FILE SYSTEM ACCESS FROM RECONFIGURABLE FPGA HARDWARE PROCESSES IN BORPH

Hayden Kwok-Hay So
Department of Electrical and Electronic Engineering
University of Hong Kong, Hong Kong
hso@eee.hku.hk

Robert Brodersen
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
rb@eecs.berkeley.edu

ABSTRACT

This paper presents the design and implementation of BORPH's kernel file system layer that provides FPGA processes direct access to the general file system. Using a semantics resembling that of conventional UNIX file I/Os, an FPGA accesses the file system through a special hardware system call interface. By extending the semantics of a UNIX pipe, a single file system access mechanism is used for both regular file I/O, as well as for hardware/software and hardware/hardware data streaming. An FPGA design may switch between different communication modes dynamically during run time by means of file redirection. Design trade-offs among system manageability, user usability and application performance are explored. An example of constructing a video processing system during run time using commodity software and FPGA applications connected by pipes is used to demonstrate the feasibility and potential of such FPGA-centric file system access capability.

1. INTRODUCTION

This paper presents a new FPGA-centric communication model enabled by the BORPH operating system[1, 2] that addresses the hardware/software communication interface problem in reconfigurable computers. In particular, we will focus on the design and implementation of BORPH's hardware system call interface and its kernel file system layer.

Figure 1 shows the logical organization of software and FPGA resources in a system managed by BORPH. In such system, user FPGA applications, called *gateware*, execute as special *hardware processes* independent of any controlling software. To the rest of the system, a hardware process is the same as any other normal UNIX processes, except they execute on FPGAs instead of using up CPU time slices. Because of this hardware process model, communication between gateware and the rest of the system is not necessarily controlled by any software program. Instead, gateware designs may take an *active* role in data I/O by initiating communication to the rest of the system.

BORPH extends the standard UNIX kernel system call concept to FPGA gateware by defining a *hardware system call* interface. To enable general UNIX file system access from FPGA gateware, a set of file system access hardware system calls is defined. With BORPH's backward compatibility with the Linux OS, standard commodity software, as well as user developed software, may execute concurrently with FPGA gateware in the system. In a UNIX system, most logical entities that enable inter-process communication (IPC), such as sockets and pipes, are abstracted and accessed as files. As a result, enabling general file system access from FPGA gateware not only provide a unified hardware/software communication interface that is well understood by both software and hardware designers, but it also opens up opportunities for novel hardware/software communication techniques that are previously impossible without BORPH's hardware-centric execution model.

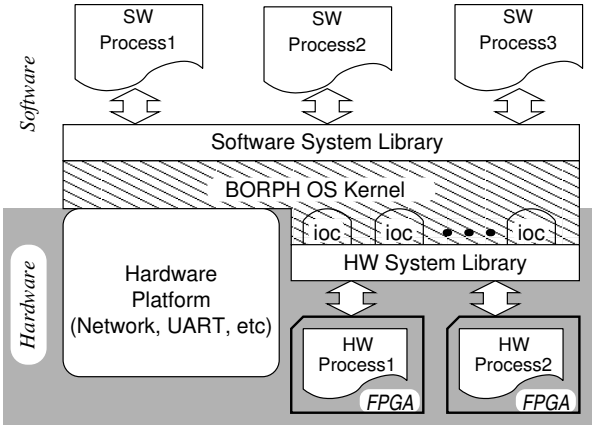

Fig. 1: Logical organization of BORPH system.

2. DESIGN ISSUES

In a UNIX system, file I/O is the main mechanism for a process to communicate with the rest of the system. Apart from accessing regular data from backing stores, the same UNIX file abstraction is also used to access most other inter-process communication entities including sockets, pipelines, FIFOs and even device driver handles. Therefore, providing gateware designs *active* access to the general file sys-

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

tem layer will open up many novel communication scenarios previously impossible.

Furthermore, UNIX files are natural representations for data streams. After all, the unidirectional UNIX pipeline, or *pipe*, is logically identical to hardware FIFOs that are widely used for data I/O in FPGA designs. As many researchers have observed[3, 4], streams are natural representations for gateware data I/Os. Therefore, it is natural to provide file system access to a gateware design that is executing as a hardware process in BORPH.

2.1. High Speed Data Access

Although logically identical, in practice, gateware streaming data access demands a much higher I/O performance than software access. As a result, the impact of kernel overhead during data I/O is also higher.

Unfortunately, because of the physical separation, the kernel overhead for accessing a regular file from a hardware process is usually higher than accessing from software. Luckily, we have observed that hardware processes almost always access files sequentially as stream of data. Therefore, pre-fetching works very well for such accesses.

In our current implementation, we have optimized one special case when both ends of a file connection are hardware processes. In this scenario, the kernel sets up a direct connection between the two gateware designs. This allows them to subsequently communicate directly without kernel's interruption in a true synchronous data flow (SDF) fashion.

2.2. Dynamic File Access Mode Switching

Although the direct hardware/hardware connection suggested in previous subsection addresses the need for high speed data streaming, it inevitably breaks the standard request-acknowledge file access operating mode of UNIX. In a standard UNIX system, a process must request data from the kernel first before the kernel replies the process with the requested data. However, FPGA gateware designs, especially most digital signal processing designs, often assume a data flow model in which data are pushed into a design without explicit request. Furthermore, the target of a file I/O operation, and thus its file access mode, may only be determined during run time due to file I/O redirections.

As a result, the BORPH file system layer must be capable of handling such run time file I/O redirections. Not only must the kernel be able to alter the target of file I/O dynamically, it must also be able to change the file access mode according to the capabilities of the redirected file.

2.3. Blocking vs. Nonblocking File Access

In a UNIX system, during the period when the OS kernel services a system call, the issuing user process is usually blocked. Once the system call is serviced, the calling process is resumed at the point where it was suspended, having the illusion that the system call has completed instantaneously. However, gateware exhibit a parallel computation model in which every part of the design executes concurrently. Therefore, it is unwise for the kernel to block the entire hardware process using techniques such as clock gating.

As a result, all hardware system calls are executed asynchronously to the user gateware designs in our current implementation. User designs must handle signals from the kernel that indicate the progress of a system call.

2.4. Concurrent File System Access

Since gateware designs compute in parallel, it is also possible that multiple portions of the same design demand OS services concurrently. As oppose to the "logical" concurrent system calls generated by multi-threaded software applications, gateware designs may issue file system requests on the exact same hardware cycle. Therefore, the OS kernel must physically be able to service such concurrent requests.

2.5. Partial Result and Error Status

The semantics of standard software file access system calls state that the OS kernel may return less data or commit less data than what the user has requested. It is useful when the file does not have enough data to return, or when the user's disk quota has exceeded. Furthermore, the kernel may return status code that indicates I/O error or reaching end-of-file.

While such concept of partial data delivery and error reporting is common among software developers, they are seldom considered by gateware designers. Our current implementation retains such software semantics and mandates gateware designs to handle them gracefully. A set of user-space gateware libraries is provided to hardware-centric designers to smooth their development efforts.

3. CURRENT IMPLEMENTATION

BORPH is currently implemented on the BEE2 systems[5]. The BORPH kernel is divided into two logical entities. The main kernel, MK, is implemented as an enhanced version of a Linux 2.4.30 kernel in the control FPGA. It performs conventional OS functions and manages software processes.

The second logical part of the BORPH kernel is a set of distributed micro-kernels called UKs. As shown in Figure 2, UKs are implemented as gateware to provide low level OS support for its corresponding user gateware design at hardware speed. The focus of the remaining of this section will be on the design of UK and its interface with user gateware.

3.1. Hardware System Call Interface

In a typical UNIX system, user programs request services from the OS kernel through a set of predefined *system calls*. BORPH defines a similar interface in FPGA through which a user gateware application accesses OS services. Similar to a software OS application binary interface (ABI), the

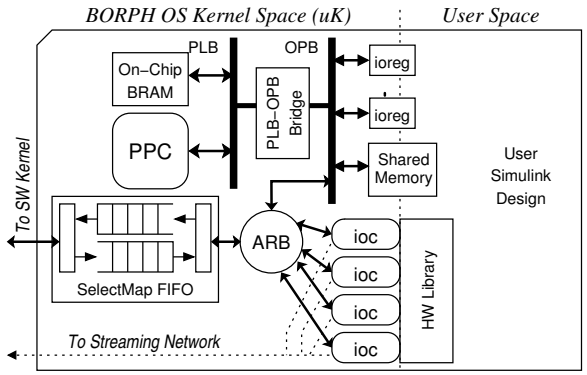

Fig. 2: Each user FPGA on BEE2 hosts BORPH's gateware microkernel UK and a user application. Access to file system is performed through one of the 4 identical `iocs`.

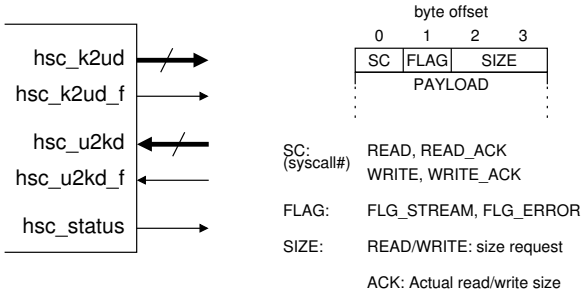

Fig. 3: Left: One copy of `ioc` with which BORPH's hardware system call interface is built. Right: Format of packet transmitted through `hsc`.

hardware system call interface is kept at the lowest possible physical level such that it is generic and design language independent. As a result, the current BORPH hardware system call interface is relatively simple.

BORPH's `hsc` is physically realized as a set of `iocs`. As shown on left hand side of Figure 3, each `ioc` essentially consists of two 8-bits wide unidirectional connections guarded with 1-bit valid (framing) signal. To allow concurrent hardware system calls, BORPH's hardware kernel interface has 4 identical copies of `iocs` dedicated to each user gateware application as shown in Figure 2.

The actual hardware kernel system calls are implemented through a message passing protocol between the kernel and the user. As shown on the right hand side of Figure 3, the packet format is simple by design for easy extensions and implementations. The protocol mandates a request-acknowledge sequence between user and kernel for most system calls. Acknowledgment messages are used by the kernel to return any requested data or error status to the user.

3.2. Implementation of file read/write system calls

In our implementation of READ and WRITE system calls, the actual amount of data written or read is returned in the

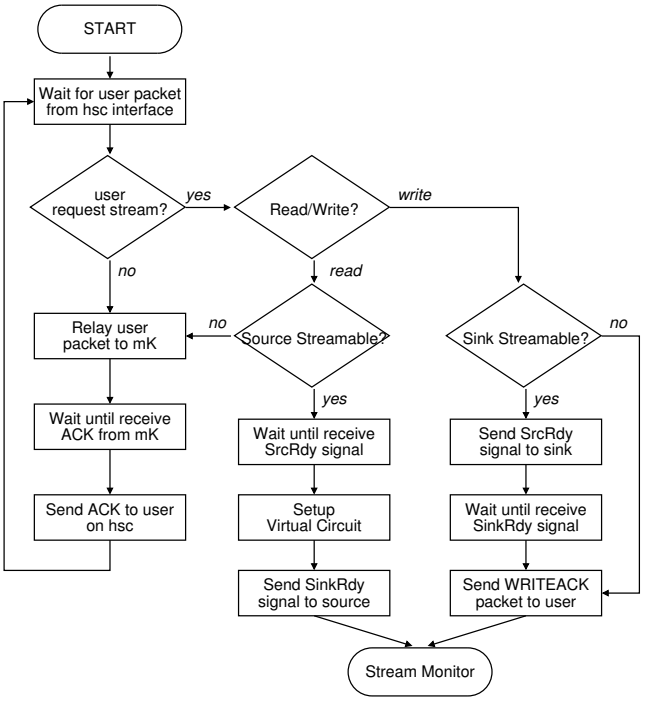

Fig. 4: A simplified flow diagram for `iock` packet handling. A streaming data connection is setup iff both source and sink are streamable and are ready to stream data.

SIZE field of the corresponding ACK message. Following UNIX convention, this size is not necessarily the same as the amount in the original request packet. Furthermore, end-of-file status and any error condition are flagged in the FLAG field, with the error code embedded in the SIZE field.

Note that there is no file identifier or current file position specified in our packet definition. This is designed both as a way to simplify design for user, and to allow such information be managed by the hardware kernel. The physical location of an `ioc` serves as a UNIX file descriptor. The logical file descriptor number corresponds to this `ioc` is stored internally when the file is *open*-ed. Furthermore, a file's current position is also stored and updated internally with each read/write call.

3.3. Kernel Side Control

On the kernel side of an `ioc` is an `iock`. It handles user file system requests and switches between different file access modes dynamically during run time.

Normally a gateware design access BORPH's file system by sending READ/WRITE messages to the kernel. In this mode, `iock` relays the user request to MK. For a file read, it signals to the user to suspend while it buffers any data returned by MK. For writes, it buffers the data before sending them to MK so that the user gateware design can resume operation earlier. `iock` answers a user's read/write requests by the corresponding READACK and WRITEACK messages.

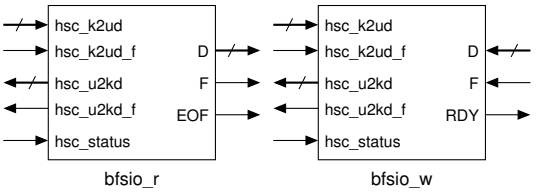

Fig. 5: The `bfsio` library contains two blocks. The read (bfsio_r) block generates read requests on behalf of the user and relay only data payload and EOF information to user. The write (bfsio_w) translates a stream of data from user into individual write request packets to the kernel.

A user design *may* opt for high-speed, cycle-accurate data streaming access to the file system during run time. In respond to such request, `iock` checks if its corresponding connection peer is capable of data streaming (called *streamable*) based on run time information from MK. If both sides of a file connection are streamable, then `iock` sets up a direct connection between the two using the protocol depicted in Figure 4. Once the connection is set up, `iock` restraints its task to non-intrusive monitoring, thereby allowing cycle-accurate communication between the two hardware processes. To keep the `ioc` user interface unified, cycle-accurate data streams are implemented as READ and WRITE packets with *infinite* payloads.

3.4. User Side Gateware Library

Although the complexity of `hsc` packet protocol has already been kept to a minimum, it is still cumbersome for gateware designers. Therefore, a layer of gateware library that takes on a similar role as `libc` in standard UNIX system is introduced to further isolate user applications from the complexity of kernel interactions. To demonstrate this idea, we have created a file access library for synchronous data flow designs as shown in Figure 5. Reducing all kernel complexities to just a simple byte-wide streaming data port with simple flow control, these blocks allow users to focus on actual application development.

4. RESULT

As a proof-of-concept, we have developed a simple gateware Sobel edge detector (`yuvedgdet.bof`) that works with the standard Linux MJPEG tools[6]. The MJPEG tools is a set of small Linux programs that collectively perform complex video editing functions. Most of the programs in the tool set communicate with one another via standard input/output using a predefined raw video format.

Using BORPH's file system support, we are able to perform simple gateware video edge detection by interoperating with rest of the software system as follow:

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

Table 1: FPGA resource used by OS vs. user for a hardware Sobel edge detector.

Resource	Avail	OS	HW lib	App	OS/Avail
Slice	33088	2127	178	95	6.5%
RAMB16	328	38	1	9	11.5%
PowerPC	2	1	0	0	50.0%

In this example, `test.avi` is a 6 mega bytes video file while `lav2yuv` and `mpeg2enc` are unmodified software video decoder/encoder respectively. Processing the video took 169.9s using this mixed software/gateware pipeline. This is slightly slower than a software-only pipeline that finishes in 143.9s due to big I/O overhead in our current suboptimal implementation.

Table 1 shows an analysis of post-mapped resource utilization of `yuvedgdet.bof`. The core edge detection algorithm consumes only 95 slices. 178 slices are consumed by the two hardware library blocks `bfsio_r` and `bfsio_w`. The rest of the design, listed under OS, is devoted to UK.

Despite the complex functionalities of UK, it consumes only 6.5% of the entire FPGA. Even when taking resources used by hardware libraries into account, less than 10% of the FPGA is needed to provide the file system and other system functionalities described in this paper.

5. CONCLUSION

In this paper, we have presented the design and implementation of BORPH's hardware system call interface and file system layer. This general UNIX file system access enables user gateware designs to initiate FPGA-centric data I/O. The UNIX pipe abstraction is extended to represent cycle-accurate hardware/hardware communication. Our proof-of-concept implementation incurs an overhead of less than 10% of a user FPGA. We have demonstrated the feasibility of combining user-developed gateware with unmodified Linux software at run time to process video files.

6. REFERENCES

[1] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *Trans. on Embedded Comp. Sys.*, vol. 7, no. 2, pp. 1–28, 2008.

[2] H. K.-H. So and R. W. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *Proc. FPL'06*, 2006, pp. 349–354.

[3] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor." in *5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, Napa Valley, CA, USA, Apr.16–18, 1997, pp. 12–21.

[4] M. B. Gokhale *et al.*, "Stream-oriented fpga computing in the streams-c high level language," in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2000, p. 49.

[5] C. Chang *et al.*, "BEE2: A high-end reconfigurable computing system," *IEEE Design & Test*, vol. 22, no. 2, pp. 114–125, 2005.

[6] mjpegtools. [Online]. Available: http://mjpeg.sourceforge.net