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

HAYDEN KWOK-HAY SO and ROBERT BRODERSEN
University of California, Berkeley

This paper explores the design and implementation of BORPH, an operating system designed for FPGA-based reconfigurable computers. Hardware designs execute as normal UNIX processes under BORPH, having access to standard OS services, such as file system support. Hardware and software components of user designs may, therefore, run as communicating processes within BORPH's runtime environment. The familiar language independent UNIX kernel interface facilitates easy design reuse and rapid application development. To develop hardware designs, a Simulink-based design flow that integrates with BORPH is employed. Performances of BORPH on two on-chip systems implemented on a BEE2 platform are compared.

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

General Terms: Standardization, Design

Additional Key Words and Phrases: FPGA, reconfigurable computers, hardware process, BORPH

ACM Reference Format:

So, H. K.-H. and Brodersen, R. 2008. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH, ACM Trans. Embedd. Comput. Syst. 7, 2, Article 14 (February 2008), 28 pages. DOI = 10.1145/1331331.1331338 http://doi.acm.org/10.1145/1331331.1331338

1. INTRODUCTION

FPGA-based reconfigurable computers (RCs) are becoming viable computing architectures that promise to deliver supercomputer class performance by

The authors wish to acknowledge the contributions of the students, faculty, and sponsors of the Berkeley Wireless Research Center, the National Science Foundation Infrastructure, Grant No. 0403427. This work was funded in part by the FCRP Focus Center for Circuit & System Solutions (C2S2), under contract 2003-CT-888.

Authors' addresses: Hayden Kwok-Hay So and Robert Brodersen, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720; email: {skhay,rb}@eecs.berkeley.edu.

computing both directly on FPGA hardware and on processors[Chang et al. 2005; Hamada et al. 1998]. Their high performance to cost ratios have drawn vast interest in areas such as bioinformatics[Dydel and Bala 2004], speech recognition[Lin et al. 2007; Ortigosa et al. 2003], network security[Sugawara et al. 2004], as well as embedded systems that demand high computational power at a manageable cost. To effectively leverage the computational power of FPGAs, two important issues must be addressed: (1) runtime operating system support; and (2) hardware/software interface.

1.1 Runtime Operating System Support

Sufficient runtime operating system support is essential for successful deployments of modern FPGA-based reconfigurable computers. Previous works in designing operating systems for FPGA-based systems focused mostly on the job of hardware task scheduling. However, modern FPGA-based systems demand features beyond basic task scheduling, such as Internet access, file system access, home network integration, and sophisticated user interaction mechanisms. In order to effectively support these features, we believe OS for modern FPGA-based systems should not only provides seamless integration of hardware and software tasks, but also provides traditional OS runtime support services, such as general file system access, to *both* hardware and software tasks. It should support commodity software applications, hardware/software applications, as well as hardware-only applications within a unified framework.

1.2 Hardware/Software Interface

Developing applications on FPGA-based RCs usually involve multiple HW/SW design teams which, as observed by [Rowson and Sangiovanni-Vincentelli 1997; van der Wolf et al. 2004], can benefit from an interface-based design methodology. To accommodate the hardware nature of FPGAs and to facilitate HW/SW codesign, such interface must be common to both hardware and software.

While traditional HW/SW codesign researches have produced encouraging results in the area of HW/SW partitioning, cosimulate, cosynthesis, and coverification, most of them rely on self-contained design environments that are based on their specific input languages or library APIs [Balarin et al. 1997; Panda 2001]. As a result, migrating existing hardware or software designs to a new RC platform using conventional codesign methodologies would have incurred major reengineering efforts, including learning a new language and API, getting familiar with a new design environment, and reimplementing existing designs in the new language environment.

Furthermore, the use of FPGA has attracted enormous interest in computational demanding fields that traditionally rely on processor-based supercomputers or computer clusters. To facilitate rapid migration of existing software designs to FPGA-based systems, a hardware interface that is familiar to software engineers is highly desirable. We believe an easy to use HW/SW interface that allows rapid application development and migration should be (a) familiar and intuitive to both software and hardware engineers with various degree of HW/SW codesign experience; and (b) language independent.

1.3 BORPH Overview

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 a hardware process and the rest of the system are accomplished through conventional UNIX interprocess communication (IPC) mechanisms, such as shared file, pipe, 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.

As an operating system that provides runtime support to FPGA-based reconfigurable computers, BORPH addresses both aforementioned requirements for successful FPGA-based RCs deployments within the same framework.

1.3.1 *Runtime Operating System Support for FPGA Designs.* BORPH models an executing instance of FPGA application as a *hardware process*. A hardware process behaves, in most aspects, identical to a normal UNIX process, except it is executed on FPGA hardware instead of the controlling processor. With the notion of hardware process established, BORPH systematically provides runtime support to FPGA designs through standard UNIX semantics. For example, BORPH extends conventional file I/O semantics to provide data I/O capabilities to FPGA applications.

All kernel/user communications are accomplished through a message-passing network. Furthermore, a set of hardware system libraries is developed to shield an FPGA application designer from the complexities involved with kernel/user interactions. With all complexities of I/O and other system operations handled by the BORPH kernel, application designers may, therefore, devote their efforts to realizing their applications in FPGA.

1.3.2 *Hardware/Software Interface at Process Level.* BORPH's model of running FPGA applications as hardware processes provides a new dimension to the hardware/software interface problem by maintaining this hardware/software application interface at the UNIX process level.

By hiding any differences between hardware and software processes, BORPH creates a homogeneous HW/SW runtime system. Instead of relying on ad-hoc and system-dependent methodologies, hardware and software processes communicate with standard UNIX interprocess communication (IPC) semantics, such as file I/O and signals. The peer-to-peer nature between software and hardware allows both conventional software-centric, as well as hardware-centric and hardware-only application development methodologies.

At the same time, the user/kernel boundary of BORPH provides an interface that is independent of user design language. Consequently, software and hardware can both be developed at the language environment a designer is familiar with.

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

Moreover, the UNIX environment of BORPH provides a familiar system to both software and hardware engineers, thus lowering the barrier to entry into FPGA computing. It also provides invaluable backward compatibilities for migrating existing software designs to FPGA-based reconfigurable computers.

1.4 Related Work

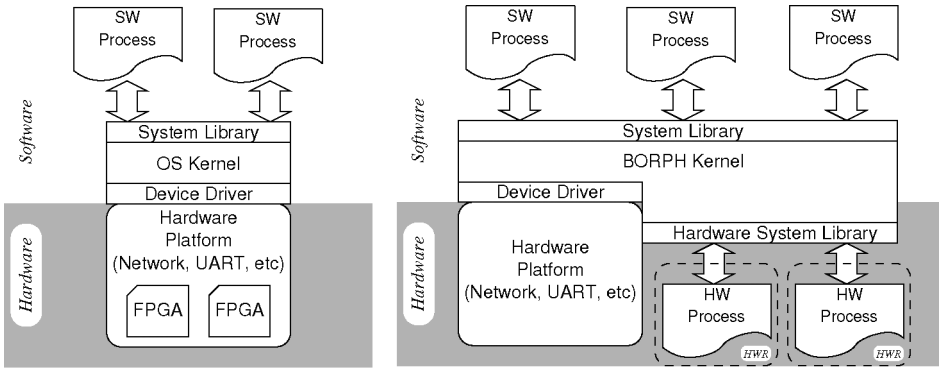
A number of research projects have approached the task of designing operating system for FPGA-based reconfigurable computers [Danne et al. 2006; Wigley et al. 2002; Walder and Platzner 2004; Mei et al. 2004]. All of them are devoted to the problem of dynamic FPGA resource allocation, memory sharing, or virtualization between software and hardware tasks on FPGA-based systems. We are not aware of any prior work that systematically offer runtime support directly to hardware processes as does BORPH. Furthermore, the use of UNIX semantics for modeling running FPGA designs instead of abstract "task" concept is unique to BORPH.

On the other hand, most commercial FPGA-based reconfigurable computers[Cray; XtremeData; DRC computer; Celoxica] are managed by off-the-shelf operating systems such as Linux and VxWorks. FPGAs on these systems are used mainly as software accelerators. Software and FPGAs communicate through conventional device driver layer while FPGA designs must utilize vendor-specific libraries with custom APIs. As a result, even if machines from different vendors are constructed using identical FPGAs, the inconsistent system interface prevent designs targeting one machine be easily ported to another. Such an inconsistent system interface greatly hinders collaborations among FPGA researchers.

In terms of hardware/software interface, the work of UltraSONIC [Wiangtong et al. 2003] 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 [Balarin et al. 1997] 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.

The main contribution of BORPH is that by leveraging conventional UNIX semantics to FPGA-based reconfigurable computing, it provides a unique, unified environment for both FPGA and software application designers. The UNIX semantics is familiar to developers across many research domains, thus lowering the barrier-to-entry into FPGA-based reconfigurable computing. Furthermore, since BORPH is implemented as an extended Linux kernel, a BORPH-managed system may leverage all commodity Linux software applications for developing, testing, benchmarking, and deploying FPGA applications.

We will first describe the general concept and interfaces of BORPH in Section 2. Then, we will describe our Simulink-based hardware design flow and how it integrates with BORPH in Section 3. Section 4 describes our current BORPH implementation on a BEE2 platform. In Section 5, we will report the performance of our two different hardware implementations and the lesson we have learned in the process of developing the second version. Section 6



(a) Traditional coprocessor-based FPGA system: SW/HW has master-slave relationship.

(b) BORPH-managed FPGA system: SW/HW has a peer-to-peer relationship.

Fig. 1. Two possible ways to organize FPGAs in a system.

illustrates actual usage model of BORPH through three different application examples. We will conclude the paper and discuss future direction of this research in Section 7.

2. BORPH: THE OPERATING SYSTEM

BORPH is an operating system designed for reconfigurable computers. It extends a standard Linux kernel to include support for FPGAs in a RC. The major difference between BORPH and other OS's for FPGA systems is that instead of treating FPGAs as coprocessors, BORPH treats FPGAs in the system as first-class computational resources.

Figure 1a illustrates the traditional way of managing FPGA resources on a single processor system. In such system, the operating system kernel acts as a layer between software and hardware. Running as software programs on the central processor, user applications must rely on the OS kernel via a set of system libraries and device drivers to communicate with the underlying hardware platform, such as a graphical display. Most systems treat FPGAs and other reconfigurable hardware resources as part of the same underlying hardware platform. Consequently, to communicate with any user hardware design running on these reconfigurable resources, one must employ the same mechanism as described before. Software and hardware programs therefore form a *master–slave* relationship, implying a software-centric design methodology.

On the other hand, BORPH logically separates reconfigurable hardware resources that are used for user applications, such as FPGAs, from the underlying hardware support platform. BORPH denotes these resources *reconfigurable hardware regions* (HWRs). Figure 1b illustrates this concept. We term a hardware design running on such HWR a *hardware process*. A hardware process communicates with the kernel through a predefined message-passing network that resembles software system calls. In practice, a layer of hardware system library will be responsible for the message-passing protocol. This layer can be thought as functions provided by a system VHDL package, or a system library

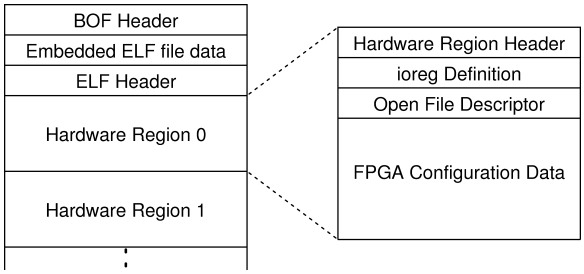


Fig. 2. Simplified BOF file format.

blocks, as in our current implementation. It is this layer that provides hardware processes with UNIX system services such as file system support.

By maintaining a consistent UNIX interface for both software and hardware processes, interacting with an FPGA design inherits the same semantics as normal UNIX software programs. In general, there is no need for a program to differentiate if a running process is a software program or an FPGA design. For example, using the standard UNIX pipe construct, a program can pipe its output to either a software program or a hardware design without being aware of it. The BORPH kernel hides any discrepancies between software and hardware during runtime. This homogeneous handling of hardware and software in the kernel forms the foundation of coarse grain hardware/software codesign boundary.

In this paper, we focus on three essential concepts of BORPH. First we will describe the concept of *hardware process*. Then two different ways to communicate with hardware processes are presented: the *ioreg* interface and the *hardware file I/O* interface. With respect to a hardware process, the IOREG interface provides a passive communication mechanism, while the hardware file I/O interface provides an active communication mechanism. Our particular implementation of these interfaces is described in Section 4.

2.1 Hardware Process

In conventional OS terminologies, a process is usually defined as "an executing instance of a program," running on a processor. BORPH extends this idea to reconfigurable hardware, defining a hardware process as "an executing instance of a hardware design."

A hardware process is created when a BORPH Object File (BOF) is `exec`-ed. As shown in Figure 2, a BOF file is a binary file format that encapsulates, among other information, configuration for FPGAs. In conventional UNIX systems, a process is created with two system calls: `fork` and `exec`. When a hardware process is created by a software process, the same `fork`-`exec` sequence is employed. During the `exec` system call, the actual hardware region is setup according to the configuration in the corresponding BOF file. Since hardware process creations are handled by the kernel, a hardware design can be started by any software program that is able to create normal UNIX process, such as the command shell, a C or Java program, etc. Figure 3 shows a simple transcript of executing a BOF file.

```
1:bash$ ./counter.bof &
[1] 1399
2:bash$ ps j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 1386  1387  1387  1387 pts/3     1400 Ss    1000   0:00 -bash
 1387  1399  1399  1387 pts/3     1400 S     1000   0:00 ./counter.bof
 1387  1400  1400  1387 pts/3     1400 R+    1000   0:00 ps j
3:bash$ cat /proc/1399/hw/ioreg/cntval
A3B498E0
4:bash$ cat /proc/1399/hw/ioreg/cntval
B289E906
5:bash$ echo 0 > /proc/1399/hw/ioreg/cnten
6:bash$ cat /proc/1399/hw/ioreg/cntval
C103D024
7:bash$ cat /proc/1399/hw/ioreg/cntval
C103D024
8:bash$ kill -9 1399
[1]+  Killed       counter.bof
9:bash$
```

Fig. 3. Executing a BOF file containing a free running counter. FPGA hardware is configured at prompt 1 and is automatically unconfigured at prompt 8 by the BORPH kernel.

Hardware process creations conform to the standard UNIX process creation semantics by maintaining all necessary parent–child and process group information. The created hardware process has its own memory space and execution domain. As a result, there is no shared memory between hardware and other processes in the system by default. Currently, memory attached to hardware processes must be exported by the IOREG interface. Each hardware process also has access to its execution environment, including its executing command line arguments.

As a normal running process, the status of a hardware process can be checked by standard command like `ps` as shown in prompt 2 in Figure 3. The output of the command `ps` shows the parent–child relationship between the starting `bash` shell and the hardware process `counter.bof`, as well as its process group information. Of interest is the `STAT` column in the output, which shows `counter.bof` is at an "interruptible sleep" state. In our current implementation, to avoid a hardware process being put on the processor's run queue, it is marked with a Linux process state of `TASK_INTERRUPTIBLE`. In the future, a new process state will be introduced to indicate to the rest of the system that a process is being run on a HWR.

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

2.2 The ioreg Interface

BORPH's IOREG interface encapsulates conventional memory mapped I/O concept with a virtual file system interface similar to that presented in [Donlin et al. 2004]. Communication between hardware and software typically involves defining a set of special hardware registers that are memory mapped by software

Table I. Hardware Constructs Supported by the IOREG Virtual File System

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

device drivers. BORPH encapsulates this common design practice by supporting it systematically via its IOREG interface. Note that this is a passive communication method with respect to the hardware process, because it is usually a software process that initiates the communication transaction.

BORPH extends the standard Linux `/proc` directory to include hardware-specific information about a hardware process. When a hardware process is started, the kernel populates a special `/proc/<pid>/hw` directory under that process ID (`<pid>`). In this directory are virtual files that provide information, such as the physical FPGA location, of this hardware process. There is also a subdirectory named `ioreg` that is used by the IOREG interface. Each virtual file in this directory corresponds to one hardware construct embedded in the user hardware design. Reading from, or writing to, these virtual 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 Figure 3 contains a free-running counter that stores its output in an IOREG register named `cntval`. The operation of this counter is controlled by an enable register called `cnten`. Once the design is running in the system as a hardware process, the value of `cntval` can be read by any program, such as `cat`, from the file `/proc/<pid>/hw/ioreg/cntval` (prompt 3 in Figure 3). Similarly, disabling this counter can be accomplished by writing "0" to the file `/proc/<pid>/hw/ioreg/cnten` by any program, such as `echo` (prompt 5).

Despite the name might have suggested, the IOREG interface supports not only simple single word register. It also provides access to on-chip FIFO, on-chip memory, as well as off-chip memory that a hardware process have access to. Table I shows the supported hardware construct by this interface and their differences when exported as virtual files in BORPH.

The virtual files under `/proc/<pid>/hw/ioreg` are created by the kernel according to the information embedded in the corresponding BOF file. Each virtual file's name, unique id, size, and access mode are embedded in the BOF file header. Based on these information, when a virtual file is read (or write), BORPH kernel sends a message to the running hardware indicating the corresponding access. The user hardware is then responsible for returning the necessary value to the kernel. The exact mechanism by which the user design obtains such value is implementation dependent; we will defer the discussion until Section 4.

No caching is performed on these virtual files. Each read (or write) access to these virtual files causes the kernel to generate the corresponding message to

the user design. The kernel's involvement again guarantees that this interface is language independent. For example, reading an on-chip memory may be accomplished by a simple shell `cp` command

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

or similarly in a C program:

```
memfile = fopen("/proc/123/hw/ioreg/Shared_Memory", "r");
fread(buf, 2048, 1, memfile);
```

Both of the above code causes the kernel to generate the same message to a user design, requesting 2048 bytes from the corresponding memory, starting at offset 0.

The format of message kernel sends to a user design is common to all IOREG virtual files regardless of the underlying hardware that they represent. The differences between, for example, a memory or an on-chip FIFO are handled on the Linux virtual file system level. For instance, since an on-chip FIFO should only logically be read from the head, its corresponding virtual file is marked as nonseekable in the Linux virtual file system layer, causing any file seek attempt to fail. As a result, all requests to a FIFO will always have an offset value of 0. On the other hand, an on-chip memory may be accessed randomly and, therefore, does not impose such restriction. Consequently, the kernel might initiate a request from a nonzero offset to the user design as needed.

2.3 File I/O

Hardware processes in BORPH have access to the general UNIX file system just like normal software processes. Unlike other coprocessor-based FPGA systems, such *active* communication mechanism that is initiated by user hardware designs is only logically possible with BORPH's hardware process concept. It enables a hardware-centric, or even hardware-only, design methodology. Despite the simple semantics, however, the fact that hardware processes are not running in the same processor as the OS implies extra care must be taken by the kernel. The BORPH kernel must take care of HW/SW discrepancies, such as process blocking, file caching, parallel file access, and error handling.

When a hardware process is started, three standard I/O files, `stdin`, `stdout`, and `stderr`, are automatically opened. They constitute the simplest form of hardware file I/O as no file `open`, or `close` system call is needed. They are currently being used for hardware debugging, as well as for interprocess data streaming.

For debugging purposes, a small shell program that utilizes `stdin` and `stdout` is implemented directly from an FPGA design, which allows low-level user interaction. Hardware designs read keyboard input and write information directly to the user's terminal. Access to `stdout` also enables printing messages to the screen in response to certain hardware event. Such "debug by printing" capability was previously only available during HDL simulation.

Another use of standard file I/O is to chain multiple processes through UNIX *pipes*. Because of the standardized file interface, a user can freely combine

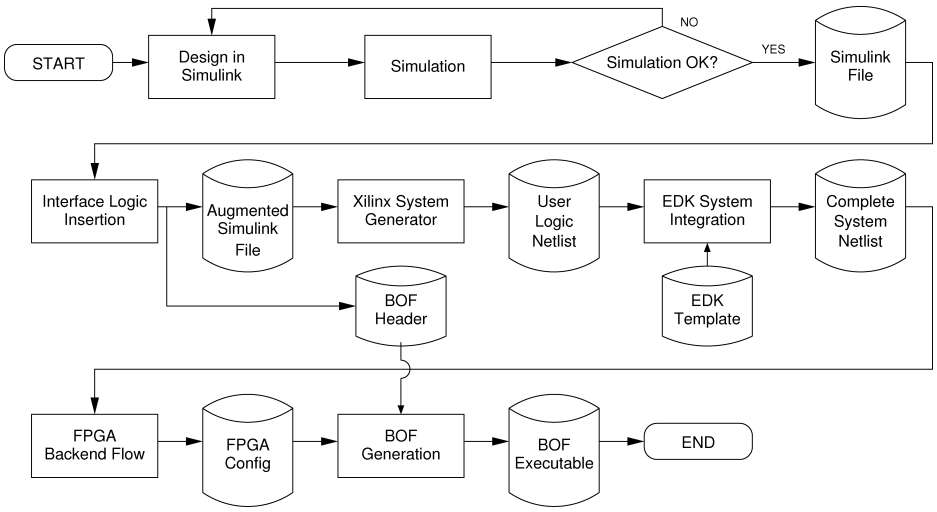


Fig. 4. An automatic hardware design flow that compiles high-level Simulink designs into executable BOF files.

software and hardware processes on either end of a pipe. For example, a video transcoding task can be accomplished by the following hypothetical command:

```
bash$ cat video.in | v_decode | v_filter | v_encode > video.out
```

where `v_decode`, `v_filter` and `v_encode` may be implemented in either software or hardware.

If the two processes on each side of the pipe are both hardware processes, the BORPH kernel will automatically setup a pure hardware route between the two processes, bypassing kernel's involvement in the middle. Such pure hardware pipe provides much higher bandwidth than a software-hardware pipe. Nonetheless, with an identical interface, user designs do not have to be aware of the hardware–software difference during compile time.

Finally, besides standard I/O, hardware processes can access any other file in the system during runtime.

3. HARDWARE DESIGN FLOW

One of the design goals for BORPH is for it to be language independent. However, to make concrete discussion on how one creates designs for use with BORPH and how various components of BORPH work together to form a complete design and runtime environment, we will describe the specific high-level hardware design flow we currently employ in this section.

Creating hardware designs for use with BORPH requires the design be able to communicate with the kernel via a specific protocol. However, it is usually desirable to isolate users from most of the complexities involved. For that purpose, we have extended our previously reported Simulink-based hardware design flow [Chang et al. 2003] for integration with BORPH. Figure 4 shows the major stages of our integrated hardware design flow.

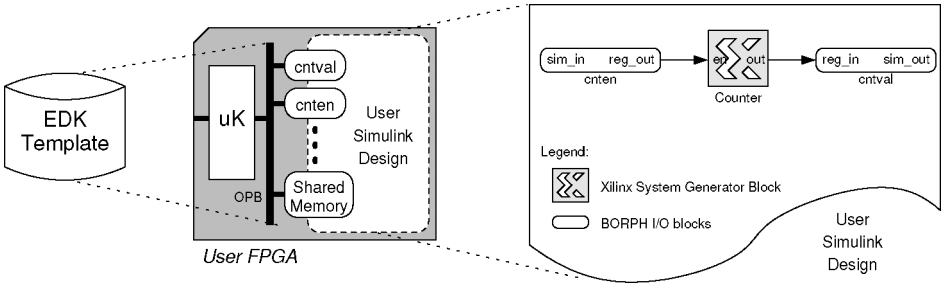


Fig. 5. Block diagram of a user FPGA. Compiled user Simulink designs are combined with a predefined EDK template of UK to generate user FPGA configurations.

Using this design flow, users describe their designs in Simulink using blocks provided by Xilinx System Generator [Xilinx]. The vendor-provided blockset includes blocks ranging from low-level single bit flip-flops, to complex hardware constructs such as adder, multiplier and finite impulse response (FIR) filter. To interact with the rest of the BORPH systems, users make use of data I/O blocks from our in-house library. It includes, for instance, custom library blocks for hardware constructs that are exported and accessible through the IOREG virtual file system such as register, shared memory, and FIFO. Once the design is created in Simulink, the user may optionally simulate the design within the Simulink environment before proceeding to hardware generation.

The hardware generation process is where the BORPH specific steps are involved. First, the user Simulink design is parsed to identify all instances of BORPH-specific library blocks. These blocks serve as the boundary between the actual generated hardware and native Simulink blocks that are only for simulation purposes. Xilinx System Generator is then called to generate the necessary netlist from the user design, instantiating native library blocks accordingly. Clock and reset insertion, as well as data sample rate resolutions, are handled by System Generator. The resulting low-level netlist is then prepared as a block for use with Xilinx Embedded Development Kit (EDK) in our next step.

Next, a processor system (UK) is inserted, as shown in Figure 5. All IOREG-related blocks from a user design are connected to a multilevel on-chip peripheral bus (OPB) that is accessible from this processor system. All runtime communications with the central BORPH software kernel (MK) are handled by this processor system. A detail block diagram of UK is shown in Figure 8 (see later). The combined system is subsequently passed to vendor-provided backend tools for synthesis, map, place and route.

Finally, from the top-level Simulink design, a symbol file is generated that lists information, such as address and size of all BORPH-specific blocks. This symbol file is combined with the FPGA configuration file generated by the vendor tools to create the final BOF executable file.

This fully automated tool flow takes the role of a compiler in conventional software development methodologies. It hides all detail interactions with the kernel from a user. Such abstraction simplifies users' experiences with hardware development, allowing them to focus their efforts on application logic

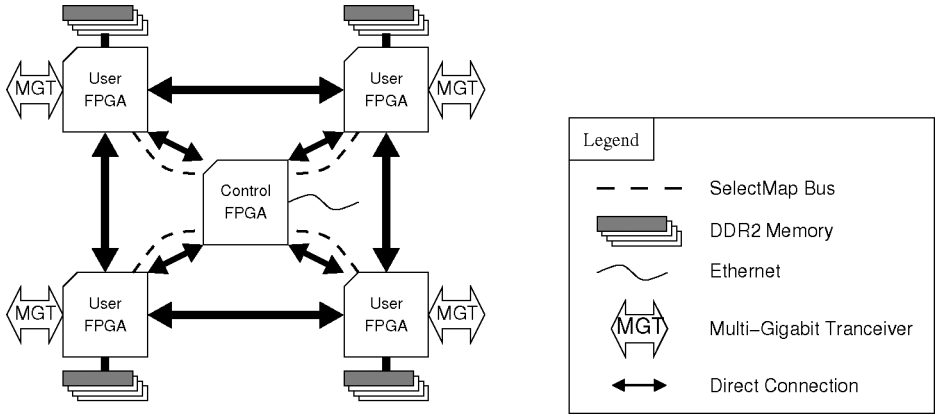


Fig. 6. A BEE2 compute platform.

design. At the same time, such abstraction allows upgrade to the OS kernel implementation without requiring user intervention. We will show one such example in Section 4.

4. CURRENT IMPLEMENTATION

This section describes our current implementation of BORPH. We will first describe the underlying BEE2 hardware platform [Chang et al. 2005]. Then we will describe the software architecture of the BORPH kernel. Finally, we will describe two different versions of on-chip hardware architecture implementations.

4.1 BEE2 Platform

Figure 6 shows the block diagram of a BEE2 compute module. Each BEE2 compute module contains 5 Xilinx Virtex-II pro xc2vp70 FPGAs. Each FPGA contains two on-chip PowerPC 405 cores. The center *control FPGA*, handles all system related functions, such as networking and FPGA configurations. The remaining four *user FPGA*s are used for implementing user designs.[2] Each of the five FPGAs is connected to four DDR2 memory banks, supporting up to 8 GB of external memory. Furthermore, each user FPGA is connected to four independent external high-speed serial I/O channels, which provides a raw bandwidth of 12.5 Gb/s. The control FPGA has two such connections. Each FPGA runs at 100 MHz, while the processors run at 300 MHz.

The four user FPGAs are connected in a ring topology with a 120-bit direct connection to its neighbor. Each user FPGA is also directly connected to the center-control FPGA with a 50-bit connection. User FPGAs are configured by the control FPGA using an 8-bit SelectMap bus. Once the user FPGA is configured, this bus is doubled as a communication channel between the control and user FPGA. BORPH currently communicates with all user FPGAs using this connection.

[2]Note that with some special arrangement, it is possible to use part of control FPGA for user applications.

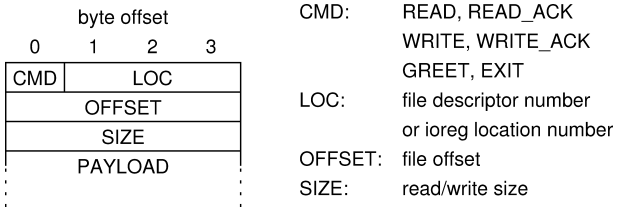


Fig. 7. A simple packet format for message exchange with BORPH kernel.

4.2 Software Kernel Architecture

BORPH kernel is a modified version of a Linux 2.4.30 kernel running on the left PowerPC 405 core in the control FPGA. A standard Debian PowerPC root file system is mounted over network file system (NFS), which provides familiar Linux applications to the user. To provide kernel support for FPGA fabric, a number of modifications have been made to the standard Linux kernel.

4.2.1 *BOF File Support.* Standard Linux kernel contains an extensible interface to support user-defined binary file format. Making use of this interface, we have developed a binary file format kernel module called `binfmt_bof` to handle execution of BOF files.

4.2.2 *Hardware Region (*HWR*).* We have defined a set of kernel APIs 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 `hwr_b2fpga`, which corresponds to a BEE2 user FPGA.

4.2.3 *Hardware Configuration and Resource Allocation.* A kernel thread `bkexecd` is created to handle all HWR allocation and configurations. When a BOF file is executed by the user, as long as the BOF file is relocatable, as in the case for `hwr_b2fpga`, `bkexecd` will pick an unused FPGA and configure that FPGA accordingly. A user may override this behavior by setting a `hwr_addr` field in the BOF file header. Doing so instructs `bkexecd` to configure a BOF file only at the specified user FPGA. This is useful for cases when a user design requires specific FPGA that is connected to special external hardware. In case the requested FPGA is not available, or no free FPGA is available in the case of relocatable BOF file, a device busy error is returned to the user with standard Linux semantics.

4.2.4 *Software/Hardware Communication.* All communications between software and hardware are handled by the BORPH kernel through a standardized message-passing network. Figure 7 shows the format of such message packet. These message packets are used for all kernel communications, including the hardware file I/O and the IOREG interface.

A kernel thread (mkd) is responsible for handling all read/write messages from hardware processes. For each file opened by a hardware process, a kernel thread, called a *fringe*, is created to perform the actual file operations. Similarly, user read/write requests to virtual IOREG files are translated automatically into packet messages that communicate with hardware processes.

4.2.5 *Process Scheduler and Signal Handler.* Since hardware processes do not run on the processor that BORPH runs on, the kernel must take special care during process scheduling. Hardware processes are handled differently such that they are never put on the software run queue.

Furthermore, since hardware processes may access terminal TTYs, they must respond correctly to "stopping" and "continuing" signals. For example, a hardware process will receive SIGSTOP when a user press CTRL-Z on the running terminal, or SIGTSTP when it tries to read from a terminal while it is running in the background. Also, a hardware process receives SIGCONT when a user put the process in background, or back to foreground. The BORPH kernel is modified to handle these cases for hardware processes so that they conform to standard UNIX semantics and thus be able to coexist coherently with other software processes in the system.

4.2.6 *Software Fringe.* A hardware process maybe blocked while accessing the general file system if the file is not ready. To handle this potential blocking, each opened file by a hardware process is managed by a software *fringe* running on the processor. The fringe performs blocking file I/O operations on behalf of the hardware process and sends data back to the hardware process only when it is ready.

4.3 On-Chip Hardware Architecture

On-chip hardware architecture refers to the hardware infrastructure that must be implemented in order to support BORPH's operations. Figure 8 shows a block diagram of the internal of the control FPGA and a typical configuration of a user FPGA created by our Simulink design flow.

The part of a BEE2 compute module that makes up BORPH's infrastructure is labeled "kernel space" in Figure 8. It includes the control FPGA (MK) and a part of the user FPGA (UK) that is responsible for communicating with the control FPGA. The part of the system that is responsible for executing user hardware designs is labeled "user space" in the diagram. This part of the user FPGA is denoted a reconfigurable hardware region (HWR). Each UK can be thought as a distributed, low-level manager for the attached HWR.

On the control FPGA, a standard processor system is built around the left PowerPC 405 core on which BORPH runs. To communicate with user FPGAs, a SelectMap bus controller is attached to the processor's on-chip peripheral bus (OPB) via the processor local bus (PLB). As hardware processes are created and destroyed, FPGA is configured and unconfigured. Since all four user FPGAs have the exact same pin connections, BOF files targeting BEE2 user FPGAs are relocatable.

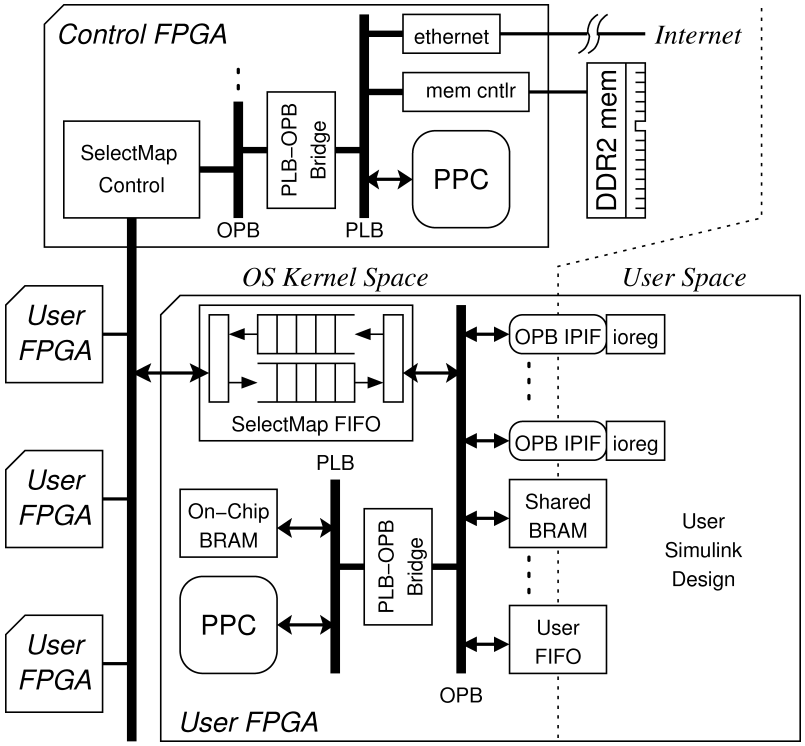


Fig. 8. Block diagram of BORPH system on a BEE2 compute module with OPB-based SelectMap controller in control FPGA (OPBSM).

The part of user FPGA that belongs to the system infrastructure is called a micro kernel (uK). It handles distributed hardware process management on behalf of the main kernel. For example, it controls the starting and stopping of hardware process on that user FPGA once it is configured. It responses to IOREG packets sent from the BORPH kernel by reading or writing the corresponding values to a user design. It also keeps records of opened files by a hardware process and handles read/write requests by a hardware process accordingly.

As mentioned before, the processor system on the user FPGA is automatically inserted by our design flow. As part of the system infrastructure, this part of hardware should never be unconfigured when a user hardware process terminates. However, in our current design, we have chosen to embed this part of the kernel within a user BOF file for simplicity sake.

We have also reimplemented part of the control FPGA as shown in Figure 9. In this version, the SelectMap bus controller is connected to the processor local bus directly. Communication performance between control FPGA and user FPGA is improved by eliminating delay through the PLB-to-OPB bridge. This second version serves as a demonstration of BORPH's hardware kernel/user separation concept, while improving HW/SW communication performance at the same time. We denote the original and the new architecture OPBSM and PLBSM, respectively.

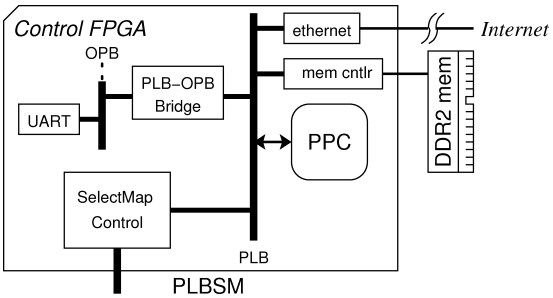


Fig. 9. PLB-based SelectMap controller implemented on control FPGA of BEE2 (PLBSM).

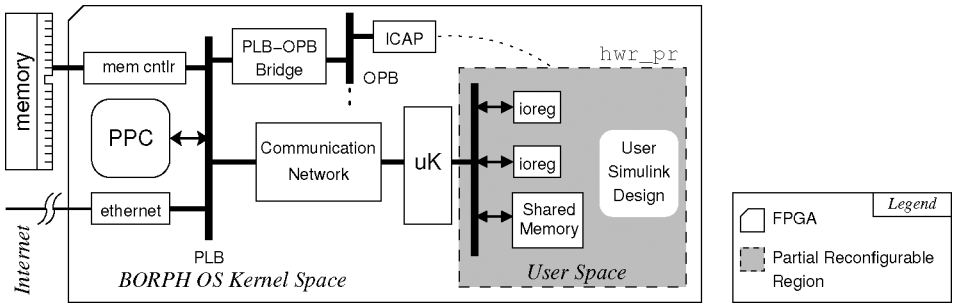


Fig. 10. An example of porting BORPH to a single FPGA, where user hardware processes are executed on a partially reconfigurable region.

4.4 Porting BORPH

Although currently implemented on a BEE2 compute module, BORPH is a general operating system design that can be applied to other FPGA-based systems. This section provides a high-level description on porting BORPH.

To make the discussion concrete, we will illustrate the proess using an example of porting BORPH to an FPGA that supports dynamic partial reconfiguration. The top-level block diagram of the resulting system is shown in Figure 10.

Porting BORPH to the platform in Figure 10 requires changes to both hardware and software from the original BEE2 implementation. Logically, all hardware and software changes involved can be classified as part of the process of implementing a new reconfigurable hardware region (HWR). Denote this new HWR as `hwr_pr`. As shown in Figure 10, we assume only one instance of `hwr_pr` will be presented.

The first hardware change required is to implement logic that configures and unconfigures the newly defined partial reconfiguration region. Modern FPGAs provide native support for this purpose, such as through the internal configuration access port (ICAP) in Xilinx FPGAs. For other platforms, different configuration logic will need to be implemented.

The second hardware change required is to reimplement the message-passing network between MK and UK. The original BORPH implementation on BEE2 utilized the SelectMap bus for communication between control and

user FPGA. Porting BORPH to a new platform requires a new communication mechanism between MK and UK. In the case of hwr_pr, since MK and UK are physically residing within the same FPGA, this communication network may be reduced to a simple memory mapped device on the processor system.

Most of the BORPH software kernel can be ported to any FPGA-based reconfigurable system without modification because they are written as high-level machine independent code. Two parts of the BORPH kernel are machine dependent. First, some part of signal-handling code is processor dependent. Therefore, porting BORPH to a FPGA-based reconfigurable system with a different processor will require small modifications to its signal-handling code. Second, the BORPH software kernel must be updated to handle any newly defined HWR type by registering a new corresponding HWR module.

As described in Section 4.2, BORPH's kernel design has a dedicated subsystem devoted to supporting different kinds of HWR types. This HWR subsystem works in ways similar to the standard Linux device driver subsystem. A set of virtual function calls must be implemented by each HWR type kernel module to handle hardware specific operations. For example, in the case of hwr_pr, the configure function must be implemented to configure the correct reconfigurable region on the FPGA using the built-in ICAP. It must also make use of the newly implemented hardware for communication between MK and UK. Work is currently underway to standardize this interface for MK–UK communication.

Supporting any additional system calls for hardware process can be accomplished by defining a new message that corresponds to each supported system call. For example, to add support of gettimeofday function to FPGA designs, a new message with unique CMD field corresponding to gettimeofday can be defined. Upon receiving such message, the main message handling thread, mkd, may then serve as a proxy that returns the current time of the day to the issuing hardware process.

5. PERFORMANCE

In this section, we will describe the implementation of various tasks that are performed by the BORPH kernel. Performance of such tasks, based on the original OPBSM is first presented. Then, we will discuss the migration process from OPBSM to PLBSM, as well as the performance differences between the two.

5.1 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 FPGAs are chosen and configured accordingly using the SelectMap bus.

Using the OPBSM on-chip architecture, creating a hardware process takes about 900 ms, while creating a normal software process takes about 13 ms on the same processor.[3] The theoretical minimum time required to start a hardware process is the sum of the time to start a software process plus the time to

[3]This is time needed when the program is already residing in memory cache.

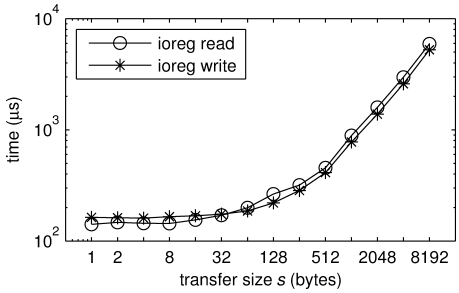


Fig. 11. Performance of reading/writing on-chip memory on a user FPGA using IOREG interface.

configure an FPGA, which amounts to 65 ms in our system. The process creation time is limited by data transfer speed from control FPGA to the user FPGA. Preliminary investigation indicates that the PLB-to-OPB bridge on the control FPGA as well as the SelectMap bus implementation, are limiting the data bandwidth. Our second implementation, PLBSM, confirms this assertion.

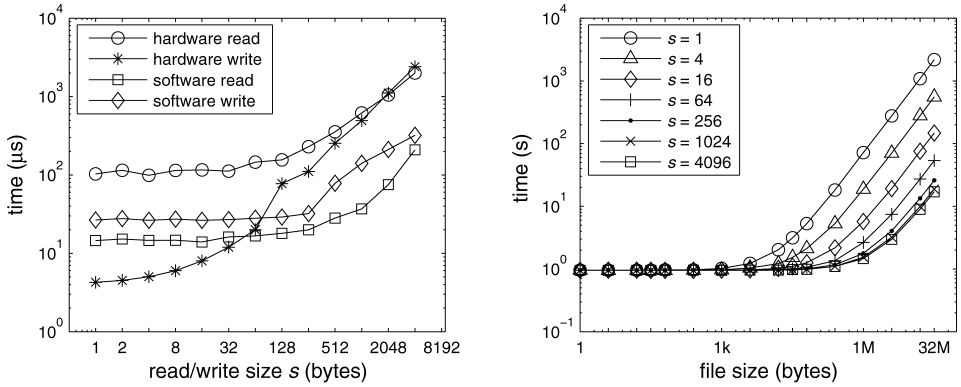
Note that modern operating systems like Linux employs demand paging techniques such as copy-on-write that significantly reduce software process start up time. Although demand paging of hardware configurations have been studied by other researchers in specialized partially reconfigured systems, such technique is not being used by BORPH. BORPH focuses on the hardware process abstraction and its integration with the rest of the software system.

5.2 Reading/Writing ioreg Virtual 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.

Figure 11 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 (16 KB), 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 MB/s[4] for both read and write.

[4]1 MB/s $= 2^{20}$ bytes/s.



(a) Measured from hardware process (b) Measured from software process

Fig. 12. Hardware process file I/O performance.

5.3 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 Figure 7, with the LOC field denoting Linux-opened file descriptor number. All messages are received by `mkd`. Each opened file is managed by a *fringe*, which is implemented as a kernel thread on the control FPGA. A fringe is woken up by `mkd` as needed.

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.

5.3.1 *Regular File I/O.* First, to determine the performance of regular file I/O from a user FPGA hardware process, the two programs are run as follows:

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

Figure 12a shows the file I/O performance of both processes with various file transfer sizes s. In the case of `stdloop.bof`, the time for each file I/O operation is measured directly on the user FPGA.

A hardware process file read is analogous to the C function call:

```
read(fd, buffer, s);
```

For each read, a read request packet is sent from the user FPGA to the control FPGA. As a result, `mkd` is woken up, which, in turn, wakes up the corresponding fringe. The fringe then carries out the file read on behalf of the hardware process, blocking as needed. Once the request data is ready, the fringe sends all data back to the user FPGA in one READ_ACK packet. Comparing a hardware file read to a software file read, a hardware file read incurs the overhead of interrupt handling and two context switches to the fringe. This overhead is reflected when value of s is small. Moreover, there is the overhead of data movement for large values of s. On the other hand, software reads require almost the same

amount of time as fringe reads. The only difference between the two is the extra kernel boundary crossing time for software processes as they run in user mode.

Similarly, a hardware process file write is analogous to the C function call:

```
write(fd, buffer, s);
```

For each write, the data to be written is sent as the payload of a write request packet to the control FPGA. Upon receiving the packet, `mkd` is woken up. Since most file writes complete successfully without blocking, as an optimization, `mkd` writes to the file on behalf of the hardware process directly without involving a fringe, eliminating one context switch for each file write. Furthermore, we have eliminated the need for hardware processes to wait for WRITE_ACK messages. Therefore, for small writes that can fit into the on-chip FIFO, the time to write a file is the same as the time needed to write the packet into the FIFO. This explains the very fast hardware file write operations for small payloads. For write operations with larger payloads, however, the hardware process is delayed further by both the limited size of on-chip FIFO and the maximum packet size limit imposed by BORPH. Consequently, large hardware file writes approach the performance of hardware file reads.

In general, file readings by hardware processes requires more than a single hardware `read` call. Figure 12b plots the time required for a hardware process to read files of various sizes using different values s for each read request. The values are measured from the control FPGA using the `time` command:

```
bash$ time sink.bof -s $SIZE < $INPUT_FILE
```

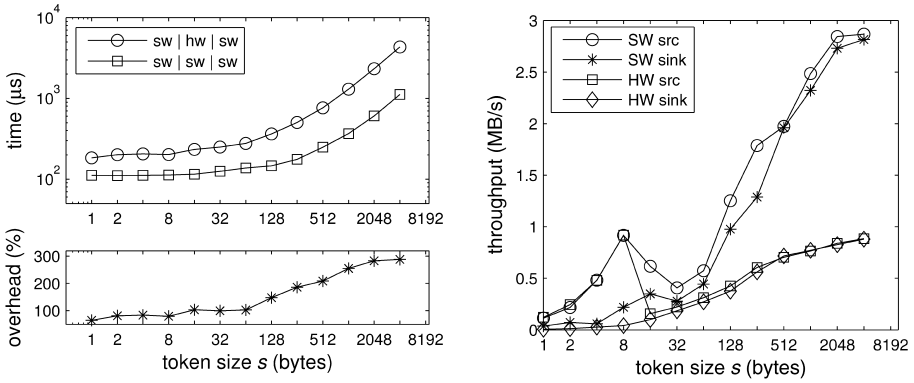
where `sink.bof` is a hardware design that reads the entire content of a file from `stdin` and then exit. This graph gives an overall system performance benchmark as it includes system time, such as the time for process creation and destruction.

From Figure 12b, it can be seen that large transfer size s is essential for large files transfer to amortize the high overhead involved for each hardware file system call. For $s = 4096$, streaming a 32 MB file to a hardware process completes in 16.68 s, resulting in a transfer speed of 1.91 MB/s. On the other hand, for $s = 1$, the same transfer takes 1871.73 s.

5.3.2 *Piped Process through Standard I/O.* Having access to system standard I/O allows hardware and software processes to communicate with standard UNIX pipes. To evaluate the performance of such pipe, a second benchmark is performed. Instead of reading regular files, the two programs from previous section are run as follows:

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

where `sendtok` and `recvtok` are software 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, is shown in the top half of Figure 13a. Communication overhead, which is the extra time needed for the mixed HW/SW pipe over the software pipe, is plotted at the



(a) Latency between when `sendtok` sends a token and when `recvtok` receives the entire token

(b) Throughput of both hw–sw and sw–sw pipes

Fig. 13. Performance comparisons between software pipeline and mixed hardware/software pipeline.

bottom half of the diagram. For $s < 128$, the mixed HW/SW pipe's latency is about 60% more than the pure software pipe. The gap increases as s increases to almost 300% for $s \geq 128$ as the data transfer latency starts to dominate.

In the above piped organization, throughput of the pipe is also an important metric. Figure 13b shows the throughput of HW/SW pipe and SW/SW pipe for different token sizes s. Throughput is measured by `sendtok` at the source, and `recvtok` at the sink. The difference in rate is a result of buffering within the system. At the receiving end, throughput increases as s increases. For $s = 4096$, the mixed HW/SW pipe has a throughput of 858.56 kB/s,[5] while the SW/SW pipe has a throughput of 2.8 MB/s.

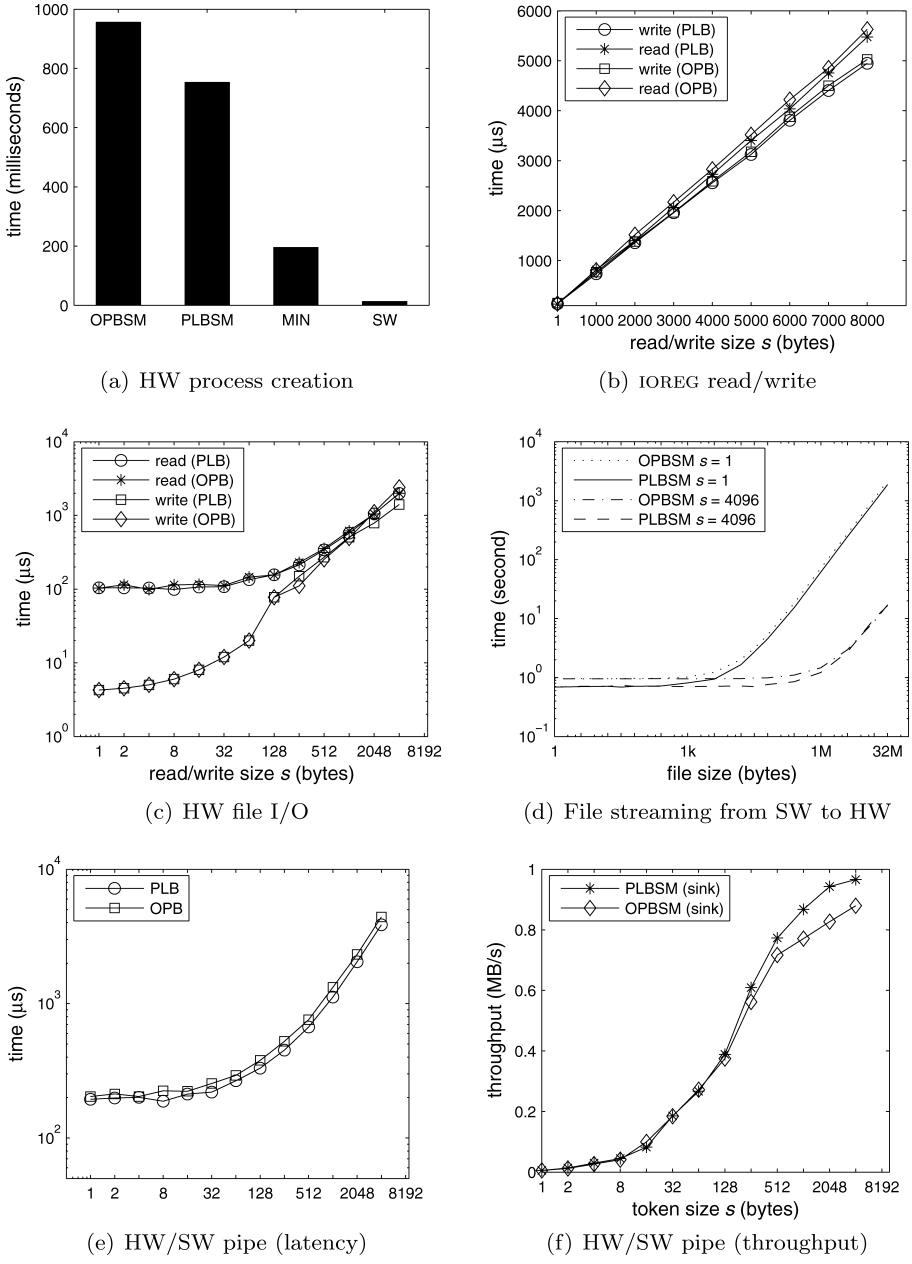
5.4 Alternative Hardware Architecture

The performance results reported thus far are all performed on the original on-chip hardware architecture, OPBSM. Figure 14 shows the results of repeating experiments from previous section in PLBSM. Each subfigure plots the results for both OPBSM and PLBSM. To highlight the performance difference between the two hardware architectures, only subsets of all data points are shown in the diagrams.

Figure 14a shows that process creation benefits most from the new architecture, with a 28.9% decrease in process creation time. To create a hardware process, no complex interrupt handling or message exchange is needed between control FPGA and user FPGA. Hardware process creation is dominated purely by the transfer of 3 MB of configuration data to the user FPGA. Therefore, all the performance gain in bypassing a PLB-to-OPB bridge is reflected in the process creation benchmark.

On the other hand, Figure 14c shows that there is only marginal improvement on hardware file I/O as measured from the user FPGA. It is as expected

[5]1 kB/s = 1024 bytes/s.



(a) HW process creation

(b) IOREG read/write

(c) HW file I/O

(d) File streaming from SW to HW

(e) HW/SW pipe (latency)

(f) HW/SW pipe (throughput)

Fig. 14. Performance comparison between OPBSM and PLBSM.

because the performance of hardware file I/O is largely limited by the speed of the processor system on the user FPGA and not the control FPGA. On a user FPGA, the processor communicates with the SelectMap bus FIFO through an OPB-connected controller. Here, the OPB-to-PLB bridge is again the bottleneck, undermining any improvement in control FPGA speed. Figure 14d shows that

for file reads with large enough size, PLBSM and OPBSM have similar performance. For small file transfers, the performance advantages of PLBSM are direct results of faster process-creation times.

Performance of IOREG interface also sees limited performance improvement: an average of 4% increase for large reads and 2% increase for large writes. The performance of the IOREG interface is largely limited by pure hardware data transfer speed. As in the case of hardware file I/O, since the user FPGA is limiting the data transfer speed, very little improvement is expected.

The chained hardware/software pipe benefits significantly from the architectural change. An average of 12% performance increase in latency (Figure 14e) is observed. With a transfer size $s = 2048$, the overall throughput increases 14% from 807 to 920 kB/s (Figure 14f). The piped process operation involves complex HW/SW interaction, such as context switching on the processor, interrupt, and message-passing, etc. Therefore, a faster data transfer speed on the control FPGA allows the processor to perform these tasks more efficiently, resulting in higher overall system performance.

5.4.1 *Comparing the Two Hardware Architectures.* The performance benchmarks of PLBSM have identified two performance bottlenecks in our current BORPH system. First, the elimination of PLB-to-OPB bridge improves raw data transfer performance, as expected. However, the increased performance in control FPGA must be matched by the user FPGA. Second, in a heavily loaded system, the processor currently spends a large portion of the processing time in data transfer, hindering overall system performance.

Based on the above two observations, we are currently implementing another hardware architecture in which direct memory access (DMA) will be employed between control processor and the SelectMap controller. This will increase raw data transfer speed and relieve the processor from intensive data transfer operations at the same time.

The implementation of PLBSM also demonstrates BORPH's concept of hardware kernel/user separation. During the migration from OPBSM to PLBSM, no recompilation was needed for hardware user designs used in benchmarking. The same designs have been used in both cases because the interface into the BORPH kernel, the SelectMap connection in this case, remains compatible. In general, it is the design philosophy of BORPH that the kernel/user interface must be kept consistent such that any change to the kernel does not affect user designs.

However, in order to improve overall system performance, we have to reimplementing uK on user FPGAs as well. Unlike changes that we have made to the control FPGA, it will require recompilation of user designs. Such recompilation violates the design philosophy of BORPH. In order to avoid future user design incompatibility issues, we are, therefore, developing new architecture for the user FPGA that employs dynamic partial reconfiguration with standardized interface into the BORPH kernel.

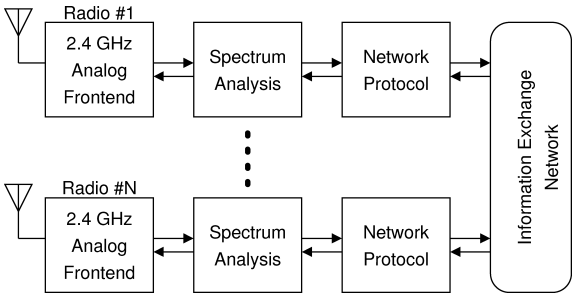


Fig. 15. Cognitive radio testbed system.

6. SAMPLE APPLICATIONS

In this section, three FPGA applications are briefly described to illustrate how various features of BORPH work together to ease their development processes.

6.1 Example 1: A Real-Time Wireless Signal Processing System

The first application is a real-time wireless signal processing system for our cognitive radio project [Mishra et al. 2005]. This system makes extensive use of the IOREG virtual file system for hardware parametrization and HW/SW communication. Furthermore, full backward compatibilities with existing Linux system allows the software team to perform HW/SW system testing remotely over the Internet.

6.1.1 *Overview.* 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 noninterfering use of the spectrum. Our system is designed to validate those techniques. Figure 15 depicts our overall system design that involves multiple cooperative cognitive radios.

Each radio is logically separated into two parts that are developed by two physically separated design teams. The partitioning is done loosely, based on the standard ISO network stack, where the physical layer is implemented in hardware and the higher layers are implemented in software.

At the physical layer, real-time spectrum analysis is performed by FPGA hardware that is connected to external RF frontend. The FPGA hardware is designed using our Simulink-based design flow described in Section 3. All high-level network protocols are independently developed in software.

6.1.2 *Using BORPH for 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 memories are exported as IOREG virtual files for data communication with software. In addition, more than 20 single word registers are defined. Most of them serve the function of controlling hardware parameters, such as RF channel and amplifier gain. Some IOREG registers, however, are used solely for synchronization

purpose. For example, each shared memory is guarded by a pair of `enable` and `ready` registers. 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 and to display hardware status.

6.1.3 *Remote System Testing.* Our software design is developed off-site. BORPH provides a remote testing environment for our protocol group who does not 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 onto the physical hardware for mixed HW/SW testing with a simple swap of hardware process in place of the emulating software process. Since BORPH runs with a fully functional Debian root file system, all necessary software development tools, such as `gdb` are available for debugging.

6.2 Example 2: Low-Density Parity-Check Decoders Emulation

In this project, FPGAs are used to study quantization effects on the performance of low-density parity-check (LDPC) codes [Zhang et al. 2006, 2007]. Because of the intense computational need for empirical study of LDPC code for even moderate bit error rates (BER), hardware emulations using FPGAs are employed.

Similar to the previous wireless signal processing example, this application relies heavily on BORPH's IOREG virtual file interface to dynamically customize design parameters to explore implementation choices. Furthermore, IOREG virtual files are used to export collected data for postprocessing.

This application is unique in that it does not require externally attached hardware devices for execution. As a result, during design exploration phase, the same relocatable BOF file can be executed concurrently on multiple FPGAs. The fact that each BORPH system is networked allows each instance to be conveniently started and parametrized through IOREG virtual files remotely. Having more than ten instances of the same design emulating concurrently have significantly improved the productivity of the designers.

6.3 Example 3: FPGA Video Processing with Commodity Software

The standard conforming file I/O capabilities for hardware processes allow FPGA designs to communicate with commodity software via standard pipes. To illustrate this, we have implemented a simple Sobel edge detection program in FPGA (`yuvedgdet.bof`) that works with the MJPEG tools [mjpegtools]. The MJPEG tools is a set of Linux programs that collectively perform complex video editing functions. Most of the programs in the tool set communicate with each other through piped standard input and output, using a predefined raw video format (YUV4MPEG2).

Using BORPH's file I/O capabilities, the FPGA edge detection filter may therefore be inserted easily with a single shell command:

```
bash$ lav2yuv test.avi | yuvedgdet.bof | mpeg2enc -o output.mpg
```

where `lav2yuv` and `mpeg2enc` are programs distributed with the MJPEG tools. In the above command, `lav2yuv` translates the source video `test.avi` into the YUV4MPEG2 raw video stream, which is then redirected to our FPGA edge detection filter. The filtered video is piped back to the software MPEG encoder `mpeg2enc` for the final encoding.

Comparing to a pure software implementation, performance of the above hardware-in-the-loop video processing is 18% slower as a result of I/O overhead. As with any other hardware acceleration scheme, the advantage of FPGA implementation is expected to be more prominent as more computation, such as MPEG encoding, is shifted to FPGA, amortizing performance degradations because of I/O.

Most importantly, this example demonstrates BORPH's unique approach to hardware/software coexecution. The use of standard file semantics for hardware/software communication is not only easy to understand for novel users, but also allows easy integration with existing commodity software, greatly improving productivity of designers.

7. CONCLUSION

In this paper, we have described the design and implementation of BORPH, an operating system designed for FPGA-based 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 a 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. BORPH's kernel interface ensures a design language independent environment.

The concept of hardware process in BORPH allows user hardware designs to integrate into traditional UNIX environment as a normal running process. It allows hardware designs to take an active role in the system. Instead of the traditional master-slave relationship with software, a hardware process forms a peer-to-peer relationship with software programs. Two ways of communicating with hardware processes have been presented. With respect to a hardware process, the IOREG interface provides a passive communication channel. The interface exports constructs from a hardware design to the rest of the system through a virtual file system. On the other hand, hardware processes initiate active communications to the rest of the system through standard UNIX file system.

We have also described our Simulink-based high-level hardware design flow as an example of how a hardware design environment integrates with BORPH. This design flow serves the same role as a compiler in software design methodology. Starting from a high-level design, a BORPH object file (BOF file) is

generated with a single-pass compilation process. The created BOF file can then be executed in the BORPH system as native binary.

Two variations of BORPH hardware implementation on a BEE2 platform were presented. By improving raw data transfer speed, user hardware process creation time is reduced by 28.9%. Latency and throughput of mixed HW/SW pipes is improved by 12 and 14%, respectively. Hardware file I/O and IOREG interface have limited performance improvement, because they are limited by the speed of user FPGA.

The two hardware implementations serve as a proof-of-concept of BORPH's kernel/user separation design philosophy. Since implementation detail of BORPH is independent of the OS interface, performance is expected to be improved through future reimplementations.

Currently, we are further exploring the semantics for hardware processes, such as blocking, parallel file system access, and HW/SW 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.

ACKNOWLEDGMENTS

The authors would like to thank Pierre Droz and Andrew Schultz for developing many fundamental BEE2 building blocks, and to thank Artem Tkachenko and Zhengya Zhang for testing various features of BORPH.

REFERENCES

BALARIN, F., GIUSTO, P. D., JURECSKA, A., PASSERONE, C., SENTOVICH, E., TABBARA, B., CHIODO, M., HSIEH, H., LAVAGNO, L., SANGIOVANNI-VINCENTELLI, A., AND SUZUKI, K. 1997. *Hardware-Software Co-design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publ., Norwell, MA.

CELOXICA. [Online]. Available: `http://www.celoxica.com`.

CHANG, C., KUUSILINNA, K., RICHARDS, B., CHEN, A., CHAN, N., BRODERSEN, R. W., AND NIKOLIĆ, B. 2003. Rapid design and analysis of communication systems using the bee hardware emulation environment. In *RSP '03: Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*. IEEE Computer Society, Los Alamitos, CA. 148.

CHANG, C., WAWRZYNEK, J., AND BRODERSEN, R. W. 2005. BEE2: A high-end reconfigurable computing system. *IEEE Design & Test 22,* 2, 114–125.

CRAY. XD1 supercomputer. [Online]. Available: `http://www.cray.com/products/xd1/`.

DANNE, K., MUEHLENBERND, R., AND PLATZNER, M. 2006. Executing hardware tasks on dynamically reconfigurable devices under real-time conditions. In *16th International Conference on Field Programmable Logic and Applications (FPL'06)*. 541–546.

DONLIN, A., LYSAGHT, P., BLODGET, B., AND TROEGER, G. 2004. A virtual file system for dynamically reconfigurable FPGAs. In *Proceedings Field Programmable Logic and Application, 14th International Conference, FPL 2004*, Leuven, Belgium, August 30–September 1, 2004. 1127–1129.

DRC COMPUTER. Development system 2000. [Online]. Available: `http://www.drccomputer.com`.

DYDEL, S. AND BALA, P. 2004. Large scale protein sequence alignment using FPGA reprogrammable logic devices. In *Proceedings Field Programmable Logic and Application, 14th International Conference, FPL 2004*, Leuven, Belgium, August 30–September 1, 2004. 23–32.

HAMADA, T., FUKUSHIGE, T., KAWAI, A., AND MAKINO, J. 1998. 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. 256–257.

LIN, E. C., YU, K., RUTENBAR, R. A., AND CHEN, T. 2007. A 1000-word vocabulary, speaker-independent, continuous live-mode speech recognizer implemented in a single FPGA. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. ACM Press, New York. 60–68.

MEI, B., VERNALDE, S., VERKEST, D., AND LAUWEREINS, R. 2004. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE Computer Society, Los Alamitos, CA.

MISHRA, S. M., CABRIC, D., CHANG, C., WILLKOMM, D., VAN SCHEWICK, B., WOLISZ, A., AND BRODERSEN, R. W. 2005. A real time cognitive radio testbed for physical and link layer experiments. In *1st IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN 2005)*. 562–567.

MJPEGTOOLS. [Online]. Available: `http://mjpeg.sourceforge.net`.

ORTIGOSA, E. M., ORTIGOSA, P. M., CAÑAS, A., ROS, E., AGÍS, R., AND ORTEGA, J. 2003. FPGA implementation of multi-layer perceptrons for speech recognition. In *Proceedings Field Programmable Logic and Application, 13th International Conference, FPL 2003*, Lisbon, Portugal. 1048–1052.

PANDA, P. R. 2001. SystemC: A modeling platform supporting multiple design abstractions. In *ISSS '01: Proceedings of the 14th International Symposium on Systems Synthesis*. ACM Press, New York. 75–80.

ROWSON, J. A. AND SANGIOVANNI-VINCENTELLI, A. 1997. Interface-based design. In *DAC '97: Proceedings of the 34th Annual Conference on Design Automation*. ACM Press, New York. 178–183.

SUGAWARA, Y., INABA, M., AND HIRAKI, K. 2004. Over 10gbps string matching mechanism for multi-stream packet scanning systems. In *Proceedings Field Programmable Logic and Application, 14th International Conference, FPL 2004*, Leuven, Belgium, August 30–September 1, 2004. 484–493.

VAN DER WOLF, P., DE KOCK, E., HENRIKSSON, T., KRUIJTZER, W., AND ESSINK, G. 2004. 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*. ACM Press, New York. 206–217.

WALDER, H. AND PLATZNER, M. 2004. A runtime environment for reconfigurable hardware operating systems. In *Proceedings Field Programmable Logic and Application, 14th International Conference, FPL 2004*, Leuven, Belgium, August 30–September 1, 2004. 831–835.

WIANGTONG, T., CHEUNG, P. Y. K., AND LUK, W. 2003. A unified codesign run-time environment for the UltraSONIC reconfigurable computer. In *Proceedings of Field Programmable Logic and Application, 13th International Conference, FPL 2003*, Lisbon, Portugal, September 1–3, 2003. 396–405.

WIGLEY, G. B., KEARNEY, D. A., AND WARREN, D. 2002. Introducing ReConfigME: An operating system for reconfigurable computing. In *Proceedings of the 12th International Conference on Field Programmable Logic and Application (FPL'02)*. Springer, New York.

XILINX. Xilinx system generator. [Online]. Available: `http://xilinx.com`.

XTREMEDATA. XD1000 development system. [Online]. Available: `http://www.xtremedatainc.com`.

ZHANG, Z., DOLECEK, L., NIKOLIĆ, B., ANANTHARAM, V., AND WAINWRIGHT, M. J. 2006. Investigation of error floors of structured low-density parity-check codes by hardware emulation. In *Proceedings of IEEE Global Communications Conference (GLOBECOM)*.

ZHANG, Z., DOLECEK, L., WAINWRIGHT, M., ANANTHARAM, V., AND NIKOLIĆ, B. 2007. Quantization effects of low-density parity-check decoders. In *Proceedings of IEEE International Conference on Communications*.

Received November 2006; revised June 2007; accepted June 2007