FISH: Linux System Calls for FPGA Accelerators

by

Kevin Nam

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

Copyright © 2017 by Kevin Nam

Abstract

FISH: Linux System Calls for FPGA Accelerators

Kevin Nam

Master of Applied Science

Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

2017

This work presents the FISH (FPGA-Initiated Software-Handled) framework which allows FPGA accelerators to make system calls to the Linux operating system in CPU-FPGA systems. A special FISH Linux kernel module running on the CPU provides a system call interface for FPGA accelerators, much like the ABI which exists for software programs. The work presents the proposed framework that facilitates the FPGA to Linux kernel system call interface. The work also presents a proof-of-concept implementation of the framework running on an Intel Cyclone V SoC device. The FISH framework could be especially useful for high-level synthesis (HLS) by making it possible to synthesize software code that contains system calls. This would expand the subset of code that is currently synthesizable, and allow behavior in accelerators that was previously not possible.

This work was presented in the 2017 International Conference on Field-Programming Logic in Ghent. The corresponding paper, titled “FISH: Linux System Calls for FPGA Accelerators” is available in the conference proceedings.

Acknowledgments

I would like to thank my research supervisor Prof. Stephen Brown for all the opportunities, knowledge, and experience that he has given me over the years.

I would like to thank my colleague Blair Fort for all his input and expertise whenever I had a question about anything.

Lastly I would like to recognize the Intel FPGA University Program for board donations that allowed me to implement my design.

Table of Contents

Acknowledgments.....	iii
Table of Contents.....	iv
List of Figures	vii
List of Abbreviations	ix
1 Introduction.....	1
1.1 Motivation.....	1
1.2 Contributions.....	2
1.3 Thesis Organization	2
2 Background.....	4
2.1 A Survey of Modern CPU+FPGA Systems.....	4
2.1.1 SoC FPGA Devices	4
2.1.2 Intel Xeon+FPGA	6
2.1.3 Workstation with PCIe FPGA Card	6
2.2 CPU-FPGA Hybrid Applications	6
2.3 High-Level Synthesis.....	7
2.3.1 HLS Limitations on System Calls	9
2.4 Linux Operating System	10
2.5 Linux System Calls	11

3	FPGA-Initiated Linux System Calls: Potential Use Cases	13
3.1	Filesystem Access	13
3.2	Memory Allocation	14
3.3	Device Access	15
3.4	Thread-Accelerator Synchronization	16
4	The FPGA-Initiated Software-Handled Framework	17
4.1	A Standard System Call Interface for FPGA Accelerators	17
4.2	Handling an FPGA-Initiated System Call	19
4.3	FPGA to Kernel Module Communication	20
4.4	Handling the System Call Request in the FISH Kernel Module	22
4.5	Handling System Calls with Persistent Effects	23
5	Framework implementation	25
5.1	Intel DE1-SoC Development Board	25
5.2	The DE1-SoC Linux Distribution	27
5.3	Allocating the Shared Data Buffers	28
5.4	Implementing the Host Program	28
5.5	Implementing the Accelerator	28
5.6	Implementing the FISH Kernel Module	31
6	Assessing the Framework Implementation	35

6.1	Verifying Correct Behavior	35
6.2	Performance	36
7	Related Works	39
8	Conclusions and Future Work	41
9	Appendix.....	42
9.1	Host Program Code	42
9.2	FISH Kernel Module Code	43
9.2.1	Main Kernel Module Code	43
9.2.2	Open() Handler	45
9.2.3	Read() Handler	47
9.2.4	Write() Handler	48
9.2.5	Close() Handler.....	48
9.3	CMA Driver Kernel Module Code	49
9.4	LegUp C-Language Code for Accelerator	52
9.5	LegUp-Synthesized Accelerator Code in Verilog HDL	52
9.6	Verilog HDL Module that Initiates the Open System Call	52
9.7	Verilog HDL Module that Initiates the Write System Call	56
9.8	Verilog HDL Module that Initiates the Close System Call	60
9.9	Linux Kernel Config File	62

10	References	63
----	------------------	----

List of Figures

Figure 1	Example of high level synthesis with C-code as input	8
Figure 2	Filesystem accesses from the FPGA accelerator	13
Figure 3:	Parallels between the traditional system call convention and the FPGA-initiated, SW- handled convention	18
Figure 4:	Steps for handling an FPGA-initiated system call	19
Figure 5	Block diagram of the DE1-SoC board [12]	26
Figure 6:	C-language function that was accelerated using LegUp HLS.....	29
Figure 7	Register interface of the FISH-compatible accelerator	30
Figure 8:	Registering a threaded IRQ handler	32
Figure 9:	IRQ handler code (abbreviated)	33
Figure 10:	Abbreviated code for the open system call handler	34
Figure 11:	An accelerator that opens a file and returns the file descriptor	35
Figure 12:	Latencies of Pure-SW and HW-initiated system calls.....	37
Figure 13	C-language code for the host program that launches the accelerator	43
Figure 14	Kernel module code for allocating physically contiguous memory using CMA	51
Figure 15	Accelerator code (in C) that was synthesized using LegUP	52
Figure 16	Accelerator code (in Verilog HDL) that was generated by LegUP	52

Figure 17 Verilog module that initiates the open() system call from the accelerator	56
Figure 18 Verilog module that initiates the write() system call from the accelerator	59
Figure 19 Verilog module that initiates the close() system call from the accelerator	62
Figure 20 Linux kernel configuration settings for the custom 3.18 kernel used in the DE1-SoC	
Linux distribution	62

List of Abbreviations

API	Application Programming Interface
ARM	Advanced RISC Machines
CMA	Contiguous Memory Allocator
CPU	Central Processing Unit
CWD	Current Working Directory
DMA	Direct Memory Access
FD	File Descriptor
FISH	FPGA-Initiated Software-Handled
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HLS	High-Level Synthesis
HPS	Hard Processor System
IRQ	Interrupt Request
KM	Kernel Module
LKM	Linux Kernel Module
OS	Operating System
PCIe	Peripheral Component Interconnect Express Bus
PID	Process ID
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
SGDMA	Scatter-Gather DMA
SoC	System on Chip

1 Introduction

1.1 Motivation

FPGAs are becoming increasingly integrated into modern day computers. This can be seen across produce ranges of FPGA manufacturers, such as the Intel/Altera SoC [1], the Xilinx SoC/MPSoC [2], and Intel Xeon+FPGA processors [3]. To support the programming of these non-traditional heterogeneous systems, much work has been done in area of CPU+FPGA co-design to provide programming frameworks that encompass the CPU+FPGA system. Popular programming frameworks include the Intel OpenCL SDK [4], Xilinx Vivado High-Level Synthesis tool [5], and LegUP HLS [6]. These frameworks allow the development of hybrid applications, which are applications consisting of “host” software running on the CPU, and accelerator(s) running on the FPGA which work together to provide the application behavior. Hybrid applications are becoming increasingly popular in the datacenter, with cloud service providers such as Amazon AWS and Microsoft Azure adding FPGAs to their datacenters to allow FPGA acceleration of tasks. Often, CPU+FPGA heterogeneous systems run an operating system like Linux. The Linux OS is frequently used as it is free and offers system management tools, greater productivity through access to higher-level APIs and device drivers, and a myriad of freely available Linux software. When a developer creates a hybrid application for a CPU-FPGA system running Linux, there are many considerations involved. One consideration is how to provide the FPGA circuitry access to Linux-provided resources and devices. In a purely-software application, the software program running on Linux requests access to these resources via Linux system calls. Unfortunately, the Linux system call interface is only available to software, and a circuit in

the FPGA is unable to use it. This means that traditionally, FPGA accelerators could not access Linux-provided resources directly, and had to instead rely on the host software to request the resources and relay them to the FPGA accelerator. Not only does the host program intermediary approach add latency, this approach also means that the developer must code the host program with this behavior in mind, adding to development complexity. In a technological landscape where CPU and FPGA are more tightly integrated than ever before, it seems that a way for FPGAs to directly make system calls to the operating system could be a potential benefit to hybrid applications.

1.2 Contributions

The objective of this work is to provide a standardized interface to FPGA accelerators for making Linux system calls. Key contributions are:

1. Development of an FPGA to operating system interface for making system calls.
2. Proof of concept implementation of the system call interface.

1.3 Thesis Organization

The rest of this document is organized as follows:

Section 2 provides background information on current CPU+FPGA systems, current approaches to CPU+FPGA hybrid applications, high-level synthesis, the Linux operating system, and Linux system calls.

Section 3 presents some possible use cases of a system call interface to FPGAs. It discusses potential uses like filesystem access, memory allocation, device access, and thread synchronization.

Section 4 presents our proposed FPGA-Initiated, Software-Handled (FISH) framework. It describes the system call interface to FPGAs, as well as the framework for handling the system calls on the operating-system side. It outlines and describes the key components of this framework, which include the host program, the FPGA accelerator, and the FISH Linux kernel module.

Section 5 describes our proof of concept implementation of the FISH framework. It goes over key considerations for implementing the framework such as the necessary modifications to existing code, how the FISH kernel module was implemented, and the hardware-architecture-specific details.

Section 5 briefly assesses the performance of our proof of concept implementation, to compare the runtime of the FPGA-to-CPU system calls against the best possible case (CPU-to-CPU system calls).

Section 7 discusses some prior related works, and Section 8 presents concluding remarks and suggestions for future work.

2 Background

2.1 A Survey of Modern CPU+FPGA Systems

2.1.1 SoC FPGA Devices

Top FPGA vendors Intel and Xilinx offer what are commonly referred to as SoC FPGA devices. Simply put, they are devices that comprise an FPGA and a CPU in the same die with some channels of communication between the two halves. The offerings from both companies are largely similar, and both vendors offer devices that contain an FPGA of varying size and capability, in addition to a single or multicore ARM processor ranging from the ARM Cortex-A9, ARM Cortex-A53, to the ARM Cortex-R5.

Communication between the FPGA and CPU are carried out through hardened communication channels built into the chip. In the case of the Intel Cyclone V SoC, three memory-mapped bridges (connected to AXI-compatible masters/slaves at either end) exist to facilitate communication. Two of the bridges, the HPS2FPGA and the LWHPS2FPGA, facilitate CPU to FPGA communication, while the FPGA2HPS bridge facilitates FPGA to CPU communication. The bridges are hard-wired to interconnect in the CPU-side of the chip, but can be configured on the FPGA-side to connect to any user-defined circuitry capable of AXI communication. Developers use Intel's QSys software to configure the connections on the FPGA side and generate the corresponding HDL for compilation.

In addition to the CPU and the FPGA, vendors place hardened controllers like USB, SDRAM, Ethernet, SDMMC, SPI, and I2C, so developers do not have to use FPGA fabric for these

commonly used tasks. An important note is that these peripheral devices are generally placed in the CPU half of the chip, allowing direct memory-mapped access to their register interfaces by the CPU. When Linux runs on the SoC device, its built-in drivers for these peripherals allow developers to use higher-level API to access and operate these devices, offering a significant productivity advantage. This is in contrast to having to write HDL to implement the equivalent peripheral device in the FPGA, and writing FSMs in hardware or baremetal code running on a soft-CPU in the FPGA to control the peripheral device. When writing a Linux hybrid application that uses one of the Linux -controlled devices, the host program (the portion of the application running on the CPU) would typically have to facilitate the operations. If the FPGA circuit required one of these devices, the host program would likely have to be the intermediary, potentially adding unwanted latency. Additionally, on the same system, there may be standalone FPGA circuits that require access to one of the Linux-controlled devices. While the SoC FPGA chip offers FPGA-to-CPU communication channel(s) that would allow the standalone circuit to directly access the device, doing so would require synchronization with the Linux OS due to potential contention. In both cases of FPGA circuitry requiring Linux-controlled devices mentioned above, it would be beneficial to have some method by which an FPGA circuit could directly request OS services and devices through system calls. This solves the unnecessary latency added by the host program intermediary, and solves the issue of synchronization with the Linux OS in the case of standalone FPGA circuit access (since it would be the Linux OS that would handle the requests from the FPGA circuit).

2.1.2 Intel Xeon+FPGA

Although not yet available, the Intel Xeon+FPGA device is significant due to the impact it may have on the datacenter market. As of December 2016, Intel Xeon processors had a 90%+ market share of the world's datacenters [7]. A push by Intel to incorporate FPGA devices into the data center would mean a huge increase overall in the world's usage of FPGA devices and in turn, the prevalence of CPU+FPGA hybrid applications. This would mean an increased demand for CPU-FPGA hybrid application development resources and methods, which the work presented in this thesis seeks to augment.

2.1.3 Workstation with PCIe FPGA Card

Vendors offer FPGA PCIe cards for the hardware-acceleration market [8] [9]. Customers ranging from scientists and researchers to stock traders who wish to accelerate their workload via FPGA can purchase these FPGA boards and plug them into their workstations. These boards tend to contain higher-end FPGA devices, such as the Stratix and Arria families from Intel or Virtex and Kintex from Xilinx. Once plugged into the motherboard of their workstation, communication between CPU and FPGA is facilitated through the PCIe lane(s). A PCIe driver is installed on the OS to transact PCIe packets to and from the FPGA board.

2.2 CPU-FPGA Hybrid Applications

Software applications are ones that run on a traditional computer as sequences of instructions that are executed serially by the CPU. FPGA applications are ones that get placed inside an FPGA as a hardware circuit. CPU-FPGA Hybrid applications are ones comprising software and hardware (placed in FPGA) where the two halves work together to accomplish some task(s).

Hybrid applications allow developers to divide up the workload of their application between the CPU and the FPGA, wherever the strength of each device is beneficial. For example, a developer may run some house-keeping tasks that are run only a few times in the application's execution on the CPU, where the CPU's general-purpose nature allows code to be executed with fairly low overhead. On the other hand, work that is highly repetitive and applied to large sets of data may benefit from running as a specialized accelerator placed in the FPGA, as it may provide better throughput.

An FPGA accelerator typically has the performance advantage over the CPU. The accelerator in the FPGA can be pipelined for the specific task (allowing high pipeline-parallelism) and duplicated multiple times in the FPGA (allowing data parallelism). For example, consider an algorithm that requires 4000 instructions on the CPU. As CPUs are not pipelined to such lengths, the CPU is incapable of fully pipelining the algorithm. If CPU ran at 4 GHz, the CPU would complete $4 \text{ GHz} / 4000 = 1 \text{ Million}$ results per second. On the other hand, if the algorithm was turned into a pipelined circuit on the FPGA capable of yielding a result per cycle, and if the circuit ran at 100 MHz, it could yield 100 million results per second. Assuming no data dependency, this circuit could then be duplicated multiple times, allowing even greater throughput.

2.3 High-Level Synthesis

When creating a hybrid application, developers can create all parts of the hybrid application from scratch, which has the advantage of being optimizable at the finest granularity, but the disadvantage of being difficult and time-consuming. The most time-consuming part of creating

the application from scratch is writing the HDL (Hardware Description Language) code to describe the FPGA circuitry. Here, the developer would have to specify their accelerator circuit by placing registers, connecting wires, creating state machines, and so on, which is significantly more time consuming than writing the equivalent functions in a software language.

Figure 1 Example of high level synthesis with C-code as input

Instead of writing HDL, developers can use high-level synthesis (HLS) frameworks, such as the Intel OpenCL SDK [4] or LegUp [6]. With HLS, the application is typically written in some high-level (typically software-language) code to describe the desired logic to be accelerated in the FPGA. The HLS tool then automatically synthesizes the code into a hardware circuit in the FPGA. An example of such synthesis is shown in Figure 1 where C-code describing an integer adder is synthesized by an HLS tool. System-specific details like CPU to FPGA communication, device drivers, and FPGA peripheral circuitry are also usually handled by the HLS tool, further reducing development times. HLS tools have been gaining popularity as the performance of the auto-generated accelerators have been deemed “good enough” whilst saving tremendous development time. Furthermore, HLS tools open up the possibility of using FPGAs to software developers who are unfamiliar with HDL. For these developers, they have no option but to use an HLS tool.

LegUp is an open source HLS tool produced by the LegUp group at the University of Toronto [6]. The tool accepts ANSI C code as the input, converts the code to LLVM IR, then uses LLVM passes to turn the input into HDL. The HDL is then sent through the Intel Quartus tool where it is compiled into an FPGA programming file that contains the accelerators.

Intel OpenCL SDK is a commercial HLS tool sold by Intel Corporation [4]. The tool, as its name suggests, accepts OpenCL code as input, and generates an FPGA programming file containing the accelerator.

2.3.1 HLS Limitations on System Calls

Today's HLS tools often place limitations on what can and cannot be synthesized. Common among these limitations is the inability to synthesize system calls. For example, whenever the LegUp HLS tool encounters a system call in the C code to be accelerated, it instantiates an empty HDL module in place of the system call function. One exception in LegUp is `printf`, which it handles specially by placing a Verilog `$display` statement in place. This however does not truly produce a circuit to handle `printf`. It is instead more of a convenience feature for helping developers debug their resulting synthesized circuits using ModelSim.

The system-call limitation can be a problem, since the input to HLS is usually some software code, and software code tends to contain many system calls (often implicitly) to carry out common tasks like printing to a terminal, and accessing files and devices. To this end, the ability to make system calls from the FPGA is attractive as it could ultimately allow system call-laden software code to be accelerated by high-level synthesis tools. The work presented in this thesis

works towards achieving an HLS tool that can transparently and seamlessly compile system calls to yield an FPGA accelerator that can make system calls as if it were running on the CPU.

2.4 Linux Operating System

Linux is a free, open-source operating system that is found in many kinds of devices such as phones, computers, servers, embedded devices, and cars. There are many distributions of Linux, with some of the most popular being Android, Ubuntu, and Red Hat.

At the core of every Linux distribution is the Linux kernel, which is the program that handles essential OS functionality like scheduling, virtualization, resource allocation, and device drivers. This functionality gives tremendous benefit to developers creating applications. The scheduling and resource allocation means that a developer can run multiple applications on the system and let the OS handle the job of divvying up the resources between them. Device drivers built into the Linux kernel provide support for many widely-used devices, and support for new devices are actively added to the kernel. This means that the developer does not need to worry about low-level interaction with devices. Rather, they can use portable higher-level API provided by Linux to access the devices that are underneath. This further reduces development time of applications.

As the kernel controls the system's resources, one of its jobs is to provide these resources to user-level programs. User-level programs are the ones written by developers that run on top of the operating system. A key difference between a user-level program and the OS is the privilege with which they run. The OS is very privileged – it can access secure (privileged) CPU states, directly access the system's devices, and access any part of memory. For security reasons, the

user-level program runs without such privileges. Whenever the user-level program needs a privileged service or resource, it cannot access them directly, but instead must request them from the Linux kernel by making a Linux system call.

2.5 Linux System Calls

The Linux Programmer's Manual describes the system call as the fundamental interface between an application and the Linux kernel [10]. In practice, a system call is a function that is callable by user processes for requesting some service or resource from the OS. Linux provides hundreds of different system calls, each of which provides a particular resource. For example, the `open()` system call opens a file in the Linux-managed filesystem, and the `write()` system call writes data to the opened file. To identify the system call, each one is assigned a unique numerical ID.

When a program makes a system call, it must use the Application Binary Interface (ABI) provided by Linux. The ABI requires a program to follow a calling convention that dictates rules like the number of arguments to the call, to which CPU register each argument is stored, and which CPU instruction is used to initiate the system call. The convention depends on the CPU architecture that is being used, but in general the user program stores the arguments to the system call in the appropriate CPU registers, then calls the architecture-specific system call instruction whilst providing the system call ID. When the system call instruction runs, the CPU changes from user mode to privileged mode, and the ID is used to jump to the appropriate subroutine that handles the system call being requested.

The goal of this work is to provide an interface that FPGA accelerators can use to request system calls from the Linux kernel, similar to what an ABI provides for software programs. The framework for providing such an interface is described in Section 4.

3 FPGA-Initiated Linux System Calls: Potential Use Cases

3.1 Filesystem Access

Often applications access the filesystem to store and read data. When a hybrid application needs to access the filesystem, the common practice today is to do the file accesses in the host program running on the processor. This is illustrated in Figure 2 as the “Traditional Filesystem Access from FPGA”. If a read is required, the host program would read a file from the filesystem using the `read()` system call, and provide the data buffer to the FPGA accelerator. If a write is required, the host program would accept a data buffer from the FPGA accelerator, and write the data to the filesystem using the `write()` system call.

With the FPGA-initiated Linux system calls, the intermediary host program is no longer required. As shown in Figure 2 as the “Direct Filesystem Access from FPGA”, the accelerator would bypass the host program and directly access the file system as needed. This eliminates the accelerator-to-host communication, which potentially reduces latency.

Figure 2 Filesystem accesses from the FPGA accelerator

3.2 Memory Allocation

In situations where the Linux OS controls the system's main memory, the FPGA accelerator may need to request some memory be allocated for its use. In a user-level program, memory is requested using the `malloc()` system call. The `malloc()` calls would return a virtual address (in the calling user process's address space) to the allocated memory buffer which the user program can simply use without any additional work. From the FPGA however, things are not so simple. The `malloc()` call provides a virtual address in the process's virtual address space. The circuit does not have such a space, as (at least in the case of the Cyclone V SoC used for experimentation in this thesis) the FPGA does not access the memory through the MMU that the Linux kernel does. In other words, the FPGA accelerator accesses main memory via physical addresses, not virtual addresses.

Furthermore, the issue of virtual addresses and physical addresses is not the only problem when trying to allocate memory for an FPGA. When Linux allocates memory, it does so in chunks called pages, which are typically 4 kB long. When Linux allocates memory for a program, it decides mappings between pages in the program's virtual address space to the physical memory. Linux does not guarantee that what is contiguous in virtual memory is also contiguous in physical memory. This is acceptable for a user-level program because whenever it accesses memory through the MMU, it consults page tables to determine which physical address maps the virtual address being accessed. The FPGA, again, does not access the memory through the MMU. This means that if it is granted a memory buffer by the Linux kernel which is longer than a page, the buffer must be contiguous in memory for the FPGA to use it.

Luckily, the problems outlined above can be solved by using a series of alternate system calls.

The Linux kernel is capable of allocating physically contiguous memory via its Contiguous Memory Allocator (CMA) feature. By leveraging this feature through system calls, an FPGA accelerator could gain a physically contiguous buffer for its purposes. This means that the system memory can be managed by the OS, and the FPGA can gain the memory that it needs by requesting it from the OS just as if it were a software program.

3.3 Device Access

Depending on the system topology, various devices may be connected to the CPU-side of the device rather than the FPGA. For example, in the Cyclone V SoC, the CPU-side of the system contains the USB, Ethernet, SDRAM controller, and SDMMC among others. If the FPGA accelerator needed to access one of these devices, it would have to gain access by requesting it from the Linux OS through system calls.

In the case of the Cyclone V SoC, the hardware manufacturer provides a hardware interface for FPGA-side components to directly control some of these CPU-side devices. However, even if the accelerator possesses the ability to access the devices directly by addressing their register interfaces, it is inadvisable to do this as the Linux kernel's drivers for these devices may contend for them. If the driver and the FPGA accelerator tried to access some device simultaneously in a destructive manner, erroneous behavior is likely to occur. To facilitate the sharing of these devices between accelerators and user-level programs running on the CPU, there must be one broker for these devices, which should be the Linux kernel. FPGA accelerators and user-level programs all must make the appropriate system calls to the kernel for these devices. Various

system calls may be used for this purpose, including but not limited to `ioctl()`, `read()`, and `write()`.

3.4 Thread-Accelerator Synchronization

An accelerator and a thread running on the CPU may in some situations need to synchronize. In high-performance compute applications where developers create highly parallelized applications, synchronization between CPU threads and FPGA accelerators could be necessary. When synchronizing between software threads in a multithreaded application, developers typically use library function calls (such as those of the `pthread` library) to implement the necessary synchronization through mutual exclusion locks, semaphores, and etc. These functions underneath make system calls to guide the Linux kernel to efficiently schedule the threads in the necessary order. System calls used may include timer-related calls like `sleep()`. By allowing system calls from the FPGA accelerator, a developer using HLS to create the accelerator could potentially use the same synchronization libraries as they would for software to implement thread synchronization in their accelerator. The HLS tool would then synthesize the necessary FPGA circuit to initiate the system calls to the operating system to implement the thread synchronization.

4 The FPGA-Initiated Software-Handled Framework

This section describes the FPGA-Initiated Software-Handled framework for allowing FPGAs to make system calls in CPU-FPGA systems that use the Linux operating system. At a high level, the proposed framework consists of three logical components:

1. Linux host program
2. FPGA accelerator
3. FISH Linux kernel module

The host program runs on the Linux operating system, and is responsible for launching the FPGA accelerator through some interface such as PCIe or AXI. The FPGA accelerator carries out its computation and eventually reaches a point in its operation where it requires a system call. The FPGA accelerator then requests a system call from the FISH Linux kernel module. The FISH kmodule runs as part of the Linux kernel and carries out the system call on behalf of the accelerator. When the system call is complete, the FISH kmodule signals the accelerator, and computation on the accelerator resumes. The following sections describe this framework in more detail.

4.1 A Standard System Call Interface for FPGA Accelerators

Parallels can be drawn between our framework and the traditional ABI used by software, as shown in Figure 3. The accelerator is analogous to the user program, in that both entities require a resource from the OS. Both the accelerator and the user program must prepare for a system call by providing data in the form of a system call ID and arguments to that system call.

While the data remains the same, the medium for transfer is different. While the user program directly writes the data to the CPU registers, the accelerator uses some global memory accessible by the CPU. Once the data is ready, the accelerator defers control to the FISH kmodule, similar to how the user program defers control to kernel-mode execution of the system call handler placed in memory by the Linux kernel. Then, the FISH kernel module will carry out the required tasks for providing the accelerator with the requested resources, as the system call handler would normally.

Figure 3: Parallels between the traditional system call convention and the FPGA-initiated, SW-handled convention

The end goal is to provide some standardized convention for accelerators to make system calls to the Linux kernel. The precise details of the interface may change over time, especially due to changes in CPU-FPGA system architectures as newer devices are released. If a standard

interface is achieved, HLS tool vendors and HDL developers can create FPGA accelerators that adhere to this standard making them portable to any CPU-FPGA system running Linux. This is no different than how a Linux binary of a given CPU architecture can make system calls regardless of system specifics like whether an Intel or AMD CPU is present, or which graphics card or hard drive, etc.

4.2 Handling an FPGA-Initiated System Call

Figure 4: Steps for handling an FPGA-initiated system call

When the FPGA accelerator initiates a system call, it is handled by the process shown in Figure

4. The steps in the process are described below:

1. Accelerator stores in memory the information needed by the FISH kmodule to handle the system call (host program's PID, system call ID, arguments to the system call, and buffers)
2. Accelerator signals the FISH kmodule to start the system call
3. FISH kmodule reads the information in memory
4. FISH kmodule consumes system resources on behalf of the accelerator
5. FISH kmodule modifies the host program's *task_struct* object to ensure persistent change as a result of the system call
6. The result of the system call (if applicable) is written to the buffer in memory
7. FISH kmodule signals the FPGA accelerator that the system call has finished
8. Accelerator reads the results of the system call from the buffer in memory (if applicable)
9. The persistent effects of the system call are visible to the host program in its execution further down the line. No special handling is required; system calls naturally access process state stored in the *task_struct* object.

4.3 FPGA to Kernel Module Communication

The framework requires a signaling method between the FPGA and the FISH kernel module for two purposes:

1. To signal the start of a system call (step 2)
2. To signal the completion of the system call (step 7)

The FISH kmodule remains asleep until it receives a signal from the FPGA. Once signaled, it wakes up and carries out the work necessary to handle the system call. When finished, it sends

the “finished” signal to the FPGA accelerator and goes back to sleep. The accelerator then continues with its computation.

In addition to sending “start” and “finished” signals, the following data must also be passed between the FPGA and the kernel (step 3):

1. Host program’s PID
2. System call ID
3. Arguments to the system call
4. Data buffers

The host program’s PID must be sent to the FISH kmodule so that it is aware of which host program is associated with the accelerator. An important takeaway here is that although both the host program and the FISH kmodule run in the software domain, they remain uncoupled. Since the FISH kmodule is unaware of the host program, the PID must be provided by the host program to the accelerator and in turn, by the accelerator to the FISH kmodule. Although this seems roundabout, this approach allows a single FISH kmodule instance to provide system call services to multiple hybrid applications running on the system. In other words, a single FISH interface can serve all of the FPGA accelerators, similar to how one ABI can be shared by all user programs.

In addition to the PID, the system call ID is necessary to specify which of the approximately 300 Linux system calls is being requested. Some of these system calls require input arguments, which have to be passed from the accelerator to the FISH kmodule. As some of the arguments

may be pointers to buffers, memory for these buffers must be allocated and initialized by the accelerator.

In the diagram shown in Figure 4, the data is transferred through system memory. This is a natural way to pass information, as CPU-FPGA systems generally have a global memory which is accessible by both CPU and FPGA. With other system topologies, an alternative communication method may be used.

4.4 Handling the System Call Request in the FISH Kernel Module

When a Linux user-level program makes a system call, it switches from executing in user mode to executing in kernel mode through a CPU instruction that requests privileged execution. For example, in the ARM instruction set the SVC (Supervisor Call) instruction is used to change the CPU to the privileged mode. After the switch, the now-privileged process executes the subroutines that carry out the system call. Because the FPGA accelerator does not run on the CPU at all, a completely different approach is required. In our framework, the FISH kmodule is used as the intermediary between system resources/services and the FPGA accelerator. This kmodule runs on the CPU in privileged mode, in place of the accelerator, and executes the necessary subroutines.

The FISH kernel module remains asleep while the FPGA accelerator does not require a system call. When the kmodule is awoken by the accelerator (step 2), it determines which system call is being invoked (via system call ID), and the process for which this system call is being invoked (via PID). The kernel module then invokes modified versions of the system call handler subroutines that correspond to the requested system call. As these system-call subroutines

were originally written to be called by a user process, they must first be modified before the FISH kernel module can call them. An example of a necessary modification is when a context-specific call is made by the subroutine (such as one requesting data or settings specific to the current process), which would have different effects being called from a kernel module context rather than from a user process running in kernel mode.

4.5 Handling System Calls with Persistent Effects

Some system calls have a persistent effect on the calling process. For example, a call to `open()` would open a file and allocate a file descriptor to that file. The file descriptor then remains reserved for the duration of the process's execution, or until `close()` is called on that file descriptor. Clearly, some persistence is required when handling system calls. Further, for the FPGA-initiated call to behave seamlessly (indistinguishable from a SW-initiated system call), the persistent change must be made visible not only to the FPGA accelerator, but also to the host program (step 9). Simply put, a file opened by the FPGA accelerator must be accessible by the software, and vice versa.

To keep track of the running processes and their various state data, the Linux kernel uses a table of *task_struct* objects. In this table is a *task_struct* object for every process, which keeps track of process-specific information like execution state, file descriptor tables, child processes, etc. To achieve the required persistent effects of system calls, the FISH kmodule modifies the host process's *task_struct* object (step 5). Then, any system call further down the line (whether it is software or FPGA initiated) consults the *task_struct* to see the persisting effects of prior system calls.

A special case that has to be handled with regards to the *task_struct* table is when the accelerator is designed to be standalone (run without a host program). In such a case, we propose that a placeholder process be created for that accelerator to hold necessary states. If an accelerator makes a system call without its PID register being initialized (IE having a value of 0) then the FISH kernel module can detect this and spawn the necessary placeholder process. Then the FISH kernel module fills in the PID register with the PID of the placeholder process that it has just spawned, linking the placeholder process to future system calls made by the standalone accelerator.

5 Framework implementation

This section describes the completed proof-of-concept implementation of the proposed framework running on an Intel DE1-SoC board containing the Cyclone V SoC (which includes an ARM Cortex-A9 CPU) and 1GB of DDR3 memory. A custom Linux distribution with kernel version 3.18 was run on the ARM processor and DDR3 memory. The DDR3 memory was also used as the global memory, to transfer data between the CPU and the FPGA. The Cyclone V SoC contains an ARM Generic Interrupt Controller (GIC) that allows FPGA to CPU interrupts. These interrupts were used for the FPGA to FISH kmodule signaling.

As a proof-of-concept demonstration, four common system calls for accessing the file system, `open()`, `read()`, `close()`, and `write()`, were implemented. The ability to access the Linux filesystem from the FPGA has been of interest to researchers in the past [11]. We describe the code needed in the host program code for launching the accelerator, the HDL necessary in the FPGA accelerator to initiate the system calls, and the FISH kernel module code for handling the system calls.

5.1 Intel DE1-SoC Development Board

The Intel DE1-SoC board was chosen as the test platform for the FISH framework. This board was chosen as it was supported by LegUp HLS (the tool which we used to synthesize the accelerator HDL), and provided the necessary CPU-FPGA coupling required to demonstrate the framework. The board is manufactured by Terasic Technologies [9] and contains an Intel Cyclone V SoC device. The SoC device contains a dual-core Cortex-A9 processor running at 925

MHz, and FPGA fabric providing 85k logic elements. The board's features are outlined in Figure 5.

Figure 5 Block diagram of the DE1-SoC board [12]

The features of the board necessary to demonstrate the FISH framework are:

- ARM Cortex-A9 Dual Core processor for running Linux, host program, and the FISH kernel module

- 1 GB DDR3 SDRAM for running the programs as well as being the buffer for FPGA to CPU data transfers
- MicroSD card slot for booting Linux from the SD card
- Gigabit Ethernet for developing code on the board via SSH and FTP
- UART to USB for serial terminal communication

5.2 The DE1-SoC Linux Distribution

A custom distribution of Linux designed for the Intel DE1-SoC board was created and used for the proof of concept implementation. The distribution is based on Terasic Technologies' Ubuntu Linux image, but contains an updated Linux kernel version 3.18 instead of the default 3.12. When compiling the Linux kernel, the following key features were enabled (the complete config file used when compiling is provided in Section 9.9):

- Contiguous Memory Allocator (CMA) support, with 256MB reserved for CMA
- Intel/Altera FPGA programming drivers and support

The CMA support is necessary for allocating physically contiguous memory. Our accelerator requires physically contiguous memory as it does not contain a Scatter-Gather DMA (SGDMA). The Intel/Altera FPGA programming support is used to program the FPGA with the accelerator before running the host program. Having this feature meant that all of the requires tasks of running a hybrid application could be done on the board itself without intervention of a host workstation. This allowed creation of scripts that automated the process of running a hybrid application, which included tasks like programming the FPGA, inserting the FISH kernel module, running the CMA allocation driver, and running the host program.

5.3 Allocating the Shared Data Buffers

In order to pass data between the accelerator and the FISH kernel module, the DE1-SoC board's DDR3 memory was used as the shared global memory. Since the Linux operating system runs on and manages this memory, it was necessary to allocate buffers before transferring data. Allocating memory using a normal `malloc()` was problematic, as the allocated memory could be paged and scattered in memory, and the accelerator cannot read discontinuous pages in memory. To solve this problem, Linux's Contiguous Memory Allocator (CMA) was used to reserve a contiguous stretch of memory.

5.4 Implementing the Host Program

The host program launches the accelerator using its memory-mapped register interface, which is access through the HPS-to-FPGA bridge. The memory-mapped interface is accessed by mapping physical addresses to virtual address using `/dev/mem` and `mmap()`. Once mapped, its registers are accessed by the host program. The only implementation detail worthy of mention here is the need to call `getpid()` and pass the program's PID to the accelerator before launching it.

5.5 Implementing the Accelerator

The LegUp HLS tool was used as a starting point to generate the accelerator. The C-language function shown in Figure 6 was accelerated using the HLS tool.

```

int fwrite(){
    int fd = open("test.txt", O_WRONLY, S_IRWXU);
    write(fd, "Hello! The accelerator wrote this file!", 40);
    close(fd);
    return fd;
}

```

Figure 6: C-language function that was accelerated using LegUp HLS

LegUp creates a memory-mapped register interface for the accelerator that contains control and status registers that allow the host program to launch it and determine its state. The generated accelerator's memory-mapped interface was connected to the Cyclone V SoC's HPS-to-FPGA bridge using the Intel Qsys tool [13]. By default, LegUp's register interface contains the 64-bit DATA register for storing the return value of the accelerator, a STATUS register that holds bits for monitoring the accelerator's status, and ARGX registers for passing arguments to the accelerator. In addition to the default registers, the following registers were manually added:

1. PID
2. System Call ID
3. System Call Arguments 0 – 3
4. System Call Return Value

This resulted in a final register interface as shown in Figure 7.

Offset	Register
+0x00	DATA
+0x08	STATUS
+0x0C	SC_ID
+0x10	SC_STATUS (not used for open,close,write)
+0x14	SC_RETVAL
+0x18	SC_PID
+0x1C	SC_ARG0
+0x20	SC_ARG1

+0x24	SC_ARG2
+0x28	SC_ARG3

Figure 7 Register interface of the FISH-compatible accelerator

The system-call related registers have the prefix SC_. The PID register is written by the host program, and is later read by the FISH kernel module to determine the host program's PID. The system call ID register is written by the accelerator, and contains the unique numerical ID of the system call being requested. This register may be written multiple times, as a single accelerator may request multiple different system calls during its execution. The argument registers hold the arguments to the system call. Any pointer arguments in these registers hold physical addresses (in the CMA-allocated memory). Note that in a full-fledged implementation, there may have to be as many as seven argument registers (for a small number of Linux system calls that accept seven arguments), but only four were added in our implementation as they were sufficient for `open()`, `read()`, `close()`, and `write()`. The return value register is used by the FISH kernel module to pass on the return value of the completed system call to the accelerator. A write to this register doubles as the *finished* signal, telling the accelerator that the system call has completed.

The LegUp tool instantiates a Verilog module for each C-language function that is synthesized. Since LegUp does not support synthesis of system calls, it instantiates an empty Verilog submodule in place of the system call. It then continues to generate the rest of the accelerator circuit, including the logic for controlling the submodule and for consuming the submodule's result. This means that if the empty placeholder submodule is implemented, the rest of the accelerator works as intended. The submodules for `open()`, `read()`, `close()`, and `write()` were

implemented by hand. These submodules were made to set the accelerator's registers (mentioned above), initialize argument buffers in the CMA memory if required, and send the interrupt request to the kernel module (via the ARM Generic Interrupt Controller) to trigger the system call. As an example, the steps carried out by the submodule for `write()` are listed below:

1. Write the system call ID of `write()`, 0x4, into the System Call ID register
2. Write the arguments `fd`, `buf`, and `count` into the argument registers
3. Write the `buf` string into the buffer pointed to by the `buf` pointer
4. Send the interrupt request to the FISH kernel module
5. Wait until the return value register is written by the FISH kernel module

Due to excessive length, HDL code generated by LegUp is not included in this document.

However, the HDL code for the `open`, `write`, and `close` modules are provided in Section 9.6, 9.7, and 9.8 respectively.

5.6 Implementing the FISH Kernel Module

The FISH kernel module was implemented to handle the system calls requested by the accelerator. When the kernel module is loaded into the kernel, it registers an IRQ handler using the call to `request_threaded_irq` as shown in Figure 8. This registers a threaded interrupt handler which executes the function `irq_handler`. It is critical that the handler is threaded, as a threaded handler is interruptible. This is necessary for calling many of the system call handler functions, which have code that can only be executed from an interruptible context. For example, code that requires thread synchronization functionality (locks, semaphores) can only

work if the thread itself can be interrupted by the Linux scheduler, allowing the other threads to run.

```
request_threaded_irq(72, NULL, (irq_handler_t)irq_handler, IRQF_TRIGGER_HIGH | IRQF_ONESHOT, "irqhandler", (void *) (irq_handler));
```

Figure 8: Registering a threaded IRQ handler

An undesired side effect of using a threaded interrupt handler is that the IRQ is lowered immediately upon waking the handler thread. This is fine for certain interrupts where the required work can be queued and completed further down the line by the threaded handler. Unfortunately, this is not acceptable in our situation due to data dependencies. As computation further down the line may depend on the result of a system call, we must ensure the system call completes before lowering the IRQ. To solve this, the `IRQF_ONESHOT` flag is used, which forces the IRQ high until the threaded handler finishes.

Abbreviated code for the `irq_handler` function is shown in Figure 9. It first reads the accelerator's registers. Using the PID, the handler gets the `task_struct` object corresponding to the host program. By using the System Call ID register, it determines which system call is being requested and calls the appropriate handler function (`sys_write_handler`, `sys_read_handler`, `sys_open_handler`, or `sys_close_handler`) in the switch case block. Each system call handler function is passed the `task_struct` object, as well as the arguments. For pointer arguments, the `phys_to_virt` function is used to convert the physical address (written by the accelerator) to the kernel space virtual address. These virtual addresses can then be directly dereferenced by the handler functions to access the buffers to which they point.

```

irq_handler_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
    // Read accelerator registers
    int sc_id = ioread32(SYSCALLREG(SC_ID));
    int sc_arg0 = ioread32(SYSCALLREG(SC_ARG0));
    int sc_arg1 = ioread32(SYSCALLREG(SC_ARG1));
    int sc_arg2 = ioread32(SYSCALLREG(SC_ARG2));
    int pid = ioread32(SYSCALLREG(SC_PID));
    int syscall_result = 0;

    // Get host program's task_struct
    struct task_struct* ts = pid_task(find_vpid(pid),PIDTYPE_PID);

    switch (sc_id){
        case SC_WRITE_ID:
            syscall_result = sys_write_handler(ts, sc_arg0, (char*)phys_to_virt(sc_arg1), sc_arg2);
            break;
        case SC_READ_ID:
            syscall_result = sys_read_handler(ts, sc_arg0, (char*)phys_to_virt(sc_arg1), sc_arg2);
            break;

        case SC_OPEN_ID:
            syscall_result = sys_open_handler(ts, (char*)phys_to_virt(sc_arg0), sc_arg1, sc_arg2);
            break;
        case SC_CLOSE_ID:
            syscall_result = sys_close_handler(ts, sc_arg0);
            break;
        // Unhandled syscall
        default:
            break;
    }

    // Write the system call result into the retval register
    iowrite32(syscall_result, SYSCALLREG(SC_RETVAL));

    return (irq_handler_t) IRQ_HANDLED;
}

```

Figure 9: IRQ handler code (abbreviated)

Abbreviated code for `sys_open_handler` is shown in Figure 10. The full code for the handler is shown in Section 9.2. The host program's `task_struct` object is used to get its files table (`files_struct`). Using a modified version of the Linux function `__alloc_fd`, a file descriptor is allocated. If the provided filename argument is a full path, it is used without modification. If it is an implicit path, then it is treated as a relative path from the host program's current working directory (CWD). The CWD is retrieved using `task_struct`'s `fs` member object. Once the path is

determined, the file is opened and installed into the host program's file descriptors table using `__fd_install`.

```
int sys_open_handler(struct task_struct* ts, char* filename, int flags, int mode){

    struct files_struct* files = ts->files;
    int fd = __alloc_fd(files, 0, rlimit(RLIMIT_NOFILE), flags);
    char filename_buf[256];
    struct file * f;

    // filename must be the full path. build it relative to the host program's cwd
    if (filename[0] != '/') {
        strcpy(filename_buf, dentry_path_raw(ts->fs->pwd.dentry, filename_buf, 256));
        strcat(filename_buf, "/");
        strcat(filename_buf, filename);
    } // if the given name is a full path already, then just use it as is.
    else {
        strcpy(filename_buf, filename);
    }

    // Open the file
    f = filp_open(filename_buf, flags, mode);
    fsnotify_open(f);
    __fd_install(files, fd, f);

    return fd;
}
```

Figure 10: Abbreviated code for the open system call handler

Code for the other handlers are provided in Section 9.2.

6 Assessing the Framework Implementation

6.1 Verifying Correct Behavior

The implemented accelerator makes two calls to `open()` to open files named *input.txt* and *output.txt* within the current working directory (CWD) of the host program. It then calls `read()` to read the contents of *input.txt* then calls `write()` to store the data that was read into *output.txt*. Finally, the program calls `close()` to close the two.

This program allows us to verify key functionality. First, it tests whether an accelerator can request the allocation of a file descriptor using the call to `open()`. Second, it tests whether the accelerator shares the same “context” as the host program, which in this case was the CWD of the host program. This was verified by executing the host program from different directories, and seeing that the file was opened in the correct location. This is validation that the technique of referencing the host program’s `task_struct` from the kernel module is effective for achieving a single cross-domain hybrid application state. Third, the `read()` and `write()` verify that an accelerator can access the Linux filesystem. Indeed, after running the accelerator, the output file contained the correct text. Finally, the `close()` verifies that an accelerator can deallocate a file descriptor.

```
int fileopen(){
    int fd = open("test.txt", O_WRONLY, S_IRWXU);
    return fd;
}
```

Figure 11: An accelerator that opens a file and returns the file descriptor

To test cross-domain persistence, a second host program and accelerator (shown in Figure 11) were created. This accelerator was created to open file and return the allocated file descriptor (without writing to the file or closing it). To ensure that the open from the accelerator correctly modified the host program's state in the `task_struct` and verify that this state was visible in the software domain, the file descriptor returned by the accelerator was used by the host program to then write to the file. This worked correctly, indicating that the accelerator's `open()` system call correctly opened the file and added it to the host program's file descriptor table in its `task_struct` object.

6.2 Performance

The latencies of the software and accelerator-initiated versions of the system calls `open()`, `read()`, `close()`, and `write()` were measured. The latencies of the software-only system calls were measured using the ARM processor's timers, accessed with `gettimeofday()`. These latencies represent the best possible system call runtimes, and the FISH interface aims to achieve latencies as close to them as possible.

The latencies of the accelerator-initiated system calls were measured using the SignalTap signal analyzer to count the number of accelerator clock cycles elapsed between sending the *start* signal to the system call submodule, and receiving the *finished* signal. The number of cycles was then divided by the accelerator clock rate (50 MHz) to obtain the "HW-Initiated (Total)" times shown in Figure 12. Similarly, a cycle count was also recorded between sending the interrupt to the FISH kernel module and receiving back a response, to measure how long the kernel module

took to handle each system call. This time is shown as the “HW-Initiated (Kmodule)” time in Figure 12.

Figure 12: Latencies of Pure-SW and HW-initiated system calls

The measured latencies show that the `open()`, `read()`, `close()`, and `write()` FPGA-initiated system calls take roughly 1x-2x as long to finish compared to their software-only counterparts. The system calls that took roughly 2x longer than their software-only counterpart had one commonality, which was the need for an intermediate buffer. Intermediate buffers were required here because the Linux implementation of their handlers expected user memory space buffers for the strings, which the FPGA accelerators did not provide. As such, an intermediate

buffer was allocated by the FISH kernel module to provide the adequate memory-space buffer to the underlying system call handler.

Although the performance numbers measured cover only four system calls out of hundreds that Linux provides, we believe they indicate the neighborhood of performance that is achievable. We consider the numbers to be promising.

7 Related Works

Other works have studied FPGA-to-OS interfaces running on CPU-FPGA systems. For example, ReconOS [14] is a RTOS which provides system resources to hardware in the FPGA. Their approach uses a hardware interface in the FPGA which communicates with an interrupt handler on the CPU side, much like our work. Instead of using the interrupt handler to handle the system call request immediately, it wakes up a user-level delegate thread which is spawned for each hardware thread by ReconOS. The delegate thread then requests the system resource from the OS, on behalf of the FPGA.

HThreads [15] is another RTOS whose goal is to allow tighter interaction between software and hardware threads. A special scheduler in HThreads allows synchronization between software and hardware threads. This approach could allow delegated system call access to the FPGA by running a software thread whose sole job is to make a system call. This thread would block (using the OS's software-to-hardware synchronization capability) until the hardware thread needs the system call.

Thomas et al created the PushPush [16] linker to allow linking of software and hardware components into a single application. Using PushPush, an application of interleaved software and hardware computation can be created, meaning the software portion could make system calls. ReconOS, HThreads, and PushPush offer a system call mechanism that effectively uses a user-level thread to make the call on behalf of the FPGA. This approach is equivalent to having a host program that makes system calls on the accelerator's behalf, rather than allowing the accelerator make the call itself.

BORPH [17] is a modified Linux operating system that presents hardware interfaces in FPGA which serve as message passers to the Linux kernel. Their work focuses on providing UNIX file system access to FPGA circuits [11]. An important difference is that their interface provides direct FPGA to OS filesystem access, with no interaction between a host program and the accelerator.

8 Conclusions and Future Work

The work presented in this thesis aims to be a starting point for defining a standardized Linux to FPGA system call interface. As such, there are many small details that remains to be solved. The work instead gave ideas on the potential interface at the high level, giving insights on the general-case interactions between FPGA and Linux, and an implementation of a limited number of system calls on one of many currently available CPU-FPGA architectures. To achieve an industry-ready interface with all detailed fleshed out would require far more work. For one, it would probably require some collaboration of hardware vendors (CPU, FPGA, interconnect), Linux developers, and Linux users. Furthermore, there are many CPU-FPGA topologies to be considered with more than just those mentioned in Section 2.1. For example, IBM has a CPU-FPGA interface in their POWER8 processors called CAPI [18], and Microsoft has its Catapult system [19].

9 Appendix

9.1 Host Program Code

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

#define HW_REGS_BASE ( 0xc8000000 )
#define HW_REGS_SPAN ( 0x00200000 )
#define HW_REGS_MASK ( HW_REGS_SPAN - 1 )

unsigned int CMA_REGION_BASE;
#define CMA_REGION_SPAN ( 0x00200000 )
#define CMA_REGION_MASK ( CMA_REGION_SPAN - 1 )

int main(void)
{
    volatile unsigned int *accelerator=NULL;
    volatile unsigned int *cmabuffer=NULL;
    void *virtual_base;
    void *cma_virtual_base;
    int fd, fd2;
    FILE* cma_dev_file;
    char cma_addr_string[10];

    // Open /dev/mem
    if ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) == -1 ) {
        printf( "ERROR: could not open \"/dev/mem\"...\n" );
        return( 1 );
    }

    // Open /dev/cma_block
    if ( cma_dev_file = fopen("/dev/cma_block", "r") == NULL ) {
        printf( "ERROR: could not open \"/dev/cma_block\"...\n" );
        close( fd );
        return( 1 );
    }

    // Determine the CMA block physical address
    fgets(cma_addr_string, sizeof(cma_addr_string), cma_dev_file);
    CMA_REGION_BASE = strtoul(cma_addr_string, NULL, 16);
    printf("Use CMA memory block at phys addr: 0x%x\n",CMA_REGION_BASE);

    fclose(cma_dev_file);

    // get virtual addr that maps to bridge
    virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );
    if ( virtual_base == MAP_FAILED ) {
        printf( "ERROR: mmap() failed...\n" );
        close( fd );
        return(1);
    }

    // get virtual addr that maps to cma region
    cma_virtual_base = mmap( NULL, CMA_REGION_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, CMA_REGION_BASE );
    if ( cma_virtual_base == MAP_FAILED ) {
        printf( "ERROR: mmap() failed...\n" );
        close ( fd );
        return(1);
    }
}
```

```

    }
    // Get the address that maps to the accelerator
    accelerator=(unsigned int*)(virtual_base);
    cmabuffer=(unsigned int*)(cma_virtual_base);

    // set the pid in the accelerator
    *(accelerator + 6) = getpid();

    // start the accelerator
    *(accelerator + 2) = 1;

    // Wait until accelerator finishes (retval is set to 123)
    while(1){
        if (*accelerator == 123) break;
    }

    // check the syscall registers
    int i;
    for (i=0;i<=10;i++) printf("%d:0x%08x\n",i,*(accelerator+i));

    if( munmap( virtual_base, HW_REGS_SPAN ) != 0 ) {
        printf( "ERROR: munmap() failed...\n" );
        close( fd );
        return( 1 );
    }
    if( munmap(cma_virtual_base, HW_REGS_SPAN ) != 0 ) {
        printf( "ERROR: munmap() failed...\n" );
        close ( fd );
        return 1;
    }
    close( fd );
    close( fd2 );
    return 0;
}

```

Figure 13 C-language code for the host program that launches the accelerator

9.2 FISH Kernel Module Code

9.2.1 Main Kernel Module Code

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/cpumask.h>
#include <linux/sched.h>
#include <asm/io.h>
#include <linux/timekeeping.h>
#include <linux/irq.h>
#include <linux/cpu.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kevin Nam");
MODULE_DESCRIPTION("Interrupt Handler for Systemcall");

void * accelerator;

// Offset (in words) from the base addr of accelerator at which

```

```

// the syscall registers reside
#define SYSCALL_REG_OFFSET 3

// The system call registers
#define SC_ID 0
#define SC_STATUS 1
#define SC_RETVAL 2
#define SC_PID 3
#define SC_ARG0 4
#define SC_ARG1 5
#define SC_ARG2 6
#define SC_ARG3 7

// Accessor function to get addr of a system call register
#define SYSCALLREG(x) (accelerator + 4*SYSCALL_REG_OFFSET + 4*x)

// System Call IDs
#define SC_READ_ID 0x3
#define SC_WRITE_ID 0x4
#define SC_OPEN_ID 0x5
#define SC_CLOSE_ID 0x6
#define SC_GETPID_ID 0x14

int sys_open_handler(struct task_struct* ts, char* filename, int flags, int mode);
int sys_close_handler(struct task_struct* ts, int fd);
int sys_write_handler(struct task_struct* ts, int fd, char* buf, int count);
int sys_read_handler(struct task_struct* ts, int fd, char* buf, int count);

irq_handler_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
    // Check which system call is being requested
    int sc_id = ioread32(SYSCALLREG(SC_ID));
    int sc_arg0 = ioread32(SYSCALLREG(SC_ARG0));
    int sc_arg1 = ioread32(SYSCALLREG(SC_ARG1));
    int sc_arg2 = ioread32(SYSCALLREG(SC_ARG2));
    // int sc_arg3 = ioread32(SYSCALLREG(SC_ARG3));
    int pid = ioread32(SYSCALLREG(SC_PID));
    int syscall_result = 0x12345678;
    struct task_struct* ts = pid_task(find_vpid(pid),PIDTYPE_PID);

    switch (sc_id){

        // read(int fd, char* buf, int count)
        case SC_READ_ID:
            syscall_result = sys_read_handler(ts, sc_arg0, (char*)(phys_to_virt(sc_arg1-0x40000000)), sc_arg2);
            break;

        // write(int fd, char* buf, int count)
        case SC_WRITE_ID:
            syscall_result = sys_write_handler(ts, sc_arg0, (char*)(phys_to_virt(sc_arg1-0x40000000)), sc_arg2);
            break;

        // open(char* filename, int flags, int mode)
        case SC_OPEN_ID:
            syscall_result = sys_open_handler(ts, (char*)(phys_to_virt(sc_arg0-0x40000000)), sc_arg1,
S_IRWXU/*sc_arg2*/);
            break;

        // close(int fd)
        case SC_CLOSE_ID:
            syscall_result = sys_close_handler(ts, sc_arg0);
            break;

        // getpid()
        case SC_GETPID_ID:
            break;
    }
}

```

```

        // Unhandled syscall
        default:
            break;
    }

    // Write the system call return value to the SC_RETVAl register
    // Even system calls that have no return value must write something to this reg,
    // as a write to this register functions as an ACK
    iowrite32(syscall_result, SYSCALLREG(SC_RETVAl));

    return (irq_handler_t) IRQ_HANDLED;
}

static int __init initialize_accelerator_syscall_driver(void)
{
    // struct cpumask cpum;
    // cpumask_set_cpu(1, &cpum);

    // get the virtual addr that maps to 0xff200000
    accelerator = ioremap_nocache(0xc8000000, 0x200000);

    // write a value to the syscall_retval reg to clear any interrupt
    iowrite32(0x12345678, accelerator + 0x14);

    // Register the irq handler
    request_threaded_irq(72, NULL, //setting this to null means the default is used, which just wakes up the threaded handler.
        (irq_handler_t)irq_handler,
        IRQF_TRIGGER_HIGH | IRQF_ONESHOT, "irqhandler", (void *) (irq_handler));

    // set irq affinity (make it run on cpu1)
    // irq_set_affinity(72, &cpum);
    // irq_set_affinity(72, cpumask_of(1));

    return 0;
}

static void __exit cleanup_accelerator_syscall_driver(void)
{
    // de-register irq handler
    free_irq(72, (void *) irq_handler);

    iounmap(accelerator);
}

module_init(initialize_accelerator_syscall_driver);
module_exit(cleanup_accelerator_syscall_driver);

```

9.2.2 Open() Handler

```

#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/fs_struct.h>
#include <linux/fsnotify.h>
#include <linux/fdtable.h>

int __alloc_fd(struct files_struct *files,
    unsigned start, unsigned end, unsigned flags)
{
    unsigned int fd;

```

```

int error;
struct fdtable *fdt;

spin_lock(&files->file_lock);
repeat:
    fdt = files_fdttable(files);
    fd = start;
    if (fd < files->next_fd)
        fd = files->next_fd;

    if (fd < fdt->max_fds)
        fd = find_next_zero_bit(fdt->open_fds, fdt->max_fds, fd);

    /*
     * N.B. For clone tasks sharing a files structure, this test
     * will limit the total number of files that can be opened.
     */
    error = -EMFILE;
    if (fd >= end)
        goto out;

    // SKIP EXPAND_FILES FOR NOW. THIS IS DANGEROUS IF FD GETS HIGH
    // error = expand_files(files, fd);
    // if (error < 0)
    //     goto out;
    //     error = 0;

    /*
     * If we needed to expand the fs array we
     * might have blocked - try again.
     */
    if (error)
        goto repeat;

    if (start <= files->next_fd)
        files->next_fd = fd + 1;

    __set_bit(fd, fdt->open_fds);
    if (flags & O_CLOEXEC)
        __set_bit(fd, fdt->close_on_exec);
    else
        __clear_bit(fd, fdt->close_on_exec);
    error = fd;

    /* Sanity check */
    if (rcu_access_pointer(fdt->fd[fd]) != NULL) {
        printk(KERN_ERR "alloc_fd: slot %d not NULL!\n", fd);
        rcu_assign_pointer(fdt->fd[fd], NULL);
    }

out:
    spin_unlock(&files->file_lock);
    return error;
}

void __fd_install(struct files_struct *files, unsigned int fd,
                 struct file *file)
{
    struct fdtable *fdt;
    spin_lock(&files->file_lock);
    fdt = files_fdttable(files);
    BUG_ON(fdt->fd[fd] != NULL);
    rcu_assign_pointer(fdt->fd[fd], file);
    spin_unlock(&files->file_lock);
}

```

```

int sys_open_handler(struct task_struct* ts, char* filename, int flags, int mode){

    // get the fdt of task_struct.
    // get_files_struct does a lock before getting files.
    // struct files_struct * files = get_files_struct(ts);
    struct files_struct* files = ts->files;
    int fd = __alloc_fd(files, 0, rlimit(RLIMIT_NOFILE), flags);
    char filename_buf[256];
    struct file * f;

    // filename must be the full path. build it.
    if (filename[0] != '/'){
        strcpy(filename_buf, dentry_path_raw(ts->fs->pwd.dentry, filename_buf, 256));
        strcat(filename_buf, "/");
        strcat(filename_buf, filename);
        printk(KERN_ERR "Opening: %s\n", filename_buf);
    // if the given name is a full path already, then just use it as is.
    } else {
        strcpy(filename_buf, filename);
    }

    // Open file
    f = filp_open(filename_buf, flags, mode);

    // notify
    fsnotify_open(f);

    // install the fd
    __fd_install(files, fd, f);

    return fd;
}

```

9.2.3 Read() Handler

```

#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/fs_struct.h>
#include <linux/fdtable.h>
#include <linux/file.h>

void strncpy_to_volatile_mem(char* src, volatile char* dest, int count){
    int i;
    for (i=0; i < count; i++){
        dest[i] = src[i];
    }
}

int sys_read_handler(struct task_struct* ts, int fd, char* buf, int count){

    ssize_t ret = -EBADF;
    struct file* f = ts->files->fdt->fd[fd];
    loff_t pos;

    char* tempbuf = kmalloc(sizeof(char)*count, GFP_KERNEL);

    if (f) {
        pos = f->f_pos;
        ret = kernel_read(f, pos, tempbuf, count);
        if (ret >= 0)

```



```

        f->f_pos = pos;
    } else {
        printk(KERN_ERR "ERROR: f == NULL. NOT WRITING ANYTHING!\n");
    }

    strncpy_to_volatile_mem(tempbuf, (volatile char*)buf, count);

    kfree(tempbuf);

    return ret;
}

```

9.2.4 Write() Handler

```

#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/fs_struct.h>
#include <linux/fdtable.h>
#include <linux/file.h>

void strncpy_from_volatile_mem(char* dest, volatile char* src, int count){
    int i;
    for (i=0; i < count; i++){
        dest[i] = src[i];
    }
}

int sys_write_handler(struct task_struct* ts, int fd, char* buf, int count){

    ssize_t ret = -EBADF;
    struct file* f = ts->files->fdt->fd[fd];
    loff_t pos;

    char* tempbuf = kmalloc(sizeof(char)*count, GFP_KERNEL);
    strncpy_from_volatile_mem(tempbuf, (volatile char*)buf, count);

    if (f) {
        pos = f->f_pos;
        ret = kernel_write(f, tempbuf, count, pos);
        if (ret >= 0)
            f->f_pos = pos;
    } else {
        printk(KERN_ERR "ERROR: f == NULL. NOT WRITING ANYTHING!\n");
    }

    kfree(tempbuf);

    return ret;
}

```

9.2.5 Close() Handler

```

#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/fs_struct.h>

```

```

#include <linux/fdtable.h>
#include <linux/path.h>
#include <linux/file.h>

static void __put_unused_fd(struct files_struct *files, unsigned int fd)
{
    struct fdtable *fdt = files_fdtable(files);
    __clear_bit(fd, fdt->open_fds); // __clear_open_fd(fd, fdt);
    if (fd < files->next_fd)
        files->next_fd = fd;
}

int __close_fd(struct files_struct *files, unsigned fd)
{
    struct file *file;
    struct fdtable *fdt;

    spin_lock(&files->file_lock);
    fdt = files_fdtable(files);
    if (fd >= fdt->max_fds)
        goto out_unlock;
    file = fdt->fd[fd];
    if (!file)
        goto out_unlock;
    rcu_assign_pointer(fdt->fd[fd], NULL);
    __clear_bit(fd, fdt->close_on_exec); // __clear_close_on_exec(fd, fdt);
    __put_unused_fd(files, fd);
    spin_unlock(&files->file_lock);
    return filp_close(file, files);

out_unlock:
    spin_unlock(&files->file_lock);
    return -EBADF;
}

int sys_close_handler(struct task_struct* ts, int fd){

    struct files_struct* files = ts->files;
    /*int ret = */__close_fd(files, fd);

    // Supposed to return 0 if success, -1 if fail, and set appropriate ERRNO.
    return 0;
}

EXPORT_SYMBOL(sys_close_handler);

```

9.3 CMA Driver Kernel Module Code

This kernel module allocates physically contiguous memory for the accelerator to use. The physical address is printed through the character device interface so that the user program (the host program) can determine the physical address of the contiguous buffer.

```

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/dma-mapping.h>
#include <linux/device.h>
#include <asm/io.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("kevin nam");
MODULE_DESCRIPTION("cma memory allocator");

void * lwbridgebase;
// CMA block allocation vars
static dma_addr_t dma_mem_base_paddr;
static void* dma_mem_base_vaddr;
static size_t size = 16384*PAGE_SIZE; // Allocate 64MB

// Character device objects
static dev_t dev_no = 0;
static struct cdev* CMA_cdev = NULL;
static struct class* CMA_class = NULL;
#define DEVICE_NAME "cma_block"

// Character device function declarations
static int cdev_open(struct inode *, struct file *);
static int cdev_release(struct inode *, struct file *);
static ssize_t cdev_read(struct file *, char *, size_t, loff_t *);

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = cdev_read,
    .open = cdev_open,
    .release = cdev_release
};

int initialize_char_dev(void){

    int err = 0;

    /* Get a device number. Get one minor number (0) */
    err = alloc_chrdev_region(&dev_no, 0, 1, DEVICE_NAME);
    if (err < 0) {
        printk(KERN_WARNING "cma driver: alloc_chrdev_region() failed\n");
        return err;
    }

    CMA_class = class_create(THIS_MODULE, DEVICE_NAME);

    // Allocate and initialize the char device
    CMA_cdev = cdev_alloc();
    CMA_cdev->ops = &fops;
    CMA_cdev->owner = THIS_MODULE;

    // Add the character device to the kernel
    err = cdev_add( CMA_cdev,dev_no,1);
    if (err < 0) {
        printk(KERN_WARNING "cma driver: cdev_add() failed\n");
        return err;
    }

    device_create(CMA_class, NULL, dev_no, NULL, DEVICE_NAME );

```

```

    return err;
}

void cleanup_char_dev(void){
    // Clean up the character device objects
    device_destroy(CMA_class, dev_no);
    cdev_del(CMA_cdev);
    class_destroy(CMA_class);
    unregister_chrdev_region(dev_no, 1);
}

static int __init initialize_cma_block(void)
{
    initialize_char_dev();

    // Allocate contiguous memory
    dma_mem_base_vaddr = dma_alloc_coherent(NULL, size, &dma_mem_base_paddr, GFP_KERNEL);
    printk("Contiguous block allocated!\n");
    printk("vaddr: %x\npaddr: %x\n", (unsigned)dma_mem_base_vaddr, (unsigned)dma_mem_base_paddr);

    return 0;
}

static void __exit cleanup_cma_block(void)
{
    cleanup_char_dev();

    // De-allocate DMA mem
    dma_free_coherent(NULL, size, dma_mem_base_vaddr, dma_mem_base_paddr);

    printk("Contiguous block deallocated!\n");
}

static int cdev_open(struct inode *inode, struct file *file)
{
    return 0;
}

static int cdev_release(struct inode *inode, struct file *file)
{
    return 0;
}

static ssize_t cdev_read(struct file *filp, char *buffer, size_t length, loff_t *offset)
{
    int bytes_read = 0;

    char themessage[256];
    char *msg_Ptr = &(themessage[*offset]);

    sprintf(themessage, "%x\n", (unsigned)dma_mem_base_paddr);

    // Write to user buffer
    while (length && *msg_Ptr) {
        put_user(*(msg_Ptr++), buffer++);
        length--;
        bytes_read++;
    }
    (*offset)+= bytes_read;
    return bytes_read;
}

module_init(initialize_cma_block);
module_exit(cleanup_cma_block);

```

Figure 14 Kernel module code for allocating physically contiguous memory using CMA

9.4 LegUp C-Language Code for Accelerator

```
// Accelerate this function
int fwrite_wrapper(){
    int fd = 1234567890;

    fd = open("test.txt", O_WRONLY);

    write(fd, "Hello! The accelerator wrote this file!\n", 40);

    close(fd);

    return 123;
}
```

Figure 15 Accelerator code (in C) that was synthesized using LegUP

9.5 LegUp-Synthesized Accelerator Code in Verilog HDL

Omitted due to length. Contact author (kevin.nam at utoronto ca) for the code.

Figure 16 Accelerator code (in Verilog HDL) that was generated by LegUP

9.6 Verilog HDL Module that Initiates the Open System Call

```
module open (
    clk,
    clk2x,
    clk1x_follower,
    reset,
    start,
    finish,
    return_val,
    arg_0,
    arg_1,
    memory_controller_waitrequest,
    memory_controller_enable_a,
    memory_controller_address_a,
    memory_controller_write_enable_a,
    memory_controller_in_a,
    memory_controller_size_a,
    memory_controller_out_a,
    syscall_retval,
    syscall_id,
    syscall_start,
    syscall_finished,
    //syscall_pid,
    syscall_arg0,
    syscall_arg1,
    syscall_arg2,
    syscall_arg3
);
    parameter [3:0] STATE_reset = 4'd0, STATE_setup = 4'd1, STATE_started = 4'd2, STATE_finished = 4'd3, STATE_readpending = 4'd4, STATE_write = 4'd5, STATE_writepending = 4'd6, STATE_readpending_2 = 4'd7, STATE_pending = 4'd8;

    // The number of characters that must be read: 9;
    // this is known at compile time!!! HLS can fill it in.
    // otherwise have a circuit to check if char is '\0' and keep going until it is.
```

```

parameter [7:0] length = 8'd9; //

input clk;
input clk2x;
input clk1x_follower;
input reset;
input start;
output reg finish;
output reg [31:0] return_val;
input [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] arg_0;
input [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] arg_1;
input memory_controller_waitrequest;
output reg memory_controller_enable_a;
output reg [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] memory_controller_address_a;
output reg memory_controller_write_enable_a;
output reg [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_in_a;
output reg [1:0] memory_controller_size_a;
input [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_out_a;
input [31:0] syscall_retval;
output reg [8:0] syscall_id;
output reg syscall_start;
input syscall_finished;
//output reg [31:0] syscall_pid;
output [31:0] syscall_arg0;
output reg [31:0] syscall_arg1;
output reg [31:0] syscall_arg2;
output reg [31:0] syscall_arg3;

reg [3:0] cur_state;
reg [3:0] next_state;

wire setup_finished;

assign setup_finished = read_buffer_cnt >= length;

always@(posedge clk)
begin
    if (reset)
        cur_state <= STATE_reset;
    else if (memory_controller_waitrequest)
        cur_state <= cur_state;
    else
        cur_state <= next_state;
end

always@(*)
begin
    next_state = cur_state;
    case(cur_state)
        STATE_reset:
            if (start)
                next_state = STATE_setup;
            // Setup state -> read -> write loop until the input is written to a buffer. Idea is to
            // give the address of the buffer to the kernel module, which reads the buffer to determine
            // the filename and opens it. BUT if bit30 of arg_0 is 1, that means the input bufer
            // is already in DDR SDRAM. If already in DDR, simply hand off address to kernel module.
            // This assumes that the caller provides PHYSICAL ADDRESS! This would be the case if
            // the SW was compiled using AISLE or is modified to do virt to phys conversion before calling
            // the accelerator.
        STATE_setup:
            if (setup_finished == 1'b1 || arg_0[30] == 1'b1)
                next_state = STATE_started;
            else
                next_state = STATE_readpending;
        STATE_readpending:
            next_state = STATE_readpending_2;
    endcase
end

```

```

        STATE_readpending_2:
            next_state = STATE_write;
        STATE_write:
            next_state = STATE_writepending;
        STATE_writepending:
            next_state = STATE_setup;
        STATE_started:
            next_state = STATE_pending;
        STATE_pending:
            if (systemcall_finished)
                next_state = STATE_finished;
        STATE_finished:
            next_state = STATE_reset;
        default:
            next_state = cur_state;
    endcase
end

always@(posedge clk)
begin
    if (cur_state == STATE_reset)
        systemcall_id <= 9'd111;
    else
        systemcall_id <= 9'h5; // the ID of open is 0x5
end

always@(posedge clk)
begin
    if (cur_state == STATE_started)
        systemcall_start <= 1'b1;
    else
        systemcall_start <= 1'b0;
end

always@(posedge clk)
begin
    if (reset)
        return_val <= 32'd0;
    else if (systemcall_finished)
        return_val <= systemcall_retval;
end

always@(posedge clk)
begin
    if (cur_state == STATE_finished)
        finish <= 1'b1;
    else
        finish <= 1'b0;
end

reg [7:0] read_buffer_cnt;
reg [7:0] write_buffer_cnt;
reg [7:0] read_char;
reg [31:0] debug_reg;

always@(posedge clk)
begin
    if (reset)
        read_buffer_cnt <= 8'd0;
    else if (cur_state == STATE_setup)
        read_buffer_cnt <= read_buffer_cnt + 1;
    else
        read_buffer_cnt <= read_buffer_cnt;
end

always@(posedge clk)
begin

```

```

        if (reset)
            write_buffer_cnt <= 8'd0;
        else if (cur_state == STATE_write)
            write_buffer_cnt <= write_buffer_cnt + 1;
        else
            write_buffer_cnt <= write_buffer_cnt;
    end
    always@(posedge clk)
    begin
        if (reset)
            read_char <= 8'd0;
        else if (cur_state == STATE_readpending_2)
            read_char <= memory_controller_out_a[7:0];
        else
            read_char <= read_char;
    end
    always@(posedge clk)
    begin
        if (reset)
            debug_reg <= 32'd0;
        else if (cur_state == STATE_write)
            debug_reg <= debug_reg + {32'd0,read_char};
    end

    end

    always@(*)
    begin
        memory_controller_enable_a = 1'd0;
        memory_controller_address_a = 32'd0;
        memory_controller_write_enable_a = 1'd0;
        memory_controller_in_a = 1'd0;
        memory_controller_size_a = 2'd0;

        // Do the transfer from buffer to buffer.
        // The number of characters that must be read: str.numwords_a = 9;
        // this is known at compile time!!! HLS can fill it in.
        // otherwise have a circuit to check if char is '\0' and keep going until it is.
        if (cur_state == STATE_setup && read_buffer_cnt < length) begin
            memory_controller_enable_a = 1'd1;
            memory_controller_address_a = arg_0 + read_buffer_cnt;
            memory_controller_size_a = 2'd0;
        end
        else if (cur_state == STATE_write) begin
            memory_controller_enable_a = 1'd1;
            memory_controller_write_enable_a = 1'd1;
            memory_controller_address_a = (`systemcall_buffer_in_cma_0 + ({24'd0,write_buffer_cnt}*32'd1));
            memory_controller_size_a = 2'd0;
            memory_controller_in_a = read_char;
        end
    end

    // output reg memory_controller_enable_a;
    // output reg [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] memory_controller_address_a;
    // output reg memory_controller_write_enable_a;
    // output reg [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_in_a;
    // output reg [1:0] memory_controller_size_a;

    end

    // setup arg0
    // let's assume that the HLS tool will allocate the appropriate amount of space
    // at some fixed address within the CMA region. CMA region: 0x1f800000 - 0x2f800000
    // write to buffer addr: `systemcall_buffer_in_cma_0
    // read from buffer addr: arg_0
    // If arg_0 is already an address in shared memory (DDR3) then simply provide the address
    // to the kernel module. ASSUMING ITS A PHYS ADDR! onus is on SW to provide phys addr.
    assign systemcall_arg0 = arg_0[30] == 1'b1 ? arg_0 : `systemcall_buffer_in_cma_0;

    // setup arg1

```



```

always@(posedge clk)
begin
    systemcall_arg1 <= arg_1;
end

// setup arg2
always@(posedge clk)
begin
    systemcall_arg2 <= debug_reg;
end

// setup arg3
always@(posedge clk)
begin
    systemcall_arg3 <= {8'd0,read_char,write_buffer_cnt,read_buffer_cnt}; // FOR DEBUGGING USE AS AN OUTPUT
end

systemcall_arg3 <= 32'd0;
end

endmodule

```

Figure 17 Verilog module that initiates the open() system call from the accelerator

9.7 Verilog HDL Module that Initiates the Write System Call

```

module write (
    clk,
    clk2x,
    clk1x_follower,
    reset,
    start,
    finish,
    return_val,
    arg_0,
    arg_1,
    arg_2,
    memory_controller_waitrequest,
    memory_controller_enable_a,
    memory_controller_address_a,
    memory_controller_write_enable_a,
    memory_controller_in_a,
    memory_controller_size_a,
    memory_controller_out_a,
    systemcall_retval,
    systemcall_id,
    systemcall_start,
    systemcall_finished,
    //systemcall_pid,
    systemcall_arg0,
    systemcall_arg1,
    systemcall_arg2,
    systemcall_arg3
);
    parameter [3:0] STATE_reset = 4'd0, STATE_setup = 4'd1, STATE_started = 4'd2, STATE_finished = 4'd3, STATE_readpending = 4'd4, STATE_write = 4'd5, STATE_writepending = 4'd6, STATE_readpending_2 = 4'd7, STATE_pending = 4'd8;

    input clk;
    input clk2x;
    input clk1x_follower;
    input reset;
    input start;
    output reg finish;
    output reg [31:0] return_val;
    input ['MEMORY_CONTROLLER_ADDR_SIZE-1:0] arg_0;

```

```

input [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] arg_1;
input [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] arg_2;
input memory_controller_waitrequest;
output reg memory_controller_enable_a;
output reg [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] memory_controller_address_a;
output reg memory_controller_write_enable_a;
output reg [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_in_a;
output reg [1:0] memory_controller_size_a;
input [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_out_a;
input [31:0] syscall_retval;
output reg [8:0] syscall_id;
output reg syscall_start;
input syscall_finished;
//output reg [31:0] syscall_pid;
output [31:0] syscall_arg0;
output [31:0] syscall_arg1;
output [31:0] syscall_arg2;
output reg [31:0] syscall_arg3;

reg [3:0] cur_state;
reg [3:0] next_state;

wire setup_finished;

// The number of characters that must be read is arg_2
assign setup_finished = read_buffer_cnt >= arg_2;

always@(posedge clk)
begin
    if (reset)
        cur_state <= STATE_reset;
    else if (memory_controller_waitrequest)
        cur_state <= cur_state;
    else
        cur_state <= next_state;
end

always@(*)
begin
    next_state = cur_state;
    case(cur_state)
        STATE_reset:
            if (start)
                next_state = STATE_setup;
            // Setup state -> read -> write loop until the input is written to a buffer. Idea is to
            // give the address of the buffer to the kernel module, which reads the buffer to determine
            // the filename and opens it. BUT if bit30 of arg_1 is 1, that means the input bufer
            // is already in DDR SDRAM. If already in DDR, simply hand off address to kernel module.
            // This assumes that the caller provides PHYSICAL ADDRESS! This would be the case if
            // the SW was compiled using AISLE or is modified to do virt to phys conversion before calling
            // the accelerator.
            STATE_setup:
                if (setup_finished == 1'b1 || arg_1[30] == 1'b1)
                    next_state = STATE_started;
                else
                    next_state = STATE_readpending;
            STATE_readpending:
                next_state = STATE_readpending_2;
            STATE_readpending_2:
                next_state = STATE_write;
            STATE_write:
                next_state = STATE_writepending;
            STATE_writepending:
                next_state = STATE_setup;
            STATE_started:
                next_state = STATE_pending;
    endcase
end

```

```

        STATE_pending:
            if (systemcall_finished)
                next_state = STATE_finished;
        STATE_finished:
            next_state = STATE_reset;
        default:
            next_state = cur_state;
    endcase
end

always@(posedge clk)
begin
    if (cur_state == STATE_reset)
        systemcall_id <= 9'd111;
    else
        systemcall_id <= 9'h4; // the ID of write is 0x4
end

always@(posedge clk)
begin
    if (cur_state == STATE_started)
        systemcall_start <= 1'b1;
    else
        systemcall_start <= 1'b0;
end

always@(posedge clk)
begin
    if (reset)
        return_val <= 32'd0;
    else if (systemcall_finished)
        return_val <= systemcall_retval;
end

always@(posedge clk)
begin
    if (cur_state == STATE_finished)
        finish <= 1'b1;
    else
        finish <= 1'b0;
end

reg [31:0] read_buffer_cnt;
reg [31:0] write_buffer_cnt;
reg [7:0] read_char;
reg [31:0] debug_reg;

always@(posedge clk)
begin
    if (reset)
        read_buffer_cnt <= 8'd0;
    else if (cur_state == STATE_setup)
        read_buffer_cnt <= read_buffer_cnt + 1;
    else
        read_buffer_cnt <= read_buffer_cnt;
end

always@(posedge clk)
begin
    if (reset)
        write_buffer_cnt <= 8'd0;
    else if (cur_state == STATE_write)
        write_buffer_cnt <= write_buffer_cnt + 1;
    else
        write_buffer_cnt <= write_buffer_cnt;
end

always@(posedge clk)

```

```

begin
    if (reset)
        read_char <= 8'd0;
    else if (cur_state == STATE_readpending_2)
        read_char <= memory_controller_out_a[7:0];
    else
        read_char <= read_char;
end
always@(posedge clk)
begin
    if (reset)
        debug_reg <= 32'd0;
    else if (cur_state == STATE_write)
        debug_reg <= debug_reg + {32'd0,read_char};
end

always@(*)
begin
    memory_controller_enable_a = 1'd0;
    memory_controller_address_a = 32'd0;
    memory_controller_write_enable_a = 1'd0;
    memory_controller_in_a = 1'd0;
    memory_controller_size_a = 2'd0;

    // Do the transfer from buffer to buffer.
    // The number of characters that must be read: str.numwords_a = 9;
    // this is known at compile time!!! HLS can fill it in.
    // otherwise have a circuit to check if char is '\0' and keep going until it is.
    if (cur_state == STATE_setup && read_buffer_cnt < arg_2) begin
        memory_controller_enable_a = 1'd1;
        memory_controller_address_a = arg_1 + read_buffer_cnt;
        memory_controller_size_a = 2'd0;
    end
    else if (cur_state == STATE_write) begin
        memory_controller_enable_a = 1'd1;
        memory_controller_write_enable_a = 1'd1;
        memory_controller_address_a = (`systemcall_buffer_in_cma_0 + ({24'd0,write_buffer_cnt}*32'd1));
        memory_controller_size_a = 2'd0;
        memory_controller_in_a = read_char;
    end
end

// setup arg0
assign systemcall_arg0 = arg_0;

// setup arg1
// let's assume that the HLS tool will allocate the appropriate amount of space
// at some fixed address within the CMA region. CMA region: 0x1f800000 - 0x2f800000
// if the input buffer is already in shared memory (bit30 is 1) then just give arg_1.
// otherwise we move the buffer into CMA in the setup stage, and we pass the cma buffer addr.
assign systemcall_arg1 = arg_1[30] ? arg_1 : `systemcall_buffer_in_cma_0;

// setup arg2
assign systemcall_arg2 = arg_2;

// setup arg3
always@(posedge clk)
begin
    systemcall_arg3 <= 32'd0; // {8'd0,read_char,write_buffer_cnt,read_buffer_cnt}; // FOR DEBUGGING USE AS AN
    OUTPUT systemcall_arg3 <= 32'd0;
end
endmodule

```

Figure 18 Verilog module that initiates the write() system call from the accelerator

9.8 Verilog HDL Module that Initiates the Close System Call

```
module close (
    clk,
    clk2x,
    clk1x_follower,
    reset,
    start,
    finish,
    return_val,
    arg_0,
    memory_controller_waitrequest,
    memory_controller_enable_a,
    memory_controller_address_a,
    memory_controller_write_enable_a,
    memory_controller_in_a,
    memory_controller_size_a,
    memory_controller_out_a,
    syscall_retval,
    syscall_id,
    syscall_start,
    syscall_finished,
    //syscall_pid,
    syscall_arg0,
    syscall_arg1,
    syscall_arg2,
    syscall_arg3
);

    parameter [2:0] STATE_reset = 3'd0, STATE_started = 3'd2, STATE_finished = 3'd3, STATE_pending = 3'd4;

    input clk;
    input clk2x;
    input clk1x_follower;
    input reset;
    input start;
    output reg finish;
    output reg [31:0] return_val;
    input [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] arg_0;
    input memory_controller_waitrequest;
    output reg memory_controller_enable_a;
    output reg [`MEMORY_CONTROLLER_ADDR_SIZE-1:0] memory_controller_address_a;
    output reg memory_controller_write_enable_a;
    output reg [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_in_a;
    output reg [1:0] memory_controller_size_a;
    input [`MEMORY_CONTROLLER_DATA_SIZE-1:0] memory_controller_out_a;
    input [31:0] syscall_retval;
    output reg [8:0] syscall_id;
    output reg syscall_start;
    input syscall_finished;
    //output reg [31:0] syscall_pid;
    output [31:0] syscall_arg0;
    output [31:0] syscall_arg1;
    output [31:0] syscall_arg2;
    output [31:0] syscall_arg3;

    reg [2:0] cur_state;
    reg [2:0] next_state;

    always@(posedge clk)
    begin
        if (reset)
            cur_state <= STATE_reset;
        else
            cur_state <= next_state;
    end
endmodule
```

```

end

always@(*)
begin
    next_state = cur_state;
    case(cur_state)
        STATE_reset:
            if (start)
                next_state = STATE_started;
            STATE_started:
                next_state = STATE_pending;
            STATE_pending:
                if (systemcall_finished)
                    next_state = STATE_finished;
            STATE_finished:
                next_state = STATE_reset;
            default:
                next_state = cur_state;
    endcase
end

always@(posedge clk)
begin
    if (cur_state == STATE_reset)
        systemcall_id <= 9'd111;
    else
        systemcall_id <= 9'h6; // the ID of close is 0x6
end

always@(posedge clk)
begin
    if (cur_state == STATE_started)
        systemcall_start <= 1'b1;
    else
        systemcall_start <= 1'b0;
end

always@(posedge clk)
begin
    if (reset)
        return_val <= 32'd0;
    else if (systemcall_finished)
        return_val <= systemcall_retval;
end

always@(posedge clk)
begin
    if (cur_state == STATE_finished)
        finish <= 1'b1;
    else
        finish <= 1'b0;
end

always@(*)
begin
    memory_controller_enable_a = 1'd0;
    memory_controller_address_a = 32'd0;
    memory_controller_write_enable_a = 1'd0;
    memory_controller_in_a = 1'd0;
    memory_controller_size_a = 2'd0;
end

// setup arg0
assign systemcall_arg0 = arg_0;

```

```
// setup arg1
assign syscall_arg1 = 32'd0;

// setup arg2
assign syscall_arg2 = 32'd0;

// setup arg3
assign syscall_arg3 = 32'd0;

endmodule
```

Figure 19 Verilog module that initiates the close() system call from the accelerator

9.9 Linux Kernel Config File

Omitted due to length. Contact author (kevin.nam at utoronto.ca) for the code.

Figure 20 Linux kernel configuration settings for the custom 3.18 kernel used in the DE1-SoC

Linux distribution

10 References

- [1] I. Corporation, "Intel SoCs: When Architecture Matters," [Online]. Available:
<https://www.altera.com/products/soc/overview.html>.
- [2] X. Inc, "Expanding the All Programmable SoC Portfolio," [Online]. Available:
<https://www.xilinx.com/products/silicon-devices/soc.html>.
- [3] P. Gupta, "Xeon+FPGA Platform for the Data Center," 2015. [Online]. Available:
<https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf>.
- [4] I. Corporation, "SDK for OpenCL," [Online]. Available:
<https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html>.
- [5] Xilinx Inc, "Vivado High-Level Synthesis," 2017. [Online]. Available:
<https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>.
- [6] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, Monterey, 2011.
- [7] T. Team, "Intel Works To Gain An Upper Hand Over Nvidia In The Data Center Coprocessor Market," Forbes, 2016. [Online]. Available:

<https://www.forbes.com/sites/greatspeculations/2016/12/16/intel-works-to-gain-an-upper-hand-over-nvidia-in-the-data-center-coprocessor-market/#5c268d9b1178>.

- [8] Nallatech, "Choose the right FPGA Card for your Application," 2017. [Online]. Available: <http://www.nallatech.com/solutions/fpga-cards/>. [Accessed 5 May 2017].
- [9] Terasic Technologies, "DE Main Boards," 2017. [Online]. Available: <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=163>. [Accessed 5 May 2017].
- [10] M. Kerrisk, "syscalls - Linux system calls," [Online]. Available: <http://man7.org/linux/man-pages/man2/syscalls.2.html>.
- [11] H. K.-H. So and R. Brodersen, "File system access from reconfigurable FPGA hardware processes in BORPH," in *FPL*, Heidelberg, 2008.
- [12] Terasic Technologies Inc, "DE1-SoC User Manual," 2014.
- [13] I. Corporation, "Qsys - System Integration Tool," [Online]. Available: <https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/qts-qsys.html>. [Accessed 19 3 2017].
- [14] E. Lubbers and M. Platzner, "ReconOS: An RTOS Supporting Hard-and Software Threads," in *FPL*, Amsterdam, 2007.

- [15] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot and J. Stevens, "Run-Time Services for Hybrid CPU/FPGA Systems on Chip," in *IEEE International Real-Time Systems Symposium*, Rio de Janeiro, 2006.
- [16] D. B. Thomas, S. T. Fleming, G. A. Constantinides and D. R. Ghica, "Transparent linking of compiled software and synthesized hardware," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, 2015.
- [17] H. K.-H. So, "BORPH: An Operating System for FPGA-Based Reconfigurable," EECS, University of California, Berkeley, 2007.
- [18] IBM, "POWER8 Coherent Accelerator Processor Interface (CAPI)," [Online]. Available: <https://www-304.ibm.com/webapp/set2/sas/f/capi/home.html>.
- [19] Microsoft, "Project Catapult," [Online]. Available: <https://www.microsoft.com/en-us/research/project/project-catapult/>.
- [20] A. E. Hindborg, P. Schleuniger, N. B. Jensen and S. Karlsson, "Hardware Realization of an FPGA Processor – Operating System Call Offload and," in *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Madrid, 2014.