A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH

Hayden Kwok-Hay So, Artem Tkachenko and Robert Brodersen
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{skhay, artemtk, rb}@eecs.berkeley.edu

ABSTRACT

This paper presents a hw/sw codesign methodology based on BORPH, an operating system designed for FPGA-based reconfigurable computers (RC's). By providing native kernel support for FPGA hardware, BORPH offers a homogeneous UNIX interface for both software and hardware processes. Hardware processes inherit the same level of service from the kernel, such as file system support, as typical UNIX software processes. Hardware and software components of a design therefore run as hardware and software processes within BORPH's run-time environment. The familiar and language independent UNIX kernel interface facilitates easy design reuse and rapid application development. Performance of our current implementation and our experience with developing a real-time wireless digital signal processing system based on BORPH will be presented.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*UNIX* C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces* D.4.0 [**Operating Systems**]: General—*UNIX*

General Terms: Standardization, Design

Keywords: Reconfigurable computers, Hardware process

1. INTRODUCTION

FPGA-based reconfigurable computers (RC's) are becoming viable computing architectures that promise to deliver super-computer class performance by computing both directly on FPGA hardware and on processors[6, 10, 1]. Their high performance to cost ratios have drawn vast interests in areas such as bioinformatic[8], speech recognition[12], and network security[15]. Developing applications on these RC's usually involve multiple design teams which, as observed by [13, 16], can benefit from an interface-based design methodology. Furthermore, since previous research in these areas mostly relied on super-computers or computer clusters as their primary computing platforms, the application designers have high degrees of variance in previous hw/sw codesign experiences. As a result, a hw/sw interface that is familiar and easy to understand will greatly facilitate the transition into hw/sw platforms like FPGA-based RC's.

While traditional hw/sw codesign researches have produced encouraging results in the area of hw/sw partitioning, cosimulate, cosynthesis, and co-verification, most of them rely on self-contained design environments that are based on their specific input languages or library API's[4, 9]. As a result, migrating existing software designs to RC platform using these traditional hw/sw codesign methodologies would have incurred major re-engineering efforts, including learning a new language and API, getting familiar with a new design environment and reimplementing existing designs in the new language environment.

Instead, an easy to use hw/sw interface that allows rapid application development and migration should be (1) familiar and intuitive to both software and hardware engineers; and (2) language independent. We achieve this goal by setting hardware/software boundary at the operating system kernel level.

1.1 HW/SW Co-Design in BORPH

In this paper, we present BORPH[1], an operating system designed specifically for reconfigurable computers. Under BORPH, hardware and software share the same familiar UNIX interface and the same level of support from the OS kernel. We introduce the concept of *hardware process*, which is the same as a normal UNIX process except its "program" is an FPGA hardware design instead of software program. Communications between hardware and software are accomplished through conventional UNIX inter-process communication (IPC) mechanisms, such as shared file, pipe, shared memory, signal, and message passing. Hardware processes have access to system resources as their software counterparts, such as the general file system, standard input, standard output.

By maintaining the hw/sw interface at the kernel level, BORPH provides a system that is language independent for both hardware and software. Software designs can be developed in any language development environment a designer is familiar with. For hardware designs to communicate with the kernel, BORPH defines a standard message passing network that resembles the software system call interface. This standardized network allows hardware designs be developed in any hardware language environment of choice.

[1]BORPH is an acronym for Berkeley Operating system for ReProgrammable Hardware

CODES+ISSS'06, October 22–25, 2006, Seoul, Korea.

1.2 Related Work

The work of UltraSONIC[17] shares a similar design philosophy as BORPH in providing a unifying coarse-grain hardware and software component interface. In their system, software and hardware tasks share the same interface into a runtime task scheduler. POLIS[4] provides a common CFSM based framework that can be synthesized to either software or hardware. However, both require the input design be specified in a specific language. Cray's XD1, utilizes FPGA's as application accelerators[1]. It provides a set of Linux software API's that resembles file I/O functions in C for FPGA communications. It does not provide kernel support for FPGA as in the case of BORPH.

BORPH by itself is not a complete system to perform typical hw/sw codesign tasks such as partitioning, cosynthesis, cosimulate, or verification. Instead, by providing basic OS services, it acts as a platform on which these tasks can be carried out.

We will describe the general concept and interfaces of BORPH in Sect. 2. Our current implementation and its performance is discussed in Sect. 3. We describe our hardware and software codesign experience of a real-time radio design using BORPH in Sect. 4. We will conclude the paper in Sect. 5.

2. BORPH: THE OPERATING SYSTEM

BORPH is an operating system designed for reconfigurable computers. It extends a standard Linux kernel to include support for FPGA's in a RC. Instead of treating FPGA's as coprocessors, BORPH treats FPGA's in the system as normal computational resources. User processes can therefore be either software programs running on processors, or they can be hardware designs running on FPGA's. We term a running design on FPGA a *hardware process*. BORPH maintains a consistent UNIX interface for both software and hardware processes. Therefore, to the rest of the system, communicating with a hardware process is no different from communicating with a normal UNIX process. This homogeneous handling of hardware and software in the kernel forms the foundation of coarse grain hardware/software codesign boundary. Fig. 1 depicts a BORPH conceptual block diagram.

In this paper, we focus on BORPH's essential concepts and their implementations related to hw/sw codesign. We will present the general concept of *hardware process*, and two hw/sw communication methods: the *ioreg* interface and the *file I/O* interface in this section. Our particular implementation of these interfaces are described in Sect. 3. More information on BORPH's conceptual design can be found in [14].

2.1 Running FPGA as a Hardware Process

In conventional OS terminologies, a process is a running copy of a program in a processor. BORPH extends this idea to FPGA, defining a hardware process as a running copy of a FPGA design. BORPH supports a new binary format, called a BORPH Object File (BOF), as shown in Fig. 3. A BOF file encapsulates, among other information, configuration for FPGA's. Executing a BOF file causes the kernel to configure FPGA's accordingly. Fig. 2 shows a simple transcript of executing a BOF file. Once a BOF file is running in the system, it is treated as a normal Linux process. For example, its status can be checked by standard command

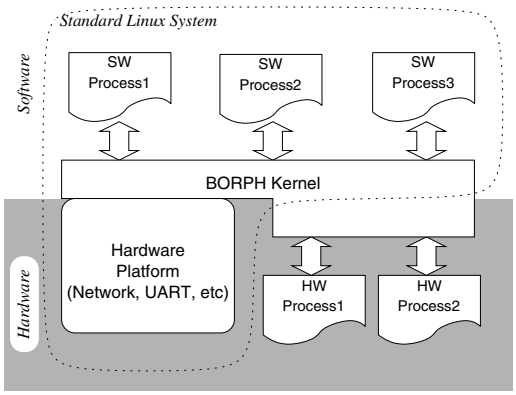

Figure 1: BORPH extends a traditional Linux system with hardware process support. HW/SW processes share the same I/O interface. Dotted line denotes a standard Linux system where the kernel and all user processes are software.

```
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
```

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

like `ps`. Similar to a software process, a hardware process can be terminated either by external UNIX signals, or it can terminate itself by sending a message to the BORPH kernel that is equivalent to the `exit` system call.

2.2 ioreg Interface

BORPH's ioreg interface encapsulates conventional memory mapped I/O concept with a virtual file system interface similar to that presented in [7]. Communication between hardware and software usually involves defining a set of special hardware registers that are memory mapped by software. BORPH encapsulates this common design practice by supporting it systematically via its ioreg interface.

BORPH extends the standard Linux `/proc` directory to

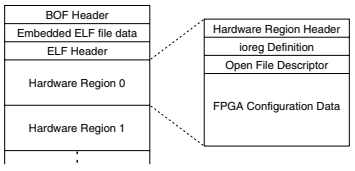

Figure 3: Simplified BOF file format

Type	R/W	Seekable	Size
Register	rw	no	4 bytes
On Chip Memory	rw	yes	any
Off Chip Memory	rw	yes	any
FIFO (from user)	r/o	no	width×depth
FIFO (to user)	w/o	no	width×depth

Table 1: Types of ioreg

include hardware specific information about a hardware process. When a hardware process is started, the kernel populates a special `hw` directory under that process ID. In this directories are virtual files that provide information, such as the physical FPGA location, of this hardware process. There is also a subdirectory named `ioreg`. Each virtual file in this directory corresponds to one ioreg embedded in the BOF file. Reading from, or writing to these files causes the kernel to read/write to the corresponding ioreg physically located inside a hardware process, using BORPH's standard message passing network.

As an example, the design in Fig. 2 contains a free running counter that stores its output in an ioreg register named `cntval`. As shown in the figure, once the design is running in the system as a hardware process, the value of `cntval` can be read by any standard Linux program, such as `cat`, from the corresponding `/proc` entry. Being able to access information from a running hardware easily allows the ioreg interface be used for both debugging and actual data transferring.

Besides simple single word register, the ioreg interface is also used to provide access to on-chip FIFO, on-chip memory, as well as off-chip memory that a hardware process have access to. Table 1 shows the supported hardware construct by this interface and their differences when exported as virtual files in BORPH. Accessing these hardware constructs is similar to accessing a single word register described above. For example, to read an on-chip memory, a user may perform a simple shell command

```
bash$ cp /proc/123/hw/ioreg/Shared_Memory ~/
```

or similarly in a C program:

```
fread(buf, mem_size, 1, MEM_FILE);
```

In our current design, memory space between software and hardware is separated. The only way to access memories attached to, or embedded in, a FPGA is through this ioreg virtual files. The concept of shared memory between software and hardware is being developed.

2.3 File I/O

Hardware processes in BORPH have access to the general Linux file system just like their software counterparts. Since hardware processes are not running in a processor, reading from and writing to files are done BORPH's standard message passing system.

Similar to conventional software process, when a hardware process is started, three standard I/O files, `stdin`, `stdout`, and `stderr`, are automatically opened. With `stdout`, for example, a hardware process can print messages to the screen for debugging purpose. Such "debug by printing" capability is previously only available during simulation. For example, we currently take advantage of this to run a small shell program directly from a FPGA which allows user interaction with a hardware design.

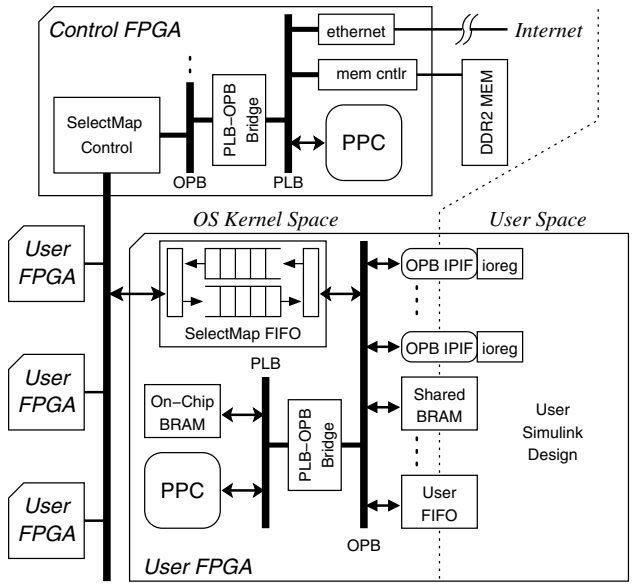

Figure 4: Block diagram of BORPH system on a BEE2 compute module.

Another use of standard file I/O is to chain multiple processes through *pipes*. Because of the standardized file interface, a user can freely combine software and hardware processes on either end of a pipe. For example, given a transmitter (`tx.bof`) and receiver (`rx.bof`) implemented on FPGA, the following command can test an entire radio transceiver chain:

```
cat data.in | tx.bof | rx.bof > data.out
```

Besides standard I/O, hardware processes can access any other file in the system as well.

3. CURRENT IMPLEMENTATION

This section describes our proof of concept implementation of BORPH on a BEE2 compute module[6]. Each BEE2 compute module contains 5 Xilinx Virtex-II pro xc2vp70 FPGA's. The center *control FPGA*, handles all system related functions. It is connected to the remaining 4 *user FPGA's* through a shared 8-bits configuration bus, called SelectMap bus, running at 50 MHz, and to each user FPGA independently with a 50-bits direct connection. The SelectMap bus doubles as a low bandwidth communication bus after a user FPGA is configured. BORPH currently communicates with all user FPGA's using this bus. Fig. 4 shows a simplified block diagram of the internal of the control FPGA and a typical configuration of a user FPGA created by our Simulink design flow.

BORPH kernel is a modified version of a Linux 2.4.30 kernel running on the left PowerPC 405 core in the control FPGA. The hardware core responsible for driving the SelectMap bus is attached to the processor's on-chip peripheral bus (OPB) via the processor local bus (PLB). Each user FPGA has a 128 bytes FIFO on the receiver side of the SelectMap bus.

As mentioned before, all communications between BORPH kernel and user FPGA's are accomplished by a standard message passing system. These message packets have a sim-

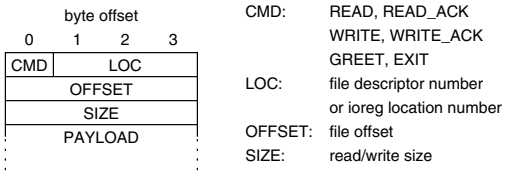

Figure 5: A simple packet format for message exchange with BORPH kernel.

ple but consistent format as shown in Fig. 5. Our current implementation of this packet network is built on top of the SelectMap bus connection. On the control FPGA, a kernel thread, `mkd`, is responsible for answering all user FPGA interrupts, and delivering messages to the corresponding waiting process.

3.1 Hardware Process Creation

A hardware process is created when an `exec` system call is received by the BORPH kernel on a BOF file. The request is then passed on to a kernel thread, `bkexecd`, for the actual configuration. Based on the BOF file header, one or more suitable FPGA's are chosen and configured accordingly using the SelectMap bus.

In our current implementation, creating a hardware process takes about 900 ms, while creating a normal software process takes about 40 ms. Since the theoretical minimum time required to start a hardware process is the sum of the time to start a software process and the time to configure a FPGA, which is 65 ms in our case, we are currently 9 times slower than the theoretical minimum. Preliminary investigation indicates that the PLB-to-OPB bridge on the control FPGA is limiting the bandwidth of streaming configuration data from the file system to a user FPGA.

Improving the reconfiguration speed, as well as incorporating hardware context switching using partial reconfiguration are planned for future implementations.

3.2 Reading/Writing ioreg Files

When a user reads or writes to a virtual ioreg file, the request is translated by the kernel into a message that is sent to the corresponding FPGA. The unique identification number of the ioreg is sent in the LOC field of the message. Each ioreg read (write) request is answered by the user FPGA by a read (write) acknowledge message, indicating the number of bytes read (written), or a negative value that indicates error condition. Adhering to the standard UNIX semantics for file read (write), the return value is passed directly back to the user process that initiated the request.

Fig. 6 shows the performance of reading/writing an on-chip memory that is exported as an ioreg file, using different read/write sizes, s. The transfer time remains low until s increases beyond about 64 bytes. Since there is no buffering in the file system level, the time needed for the operation is determined solely by data movement time, which includes memory copy time and hardware data transfer time. The effect of a small data cache (16k bytes), combined with a small 128 bytes SelectMap FIFO are contributing factors for the slowing down. Nonetheless, for large enough s, the speed levels at about 1.38 MBs[2] for both read and write.

[2]mega-bytes per second

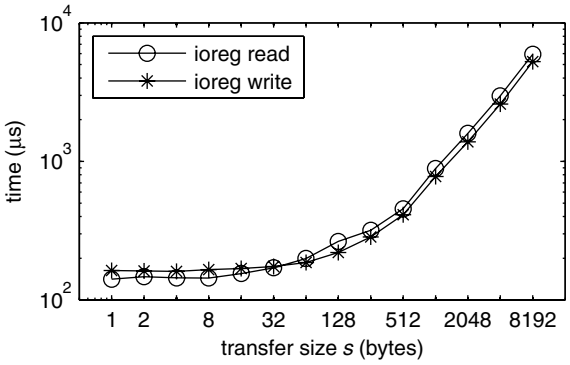

Figure 6: Performance of reading/writing on-chip memory on a user FPGA using ioreg interface.

3.3 General File I/O from Hardware Processes

Hardware processes initiate file I/O by sending messages to the BORPH kernel using the format described in Fig. 5, with the LOC field denoting Linux opened file descriptor number. A kernel thread, called a *fringe*, is created for each opened file to handle the actual file operation on behalf of the FPGA hardware. When a packet is received from the user FPGA, `mkd` is woken up, which in turn wakes up the corresponding fringe based on the packet. The fringe then reads the file on behalf of the hardware process, blocking as needed. The read data is sent as the payload of a read acknowledge packet back to the user FPGA. Since most file writes complete successfully without blocking, as an optimization, no fringe is created if a file is opened as write only. Instead, `mkd` writes to the file on behalf of the hardware process directly, thus eliminating one context switch for each file write. Furthermore, no WRITE_ACK is sent to the user FPGA.

For purpose of benchmarking hardware file I/O performance, an FPGA design, `stdloop.bof`, and an equivalent software C program, `pipetok`, are created. Both designs repeatedly read s bytes of data from its stdin, and write the data back to stdout, until the end of file is reached.

First, to determine the overhead of file I/O from a user FPGA hardware process, the two programs are run as follow:

```
bash$ stdloop.bof < datafile > outfile
bash$ pipetok < datafile > outfile
```

The time to complete file I/O operations of various sizes, s, are shown in Fig. 7. Hardware processes are about 5 times slower than software for small file reads. The gap increases as s increases. The time for interrupt handling, context switching to `mkd`, and then to the correct fringe on the control FPGA all contributes to this overhead. On the other hand, small hardware file writes are faster than their software counterparts due to the elimination of WRITE_ACK message from control FPGA. For large writes, the small SelectMap FIFO size of 128 bytes causes data transfer speed the limiting factor.

To determine the overhead of communication in typical mixed hardware software computation scenarios, a second benchmark is perform where both programs are run in a piped process chain as follow:

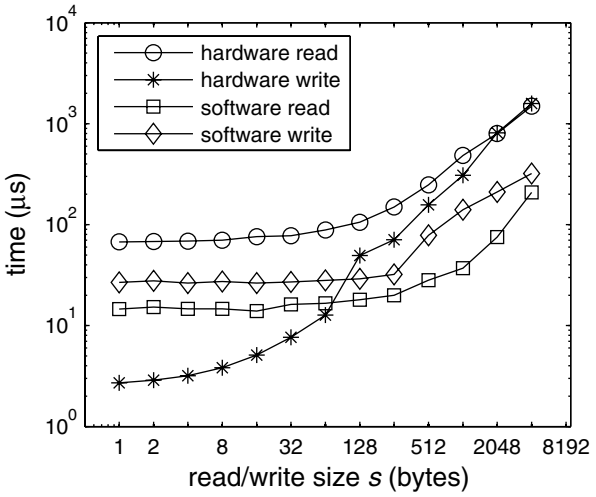

Figure 7: Hardware process file I/O performance.

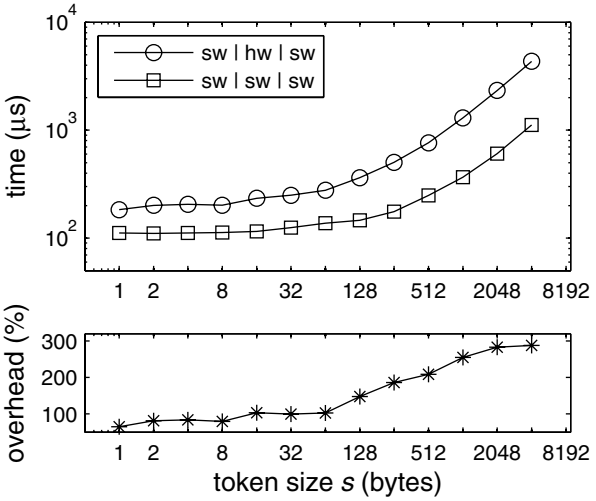

Figure 8: Comparing software piped process chain with a mixed hardware/software chain.

```
bash$ sendtok | stdloop.bof | recvtok
bash$ sendtok | pipetok | recvtok
```

`sendtok` and `recvtok` are programs that repeatedly send and receive token of s bytes from their stdout and stdin respectively. The time for `recvtok` to receive an entire s bytes token from `sendtok` in both cases are shown in the top half of Fig. 8. Communication overhead, which is the extra time needed for the mixed hw/sw pipe over the software pipe, is plotted at the bottom half of the diagram. Overhead remains at about 60% level for $s < 128$ and increases significantly for $s \geq 128$ mainly as a result of the limited SelectMap FIFO size.

4. CASE STUDY: A REAL-TIME WIRELESS SIGNAL PROCESSING SYSTEM

In this section, we present our experience designing a real-time wireless signal processing system for our cognitive radio project[11] to illustrate how various features of BORPH facilitate our development and testing process.

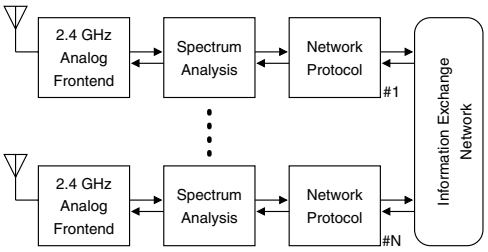

Figure 9: Cognitive radio testbed system

Cognitive radios are smart radios that take advantage of under-utilized licensed spectrum for opportunistic tranceiving. In order to prevent interference to licensed primary users of the spectrum, a variety of techniques have been proposed for reliable sensing and non-interfering use of the spectrum. Our system is designed to validate those techniques. Fig. 9 depicts our overall system design that involves multiple cooperative cognitive radios.

BORPH allows us to test our design in real-time remotely by two physically separated hardware and software groups.

4.1 HW/SW Partitioning

As mentioned before, we do not have an automatic partitioning system in place. Therefore, hw/sw partitioning for this system is done loosely based on the standard network stack, where the physical layer is implemented in hardware and the higher layer implemented in software.

The "cognitive" nature of the radio adds complexity to this hw/sw partitioning because the link and MAC layers must adapt according to changes in the physical spectrum. In our case, the task of real-time spectrum analysis is performed by FPGA hardware, leaving the decision making protocol to software. Software of each radio also communicate with each other using standard internet to form a cooperative information network.

4.2 Development Environment

Software is developed with conventional C development tools, as well as in Simulink[2], which is also the program of choice for our hardware design flow[5]. To work with BORPH, we have augmented standard Xilinx System Generator[3] with our in-house Matlab program. It is responsible for inserting all interfacing logic for a user FPGA, as shown on the left hand side of the dotted line in Fig. 4. It also generates FPGA configuration file using Xilinx backend tools, which is then used to produce a BOF file for run-time execution.

4.3 Communication and Synchronization

All communications between the spectrum sensing hardware and the software protocol stacks are done via BORPH's ioreg interface. Two 8192 bytes shared memory are exported as ioreg virtual files for data communication with software. In addition, more than 20 single word registers are defined. Most of them are standard registers that control design parameters such as RF channel select, amplifier gain control, etc. Some ioreg registers, however, are used solely for synchronization purpose. For example, each shared memory is guarded by a pair of `enable` and `ready` register. The `enable`

register is used by software to notify hardware its intention to read memory. When the data in the shared memory is ready, the hardware asserts the corresponding `ready` register. This two-way handshaking mechanism forms the basis of simple synchronization between software and hardware processes.

The file I/O capability of a hardware process is used to implement a low level debugging shell. It provides an additional way to debug the running FPGA.

4.4 System Testing

Hardware/software cosimulation is done within the Simulink environment. Each supported ioreg construct shown in Table 1 is modeled by a custom Simulink block. Besides allowing cycle-accurate simulation within Simulink, these blocks are synthesizable to FPGA. Unfortunately, cosimulation provides very limited information because of its limited speed. As a result, our system is often tested *in situ*.

Our hardware design is tested in real-time together with its associated radio frontend. Instead of artificial data, real-time data is fed using signal generators as testbench. The analyzed result is stored in on the on-chip memory for software to display and debug. All test setups and data transfer are done via BORPH's ioreg.

Our software design is developed off-site. BORPH provides a remote testing environment for our protocol group who doesn't have physical access to the hardware. With BORPH, our software team independently develops the protocol stack without the presence of the hardware by emulating it with software processes. As development progress, they then remote log in to the physical hardware for mixed hw/sw testing with a simple swap of hardware process in place of the emulating software process. Because BORPH is running on with a fully functional Debian root file system, all the necessary software development tools, such as `gdb` are available for debugging.

5. CONCLUSION

In this paper, we have described our hw/sw codesign experience using BORPH, an operating system designed for reconfigurable computers. BORPH encapsulates FPGA hardware designs as running hardware processes and provides conventional OS services such as file system support to them. By setting the hw/sw boundary at OS kernel level, BORPH provides an unified hw/sw runtime environment with a familiar UNIX interface. It extends the familiar notion of process-level parallelism to include both hardware and software. Designing with BORPH is not tied to any particular language, and all existing software for Linux can be reused as needed.

We have described our current implementation of BORPH on a BEE2 platform and our experience designing with it. Despite the sub-optimal performance of our current implementation, it has served as a proof of concept demonstrating the usefulness of the BORPH design concept. Since implementation detail of BORPH is independent of the OS interface, performance is expected to be improved through future re-implementations without affecting existing user designs.

Currently, we are further exploring the semantics for hardware processes, such as blocking, parallel file system access, and software/hardware notification mechanisms. Partial reconfiguration of user FPGA is also being developed to further enhance kernel/user space separation. Moreover, we are developing a direct in-system hardware process debugging methodology based on BORPH.

6. ACKNOWLEDGEMENT

We would like to thank Pierre Droz and Andrew Schultz for their effort in developing many fundamental building blocks and infrastructures on BEE2 that this work is built upon. This work was funded in part by C2S2, the MARCO Focus Center for Circuit & System Solutions, under MARCO contract 2003-CT-888

7. REFERENCES

[1] http://www.cray.com/products/xd1/.

[2] http://www.mathworks.com/.

[3] http://www.xilinx.com.

[4] F. Balarin et al. *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[5] C. Chang et al. Rapid design and analysis of communication systems using the BEE hardware emulation environment. In *IEEE Rapid System Prototyping Workshop*, June 2003.

[6] C. Chang, J. Wawrzynek, and R. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Des. Test. Comput.*, 22(2):114–125, Mar. 2005.

[7] A. Donlin, P. Lysaght, B. Blodget, and G. Troeger. A virtual file system for dynamically reconfigurable FPGAs. In *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, pages 1127–1129, 2004.

[8] S. Dydel and P. Bala. Large scale protein sequence alignment using FPGA reprogrammable logic devices. In *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, pages 23–32, 2004.

[9] A. Habibi and S. Tahar. Design and verification of SystemC transaction-level models. *IEEE Trans. VLSI Syst.*, 14(1):57–68, Jan. 2006.

[10] T. Hamada et al. Progrape-1: A programmable special-purpose computer for many-body simulations. In *6th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '98), 15-17 April 1998, Napa Valley, CA, USA*, pages 256–257, 1998.

[11] 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*, pages 560–567, Nov 2005.

[12] E. M. Ortigosa et al. FPGA implemenation of multi-layer perceptrons for speech recognition. In *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal*, pages 1048–1052, 2003.

[13] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 178–183, New York, NY, USA, 1997. ACM Press.

[14] H. So and R. W. Brodersen. Improving usability of FPGA-based reconfigurable computers through operating system support. In *16th International Conference on Field Programmable Logic and Applications (FPL'06)*, 2006.

[15] Y. Sugawara, M. Inaba, and K. Hiraki. Over 10gbps string matching mechanism for multi-stream packet scanning systems. In *Field Programmable Logic and Application, 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, pages 484–493, 2004.

[16] P. van der Wolf et al. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, New York, NY, USA, 2004. ACM Press.

[17] T. Wiangtong, P. Y. K. Cheung, and W. Luk. A unified codesign run-time environment for the ultrasonic reconfigurable computer. In *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, pages 396–405, 2003.