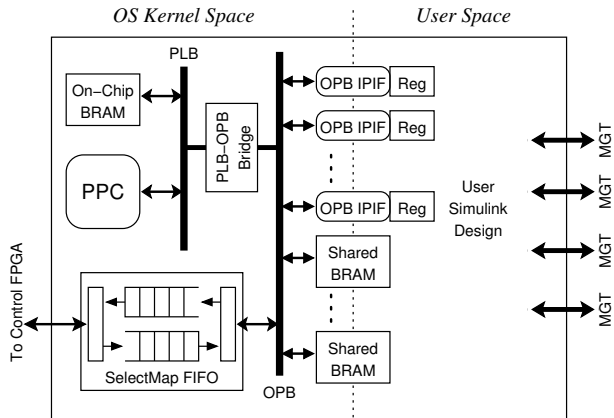


Fig. 4. Block diagram of a user FPGA. The processor system on the left of dotted line is inserted automatically by our design flow.

kernel through a standardized message passing network. Data transfers are initiated when (1) processes read/write virtual `ioreg` files; and (2) when hardware processes read/write standard I/O file streams. A kernel thread (`mkd`) is responsible for handling all read/write messages from hardware processes. These hardware requests are translated into kernel file reads/writes to the corresponding files. Similarly, user read/write requests to virtual `ioreg` files are translated automatically into packet messages that communicate with hardware processes.

4.2. The BORPH Hardware Kernel

In BEE2, the control FPGA and the parts of user FPGA's that communicate with the control FPGA are solely responsible for infrastructural support. They can thus be thought as part of BORPH's hardware kernel. The part of BORPH that runs on control FPGA is called `mk`, while the part that runs on user FPGA is called `uk`.

Fig. 4 illustrates our current implementation of a user FPGA. The processor system, `uk`, on the left hand side of the figure, is automatically inserted by our design flow when generating the corresponding BOF file from Simulink design sources. All user defined `ioreg`'s are connected to `uk` through standard bus connections. The software running in this processor system then coordinates the communication between `mk` and the user FPGA.

In this first implementation, since the entire user FPGA is reconfigured for each loaded user hardware process, `uk` is compiled with the user design statically into a BOF file. In the future, with help of dynamic reconfiguration, `uk` may be implemented as static module within user FPGA's, while new user designs are reconfigured as partial reconfigurable modules. This will eliminate the need for recompiling BOF files when the OS is updated.

Resource Type	Avail	OS	User	OS/Avail
Slice	33088	2969	11587	8.9%
RAMB16	328	34	96	10.3%
PowerPC	2	1	0	50.0%

Table 2. User FPGA resource used by OS vs. user.

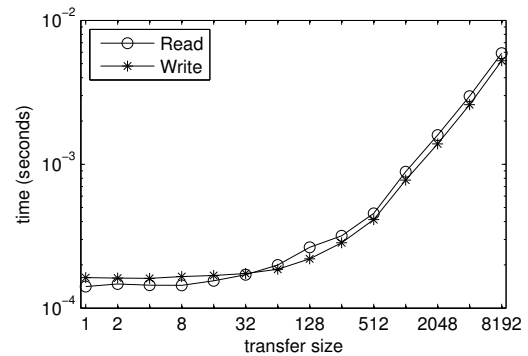


Fig. 5. Latency of accessing BRAM on a user FPGA via `ioreg` interface. Latency for each access size is measured by averaging 1000 standard C library `read` and `write` function calls from the control FPGA to a 8192 bytes BRAM located on a user FPGA.

4.3. Resource Utilization and Performance

Incorporating BORPH into our system inevitably consumes precious FPGA resources on user FPGAs. Table 2 shows the current utilization of `uk` on a user FPGA. On average, BORPH consumes only 10% of FPGA slices and RAMB16 blocks. With positive feedback from our users, this kind of utilization is acceptable as a first implementation.

Fig. 5 shows the latency of accessing an on-chip BRAM via BORPH's `ioreg` interface. When the access size is smaller than 128 bytes, read/write time is level at around $160 \mu\text{s}$, which is a result of OS kernel crossing overhead. As read/write size increases beyond 128 bytes, the access time increases linearly accordingly as a result of packetization. In our current implementation, user FPGA's communicate with control FPGA via a shared 8-bit bus running at 50MHz, which provides a theoretical bandwidth of 50 MB/s. The current `ioreg` performance of about 1.5 MB/s is using only a fraction of this theoretical maximum bandwidth.

5. EXPERIENCE WITH SOFTWARE/HARDWARE CO-DESIGN USING BORPH

This section describes our experience with implementing and testing a software-defined radio testbed design on BEE2 that runs BORPH. Like most FPGA system designs, this system consist of a hardware and software component. In

our case, our software team is physically located remotely in Germany, while the hardware team is located in the United States which presents unique challenges to us. Our sw/hw codesign methodology takes advantage of BORPH in the following ways:

- *Independent software and hardware testing.* The software and hardware components of the design communicate via `ioreg` interface of BORPH. Before sw/hw integration, we take advantage of the Linux file abstraction layer such that our software team can emulate responses from hardware by using “fifo files”. Our software team can therefore test their designs independently on their own Linux machines.
- *Remote log in.* To start the sw/hw integration process, our software team needs to access physical hardware on BEE2 remotely. The fact that BORPH is backward compatible with Linux allows it to run all standard Linux networking software. It allows simple `ssh` login through internet from Germany.
- *Seamless integration between sw/hw components.* The use of `ioreg` service as a standardized I/O interface eliminates the need for our software team to develop additional device drivers for sake of integration. The software code developed can start communicating with our hardware by simply changing `ioreg` file names in the code.
- *In-system debugging.* Software bugs that only show up during the sw/hw integration are difficult to locate and fix independently by either team. Since BORPH supports running standard software development tools such as `gdb` and `gcc`, we can debug the software with actual running hardware in-system.

The synergy between software and hardware in BORPH has shown to be invaluable in our experiences.

6. CONCLUSIONS AND FUTURE EXTENSIONS

In this paper, we have presented BORPH, an operating system framework for FPGA-based reconfigurable computers. It increases usability of RC by providing a systematic and intuitive OS interface for running and communicating with FPGA fabrics. The kernel level synergies between hardware and software processes have provided natural semantics for sw/hw codesign. Early feedback from users have indicated encouraging prospect of this abstraction.

Currently we are incorporating the use of dynamic reconfiguration in BORPH, which provides a natural kernel/user partition on user FPGA's. A more elaborated general file system support is being developed. We are also investigating the possibility of incorporating hardware process switching

into BORPH. Finally, we are trying to improve performance by using direct connections among user and control FPGA's.

7. ACKNOWLEDGEMENTS

This work cannot be completed without the endless effort by Pierre Droz and Andrew Schultz in developing many fundamental building blocks and infrastructures on BEE2 that BORPH is built upon. Also, many thanks go to Artem Tkachenko for his patient and hard work debugging and giving feedback on early versions of BORPH.

8. REFERENCES

- [1] S. Dydel and P. Bala, “Large scale protein sequence alignment using FPGA reprogrammable logic devices.” in *Proceedings of FPL'04*, 2004, pp. 23–32.
- [2] E. M. Ortigosa *et al.*, “FPGA implementation of multi-layer perceptrons for speech recognition.” in *Proceedings of FPL'03*, 2003, pp. 1048–1052.
- [3] S. M. Mishra *et al.*, “A real time cognitive radio testbed for physical and link layer experiments,” in *1st IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks*, Nov 2005, pp. 560–567.
- [4] A. Habibi and S. Tahar, “Design and verification of systemc transaction-level models,” *IEEE Trans. VLSI Syst.*, vol. 14, no. 1, pp. 57–68, Jan. 2006.
- [5] C. Chang *et al.*, “Rapid design and analysis of communication systems using the BEE hardware emulation environment,” in *IEEE Rapid System Prototyping Workshop*, 6 2003.
- [6] I. Page, “Closing the gap between hardware and software: hardware-software cosynthesis at oxford,” in *IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*. IEE, 1996, pp. 2/1–2/11.
- [7] R. W. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective.” in *DATE*, 2001, pp. 642–649.
- [8] G. B. Wigley, D. A. Kearney, and D. Warren, “Introducing reconfigme: An operating system for reconfigurable computing,” in *Proceedings of FPL'02*. Springer, 2002.
- [9] H. Walder and M. Platzner, “A runtime environment for reconfigurable hardware operating systems.” in *Proceedings of FPL'04*, 2004, pp. 831–835.
- [10] B. Mei *et al.*, “Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study.” in *DATE*, 2004, pp. 1224–1229.
- [11] “<http://www.mathworks.com/>”
- [12] “<http://www.xilinx.com/>”
- [13] A. Donlin *et al.*, “A virtual file system for dynamically reconfigurable FPGAs.” in *Proceedings of FPL'04*, 2004, pp. 1127–1129.
- [14] C. Chang, J. Wawrzynek, and R. Brodersen, “BEE2: A high-end reconfigurable computing system,” *IEEE Des. Test. Comput.*, vol. 22, no. 2, pp. 114–125, Mar. 2005.