FUSE: Front-end user framework for O/S abstraction of hardware accelerators

Aws Ismail
School of Engineering Science
Simon Fraser University
Burnaby, Canada
Email: aii@sfu.ca

Lesley Shannon
School of Engineering Science
Simon Fraser University
Burnaby, Canada
Email: lshannon@ensc.sfu.ca

***Abstract*—SoCs can be implemented on a single FPGA, offering designers a unique opportunity for Embedded Systems. Instead of defining a fixed architecture early in the design process, the reconfigurable platform allows architectural redesign to meet the system's specific needs. However, the ability to instantiate new modules in the reconfigurable hardware provides a unique set of challenges for integration, particularly to the software (SW) designer. Specifically, the Operating System (OS) cannot automatically abstract these platform changes without redesign.**

In this paper, we present FUSE, a framework for HW accelerator abstraction that provides: 1) transparency to the SW designer at the application level; and 2) OS support for easy HW accelerator integration. We illustrate FUSE as an API for an embedded Linux OS with POSIX threads on Xilinx's MicroBlaze on a Virtex5. For three different applications and HW accelerators, we achieve performance speedups ranging from 6.4-37x.

I. Introduction

With the increasing density of Field Programmable Gate Arrays (FPGAs), they are able to implement Systems-on-Chip (SoCs) for embedded computing platforms. Thus, FPGA companies provide soft processors that can be integrated with custom HW accelerators on the same device. To complement the increasing complexity of SoCs implemented on FPGAs, improved software (SW) systems support for embedded computing applications is needed.

Therefore, Operating Systems (OS) are becoming increasingly common as they enable programming model abstractions that simplify SW development by hiding the low-level details of HW peripherals from the SW designer [1]. By extending OS support to include HW accelerators, the low-level HW interaction details can also be masked, facilitating the development of applications that use HW accelerators. Traditionally, this requires redesigning and recompiling the OS kernel for each new HW accelerator [2]. Instead, we propose customizing the OS at runtime to support existing HW accelerators as additional computing resources for SW designers, allowing the OS to automatically schedule the application(s) to leverage them.

In this paper, we present a Front-end USEr framework, *FUSE*, which abstracts embedded computing architectures away from SW designers and their applications. This is crucial for systems implemented on FPGAs, so HW designers can create and update HW accelerators to suit an application/user's changing requirements, independent of the SW designer. HW accelerators are virtualized from SW designers as *hardware tasks* (HW tasks) similar to [3] [4] in the context of a multithreading application. FUSE combines this concept with a modular approach to provide a customizable data/control communication interface between HW accelerators and their OS kernel support to enable *on-demand instantiation* of accelerators, similar to a SW dynamically linked library (DLL).

The designer's chosen HW accelerator interface (discussed in Section IV) is supported by the OS kernel through a loadable kernel module (LKM) (see Section III). The LKM acts as the low-level SW abstraction of the HW accelerator interface, thus providing the communication link across the HW/SW boundary in the system. By using a layered, modular structure for the FUSE framework design, we achieve the desired separation between the user SW and the computing platform. As such, updates made to a HW accelerator's design can only result in changes to its interface and its corresponding LKM, and will not affect the user application and user-space part of FUSE.

To demonstrate FUSE, we have designed an API based on the POSIX thread standard and integrated it with the PetaLinux OS [5] as a user library for a Microblaze CPU. We use a Xilinx V5 ML505 for three case studies. We show performance speedups ranging from 6.4-37x, excluding the overhead of loading LKMs at runtime.

The remainder of this paper is organized as follows. Section II provides an overview of related work on OS support for systems with HW accelerators on FPGAs. Section III introduces the FUSE framework and its components. Section IV describes the implementation of FUSE's components and its support for HW accelerator virtualization. Section V demonstrates the use of FUSE, along with its overhead and performance speedups for a few case studies. Finally, Section VI concludes the paper and outlines future work.

II. Background

For an OS to provide efficient abstraction of communication between SW and HW tasks, it requires mechanisms that allow user applications to access low-level HW in a transparent and safe manner [6]. The remainder of this section discusses the related work and contrasts our contributions within this context.

A. Related research into accelerator/OS integration

Researchers have used several methods of encapsulating HW resources in order to provide abstraction for reconfigurable computing. For example, the BORPH project [7] abstracts HW accelerators as UNIX processes that have access to OS services and communicate using First-in-First-out (FIFO) buffers. Similarly, Kociuszkiewicz *et al.* [8] model HW tasks as drop-in replacements for SW tasks by mapping them to synthesized coarse-grained processor cores, which also communicate via FIFOs.

Extending the thread programming model to abstract HW accelerators from user applications requires support for synchronized communication between HW and SW tasks. The ReconOS project [3] introduced an execution environment that extends the POSIX multithreaded programming model from the SW domain to reconfigurable hardware. Also, the “HybridThreads” project [2] focused on implementing the synchronization primitives provided by the POSIX multithreaded programming model (e.g. semaphores, mutexes, etc.) as dedicated HW cores. “HybridThreads” also presented a tool for generating sequential HW threads directly from SW code. Finally, Compton et. al. [9] introduced a configurable HW interface for HW tasks. The interface uses memory-mapped I/O to communicate with its accelerator, with each SW application having access to only its own set of accelerators.

Additionally, extensive research has been conducted on effective scheduling algorithms for managing HW tasks on computing resources [10] [11] [12] [13]. In particular, the *MOLEN/SESAME* project [12] [13] uses profiling information to decide whether to schedule tasks to run in SW or HW. More complex scheduling policies of reconfigurable hardware tasks are introduced by Compton et. al. [11] including task preemption and the concept of saving the restoring the HW task’s context. While these works simulate runtime reconfiguration of HW tasks, Santambrogio et. al. [10] introduced a run-time environment that is able to dynamically place and/or remove HW tasks on demand. They use online partial bitstream manipulation for proper placement of tasks on a multi-FPGA system.

B. Comparison between FUSE and previous work

Adding OS support to virtualize HW resources in FPGA-based SoC platforms is challenging because they have greater heterogeneity and their resources can be dynamically reconfigured at run-time. SW designers targeting these embedded computing platforms wish to leverage their unique HW accelerators without requiring knowledge of low-level architectural details. This requires OS support for SoCs with either static HW accelerators or using dynamic partial reconfiguration (DPR). FUSE gives a unified view of available processing resources as well as communication between processor(s) and HW accelerators. Unlike BORPH [7] and Kociuszkiewicz *et al.* [8], we do not fix the type of physical link used for communication. Furthermore, our model virtualizes HW accelerators as tasks, instead of processes like BORPH [7].

The two closest works to ours are the ReconOS [3] project, and the work by Compton *et al.* [9]. We use a shared memory model as recommended in [3] to reduce data communication overhead between SW and HW tasks. As outlined in [3], data transfers between SW and HW threads are complicated by the fact that the OS usually employs virtual memory; shared memory buffers set aside by an application to transfer data to or from HW threads are not necessarily contiguous. However, we do not assume virtual memory is available. In FUSE, we allocate a contiguous buffer in kernel memory, map it to user space, and provide HW tasks with its physical address.

Unlike ReconOS [3], which uses a statically loaded abstraction layer, FUSE allows users to dynamically load the OS abstraction for HW tasks at run-time. Furthermore, whereas ReconOS requires user-space SW delegate threads for each HW thread, we do not. Finally, unlike ReconOS, we do not require each HW thread to adhere to a strict HW interface or a fixed signalling protocol for OS calls from HW to SW. Instead, each HW accelerator can have customized interfaces, encapsulating communication protocols in their low-level OS support.

In comparison to Compton *et al.*’s [9] work, where the OS support is registered at OS boot-time, FUSE treats each HW accelerator and its OS support as a general system resource that can be included/updated at runtime and is available to all applications. HW accelerators are viewed as memory-mapped I/O devices; the OS can load their abstraction on-demand without rebuilding and rebooting the OS. Additionally, while the work in [9] has been implemented within a simulation environment, our work has been prototyped on a platform FPGA.

The additional OS support provided by FUSE allows existing SW synchronization mechanisms in the OS to support HW/SW task synchronization in comparison to “HybridThreads” [2]. FUSE also incorporates the idea of tasks as units of execution across the HW boundary but, unlike [2] and [3], is not integrated with a specific programming model (e.g. Pthreads). Instead, we adopt a more generalized OS approach to reduce the SW changes required to port a user application to use existing and newly added HW accelerators. Finally, the “HybridThreads” project’s ability to generate sequential HW tasks from SW code is complimentary to our work. Both this work and ours aim to insulate the SW designer from the expertise of HW design be it via automated tools or an independent HW designer.

Finally, as mentioned earlier, several recent projects looked into effective scheduling algorithms for managing reconfigurable computing resources, where HW accelerators are dynamically instantiated on the FPGA [10] [11] [12] [13]. We view the work presented in this paper as complementary to these projects. *Our objective is to provide a framework for OS abstraction of the underlying architectural configuration to run hardware tasks, as opposed to a new scheduling algorithm.*

III. Proposed FUSE Framework

Operating systems typically support concurrency using multithreading programming models. In addition, most recent concurrency models decompose an application into multiple units of execution called tasks—defined as indivisible pieces of work. FUSE abstracts HW accelerators

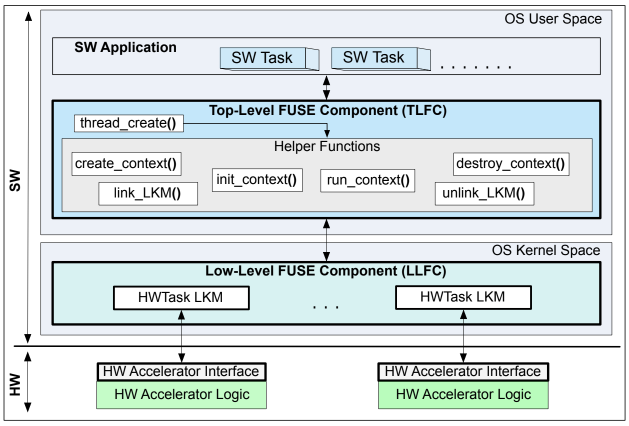

Figure 1. FUSE System Architecture

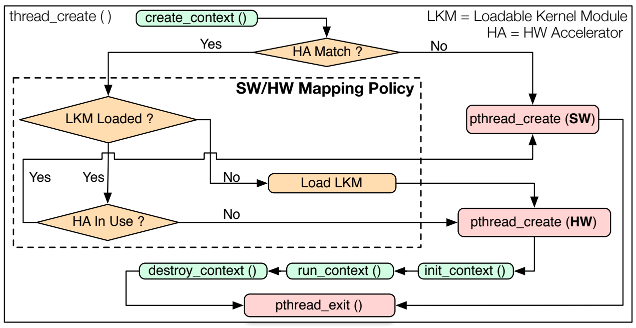

Figure 2. Decision Flow for Top-Level FUSE Component

along with their FUSE support as HW tasks, sharing the resources available to SW tasks. We use multithreading programming models to provide SW designers with access to HW accelerators, enabling them to be viewed as additional computing resources to the CPU. FUSE's new higher-level of abstraction allows SW designers to create their applications with no knowledge of the platform's HW accelerators. This allows SW designers to program using the multithreading programming model's API function calls. The FUSE API includes additional function calls to create and destroy SW/HW tasks. They act as wrappers for the "create"/"destroy" functions to augment their abilities to support HW tasks when a platform contains HW accelerators. This provides a clean and simple way to utilize existing HW accelerators. Furthermore, FUSE introduces no changes incompatible to the underlying OS layer: the programming concept of a task as a unit of execution remains unchanged whether it is scheduled to run as a SW task or a HW task. Figure 1 shows FUSE's two main components, which span the OS user and kernel layers to insulate changes to user and kernel SW from each other, and facilitate portability.

A. *Top-Level FUSE Component (TLFC)*

The *Top-Level FUSE Component* (TLFC) provides middleware between the SW designers and platform HW designers. SW designers are able to port existing FUSE-enabled multithreaded applications to any platform, whether or not it provides HW acceleration. HW designers determine which SW task(s) provide the best performance gains if they can be executed using HW accelerators. The TLFC is provided as a user-space header library that SW designers include to act as a wrapper for their multithreading model's header library. In particular, this header library includes equivalent wrapper functions for creating and destroying HW tasks in lieu of SW tasks on FUSE-enabled platforms.

In addition to these wrapper functions, the TLFC contains "helper functions" (see Figure 1) used to communicate with the *Low-Level FUSE Component*. They create, initialize, run and destroy contexts; they also dynamically load/unload the necessary low-level OS support for a HW accelerator to kernel. FUSE's helper functions utilize the concept of a context to store a dynamic snapshot of the current state of a running task for use within the API wrapper functions; they are not exposed to the end user.

The semantics of creating and destroying tasks of execution within an application remain the same as FUSE preserves the original program flow independent of HW acceleration. Minimal code porting is required though by the SW designer in order to use FUSE. We discuss the implementation details of these function calls and outline what changes the SW designer are required to perform in Section IV. To determine if a task is run in SW or HW, the TLFC follows the decision flow shown in Figure 2. While FUSE allows designers to implement any of the existing algorithms for mapping tasks to SW or HW (e.g. [11] [12]), we use a simple mapping policy to facilitate our discussion. Each time a call to create a task is made, the TLFC decision flow is evaluated to enable adaptation to the current state of all existing accelerators while the system remains live. Nevertheless, the SW designer's perspective of the created tasks remains unchanged: all tasks are created and executed upon request. If a task is not mapped to a HW accelerator, then FUSE reverts to creating a normal SW task.

From Figure 2, FUSE first checks if a matching HW accelerator exists for the SW task (*HA Match?*). If no match exists, then FUSE creates a SW task using the original multithreading model's "create" function. However, if an accelerator exists, then a check is made to see if its corresponding OS support (i.e. LKM) is already loaded. If it is not loaded, then the accelerator is idle and FUSE proceeds to load OS support and run the task in HW. However, if OS support is loaded, then FUSE checks to see if the accelerator is in use by another task. If not, a HW task is created. The case where matching HW accelerator(s) exist, but are in use by other tasks, is also handled; FUSE creates a SW task instead. While our proof of concept of FUSE does not require DPR support, this mapping policy can easily be altered to check and see if an additional HW accelerator can be instantiated. All the necessary OS support for reconfigurable computing systems using DPR (e.g. [10]) currently exists in FUSE.

B. *Low-Level FUSE Component (LLFC)*

The second part of the framework, the *Low-Level FUSE Component* (LLFC), consists of the low-level support added to the OS kernel (see Figure 1). It exposes HW accelerators on the FPGA's fabric to the TLFC using *run-time* loadable device drivers (i.e. LKMs) [14]. The drivers implement the low-level communication mechanisms that enable the TLFC to load, initialize, and perform data or control I/O transactions with HW accelerators.

As mentioned earlier in Section I, each HW accelerator attaches to the system via a HW interface (discussed in Section IV) that is accessible by its own LKM (see Figure 1). The LKM provides a low-level SW abstraction of the HW accelerator interface, thus providing the communication link between the HW/SW boundary in the system. Given the modular layered structure of the FUSE framework, changes made during the design process to one LKM will not effect other modules in the system. This modular approach also applies to the HW design: changes to a HW accelerator need only conform with its corresponding HW accelerator interface. If necessary, its LKM can be patched in order to expose the new functionality. However, FUSE's ability to dynamically load/unload new LKMs will provide faster, possibly live, integration of new/updated HW accelerators and their LKMs into existing applications as the TLFC and application SW remain unchanged.

IV. Implementation of the FUSE Framework

In this section, we detail the implementation of the FUSE framework. We also outline our design choices and how they affect the overall design outcome. To demonstrate our FUSE concept, we use an embedded Linux OS [5] as our run-time environment and POSIX threads (Pthreads), a widely used multithreading programming model. The Pthreads API contains an extensive list of functions that allow SW designers to add concurrency into their application code. One of the most important functions is *pthread_create()*, which creates a new concurrent thread of execution. The user-space header library provided by the TLFC is called <*fuse.h*> and includes the definition of the *thread_create()* function call. *thread_create()* acts as a wrapper for Pthreads' *pthread_create()*, incorporating additional policies to enable transparent migration of tasks to HW accelerator(s).

A. User-space Implementation of TLFC

During the OS startup, HW accelerators attached to the system are detected and an initial housekeeping step is performed for each of them that includes registering their corresponding LKM. Additional LKMs can be loaded at runtime as desired. Information about existing accelerators is saved into a look-up table stored in the OS, which is continually updated to indicate the current state of all the accelerators in the system. During application execution, FUSE uses the look up table to associate the task's function name with an existing HW accelerator. HW designers name each HW accelerator to match the corresponding function name specified by the SW developer as part of the parameter list of the *pthread_create()* function. The only application SW changes required to enable FUSE support are: 1) each *pthread_create()* call is replaced with the FUSE-based version *thread_create()* call, and 2) <*fuse.h*> is included instead of <*pthread.h*>. For example, Figure 3 shows part of a user application that includes the FUSE header library (Line 4) and creates two threads of execution using *thread_create()* (Line 26 and 27) that have the same four parameters as the original *pthread_create()*.

```
1   /* Example of a user program */
2
3   #include <stdio.h>
4   #include <FUSE.H>
5
6   void* rbg2yuv_thread_function(void *arg); //color conversion thread in SW
7   void* dct_thread_function(void *arg); //Discrete-Cosine Transform in SW
8
9
10  thread_param param[data_size]; //global (shared between threads)
11
12  int main( )
13  {
14
15  pthread_t sw_thread_1;
16  pthread_t sw_thread_2;
17
18  hwthread_attr_t  sw_thread_attr;
19
20  int sw_ret_1, sw_ret_2;
21  void* sw_thread_result_1;
22  void* hw_thread_result_2;
23
24  //two threads are crated joinable.
25
26  sw_ret_1 = thread_create(&sw_thread_1, NULL, rgb2yuv_thread_function, &param);
27  sw_ret_2 = thread_create(&sw_thread_2, NULL, dct_thread_function, &param);
28
29  .....
30
31  sw_ret_1 = pthread_join(sw_thread_1, &sw_thread_result_1);
32  sw_ret_2 = pthread_join(sw_thread_2, &sw_thread_result_2);
33
34  ....
35  }
36
37  void* dct_thread_function(void* arg) {
38    /* initial function code by the SW designer before using FUSE (Left unchanged)
39  }
```

Figure 3. Partial source-code of an application using FUSE

```
1   /* Inside FUSE.H */
2
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <unistd.h>
6   #include <sys/mman.h> /* MMAP_SHARED for shared memory */
7   #include <linux/autoconf.h> /* memory map used to populate table of accelerators */
8   #include <pthread.h>
9
10  #typedef hw_task_t __u32;
11  #typedef hw_task_attr __u32;
12
13  pthread_mutex_t context_mutex;
14
15  void* hw_task(context_structure* cs)
16  {
17      init_context(&cs);
18      mutex_lock(&context_mutex);
19      run_context(&cs);
20      mutex_unlock(&context_mutex);
21      return NULL;
22  }
23
24  int thread_create(pthread_t *t,pthread_attr_t *attr , void* (*fname) (void*), void *arg)
25  {
26    int ret;
27    char* hw_task; /* the hardware accelerator's function name */
28    struct context_structure* cs; /* context structure */
29
30    create_context(&cs);
31
32    accelerator_table_query(&fname, &hw_task); /* (HA_Match?) step */
33
34    if(accelerator_loaded(&hw_task) && !accelerator_idle(&hw_task))
35    {
36      ret = pthread_create(&t, &attr, hw_task, &cs);
37    }
38    else
39    {
40      ret = pthread_create(&t, &attr, fname, arg);
41    }
42    destroy_context(&cs);
43    return ret;
44  }
```

Figure 4. *thread_create()* inside <*fuse.h*> header file

As part of the *thread_create()* function, FUSE uses several internal "helper functions", that exist in the *Top-Level FUSE Component* (See Figure 2), to manage thread creation. Figure 4 shows part of the SW implementation for the *thread_create()* function in the FUSE header library. This implementation uses the concept of a context, which stores a dynamic snapshot of the current state of a running thread. FUSE utilizes four "helper functions" to create, initialize, run and destroy contexts (Lines 30, 17, 19 and 42 respectively).

When creating a thread, FUSE uses *create_context()* (Line 30) to allocate a context structure (Line 28) to hold information related to the thread, such as the function name, user data, and current state. Then the look-up table is checked via *accelerator_table_query()* (Line 32) to find a matching accelerator name. The *accelerator_loaded()* and *accelerator_idle()* (Line 34) functions assert whether a matching accelerator can be locked to this context. If an accelerator exists and is currently not in use, then FUSE creates a thread that performs HW task initialization (Line 36).

This newly created thread will use *init_context()* (Line 17) and *run_context()* (Line 19) to run the HW accelerator. *init_context()* fills the context structure with the related information (fetched from the look-up table) and loads the corresponding LKM for that particular accelerator using the *insmod()* system call provided by the OS. This system call is made only once when the accelerator is first instantiated. In addition, *init_context()* opens a device file handler to the accelerator's device file using the *open()* system call, and then performs memory mapping to the accelerator's local memory space via the device handler using the *mmap()* system call. Each HW accelerator has special implementations of these OS system calls defined by their loaded LKM.

When the accelerator is ready to process data, *run_context()* (Line 19) performs data and control I/O on the memory mapped space, along with proper data marshalling, according to how the accelerator processes data. This entails using the *read()*/*write()*/*ioctl()* system calls for sending/receiving control signals in addition to simple array dereferencing of the shared-memory space when sending/receiving data values. Finally, FUSE calls *destroy_context()* (Line 42) to deallocate the context structure, update the look-up table, release the shared memory, and close the device file handler. This is done after the HW accelerator is finished processing any remaining data.

B. Kernel-space Implementation of LLFC

The OS kernel is structured as a collection of modules, some of which can be automatically loaded and unloaded on demand. The *Low-Level FUSE Component* (LLFC) resides in the OS kernel space and handles the direct abstraction of the HW accelerator resources through their HW accelerator interface. Therefore, to accommodate HW accelerator functionality that can be loaded/unloaded per request, the kernel-space implementation of the FUSE framework maps a loadable kernel module (LKM) to each HW accelerator. Thus, HW accelerators become miscellaneous platform devices that appear as autonomous entities in the system and have direct addressing from the CPU; each LKM is an object file, with code that can be loaded and unloaded from the kernel during runtime while the kernel is already in memory and executing.

A HW accelerator's LKM implements miscellaneous device driver functionality that treats the accelerator as a memory-mapped I/O device peripheral. The memory-mapped I/O implements communications between the CPU and the system peripherals using a common instruction set to simplify system design. In a system containing memory-mapped I/O devices, the OS creates an address space map that assigns different parts of the memory space to different components of the system. The OS uses FUSE's LLFC to abstract such memory-mapped devices from the user-space as device files that can be opened, read, written and closed. As shown in Figure 5, the LKM uses a special structure called the *hwthread_instance* (Line 20), which contains the physical base address of the memory-mapped device. This allows the kernel-space side of FUSE, the LLFC, to recognize each HW accelerator by its base physical address in the address space map. Meanwhile the user-space FUSE implementation (TLFC) treats each LKM as a file that can be operated upon. Furthermore, each LKM exports a set of file operations that are used by the helper functions inside the top-level FUSE component. As shown in Figure 5, each LKM provides services for *open*, *close*, *read*, *write*, *ioctl*, and *mmap* system calls (Lines 33 to 38). These calls are used in the implementation of the helper functions, *init_context()* and *run_context()*, in the user-space part of the FUSE framework (i.e. TLFC).

```
1   /* LKM Code */
2   #include <linux/module.h>
3   #include <linux/kernel.h>
4   #include <linux/errno.h>
5   #include <linux/mm.h>
6   #include <linux/init.h>
7   #include <linux/list.h>
8   #include <linux/miscdevice.h>
9   #include <linux/xilinx_devices.h>
10  #include <linux/platform_device.h>
11  //asm refers asm-microblaze
12  #include <asm/io.h>
13  #include <asm/uaccess.h>
14  //xilinx dependent definitions
15  #include "hwtask_dct.h"
16  #include <asm/hwtask_ioctl.h>
17  #define BUFSIZE    10 //we have 10 slave registers in HWthread PLB IP core
18  #define DEBUG      0
19  //hardware task module isntance
20  struct hwtask_instance {
21      struct list_head link;  /* for the linked list of hwthread instances */
22      unsigned long base_phys; /* hwthread IP core base address - physical */
23      unsigned int  base_addr; /* virtual addr. for the regfile IO MEM resource */
24      unsigned long remap_size;
25      u32 device_id;
26      wait_queue_head_t wait; /* wait queue for Blocking I/O read or write */
27      int is_inuse;
28      unsigned int buf[BUFSIZE]; /* register file (10 slave regs, 32-bit each) */
29      struct miscdevice *miscdev;
30  };
31  static struct file_operations hwthread_fops = {
32      owner:THIS_MODULE,
33      ioctl:hwthread_ioctl,
34      open:hwthread_open,
35      read:hwthread_read,
36      write:hwthread_write,
37      mmap:hwthread_mmap,
38      release:hwthread_release
39  };
```

Figure 5. partial code showing part of the LKM implementation

C. Hardware Accelerator Interface

The HW accelerator interface physically connects the accelerator's logic to the system's communication network, but does not presume a specific communication network. Furthermore, the HW interface's design can be created to reflect the needs of the accelerator. Its LKM is then customized to abstract the interface's architecture from the rest of the system (See Figure 1). To demonstrate how FUSE enables HW accelerators with LKM support to mimic SW threads, we used the example HW accelerator interface shown in Figure 6 for the shared-bus used in the experimental system discussed in Section V.

Our example HW accelerator interface is comprised of a *Bus Interface*, which performs the appropriate address decoding and signal control for the system communication

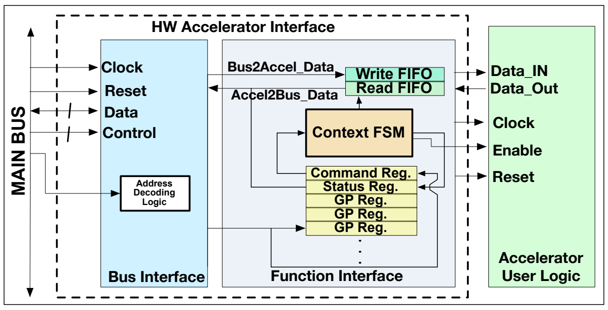

Figure 6. HW Accelerator Interface

network, and the accelerator's *Function Interface* (see Figure 6). The *Function Interface* handles the OS kernel communication with the accelerator's logic using a finite state machine (FSM), read/write buffers, and a register file. The size of the 32-bit register file is configurable to meet our application's requirement, with a minimum of two registers for status and control.

Control values written from the FUSE API to the *Command Reg.* indicate the desired execution state for the HW accelerator, while data read from the *Status Reg.* shows its current execution state. These execution states are similar to those of a POSIX SW thread (e.g. IDLE, RUNNING, and BUSY) and controlled via the *Context FSM*. The TLFC reads the *Status Reg.* to update its HW accelerators' look-up table. In the TLFC implementation described in Section IV, the *run_context()* helper function uses LKM-supported system calls to send commands (e.g. RESET, RUN, and STOP) to the *Command Reg.* Marshalled data is sent to the accelerator via a write FIFO buffer to speed up communication and returned via a read FIFO buffer to be read back by the TLFC. Other possible HW interface designs may include autonomous memory access, in the form of Direct Memory Access (DMA), for better data marshalling. Our current experimental system does not use DMA and is left for future work.

V. Evaluation and Experimental Results

This section outlines the current system configuration used to demonstrate the FUSE framework with three SW case studies that utilize HW accelerators accessed via FUSE. We discuss the overhead incurred from migrating a SW thread to a HW accelerator, and quantify the resource utilization of each HW accelerator with respect to the entire platform. Finally, we outline the performance speedups obtained from using FUSE in conjunction with HW accelerators for each application.

A. Experimental Setup

We demonstrate FUSE using version 2.6.20 of the Petalinux [5] kernel, an embedded version of Linux. Figure 7 highlights the key components of our HW system, adapted from a Petalinux example built on a Xilinx Virtex 5 LX50T FPGA [5] using Xilinx's EDK 10.1.02. System peripherals that are only used during the OS boot process (e.g. the GPIO, FLASH controller, and ethernet controller) are not shown. Our design uses a Xilinx MicroBlaze 32-bit soft-processor and three HW accelerators that are

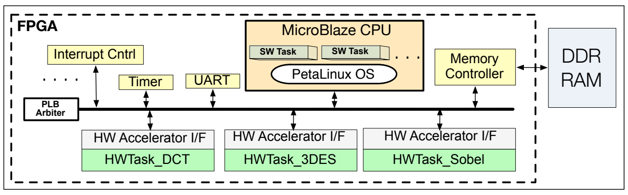

Figure 7. Experimental System Architecture

connected to a shared system bus through HW accelerator interfaces. The MicroBlaze CPU is configured with an MMU, a 4KB data cache and a 2KB instruction cache. As in the example system, the CPU is clocked at 125 MHz, while the Processor Local Bus (PLB) runs at 100 MHz. A 256MB DDR RAM is used by the system as the physical memory through Xilinx's multi-port memory controller.

B. Case Studies

Three multithreaded application examples are implemented: JPEG image compression, Triple-DES encryption–and–decryption, and an image filter. Three HW accelerators are integrated into our system with the example HW accelerator interface shown in Figure 6. Each provides specific functionality for a given application: the JPEG image compression application uses a Discrete Cosine Transform (DCT) accelerator; the Triple-DES (3DES) application uses an encryption accelerator; and the image filtering application uses a SOBEL edge detection accelerator. Table I outlines the resource utilization for the accelerators and their LKMs, along with the overall experimental system, on the Virtex 5 LX50T. These SW tasks comprise the majority of their application's execution time and require minimal HW resource utilization, making them logical choices for HW acceleration. The corresponding LKM size is dependent on the nature of the HW task being abstracted.

Our objective is to quantify the overhead of the runtime abstraction of HW accelerator(s) in the OS. This has two possible uses: 1) static SoC configurations with unique HW accelerator(s) and 2) SoC configurations with DPR support. To isolate the overhead incurred by FUSE from the reconfiguration overhead of DPR, we opted for statically configured SoC platforms. This allows us to separate the impact of FUSE's overhead on performance speedup from the additional bitstream reconfiguration overhead, which has been quantified by the vendor [15]. Our baseline execution time is for the SW versions of these applications using the Pthreads library executing on the CPU without any HW acceleration. On the same platform, we compared these to using <*fuse.h*>. The overhead of running an application using FUSE when there are no HW accelerators is incurred by the check for a (*HA Match?*) (see Figure 2); each check takes ∼100us. For these three applications, using FUSE without HW accelerators increases the runtime negligibly (<0.99%). When HW accelerator(s) exist, the overhead incurred from using FUSE is divided into: 1) the overhead of loading the LKM(s) the first time a HW accelerator is used (12 ms), 2) calling the *open()*, *mmap()*, *ioctl()*, and *close()* functions to communicate with the HW accelerator (0.97ms), and 3) the potential unloading of

Table I
RESOURCE UTILIZATION ON A VIRTEX5 LX50T

Application	HWTask	LKM Size	Flip/Flops	LUTs	DSP48E	BRAMs
JPEG Encoder	DCT	7.3 KB	1329 (4%)	1635 (5%)	4 (8%)	8 (13%)
3DES	Encryption	6.9 KB	1260 (3%)	1550 (5%)	4 (8%)	8 (13%)
Image Filter	SOBEL	6.9 KB	960 (2%)	1620 (5%)	2 (4%)	8 (13%)
Overall System			16416 (57%)	16992 (59%)	10 (20%)	28(43%)

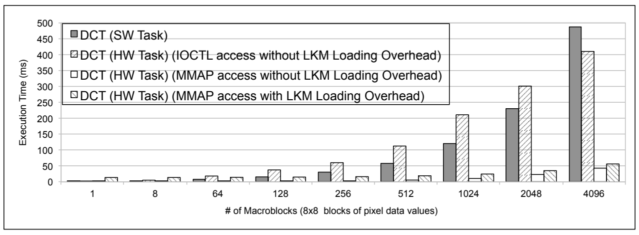

Figure 8. Execution Time of the JPEG Encoder application with different implementations for the DCT task

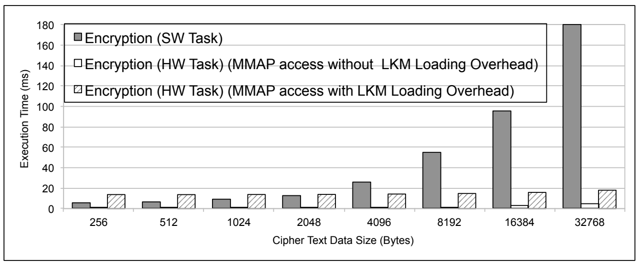

Figure 9. Execution Time of 3DES application with different implementations for the encryption task

the LKM (12ms). For static systems, only the runtime system function calls (e.g. *open()*, etc.) contribute to the runtime overhead as the LKM support would be loaded to the OS upon booting. However, systems supporting DPR would also need to include the overhead of loading their LKMs at runtime. Unloading a LKM would only be performed if the OS is running out of memory space (i.e. 100s of unique LKMs have been loaded at runtime) or when replacing an existing LKM with an updated version. Thus, we consider this scenario an irregular contributor to the runtime overhead.

Image Compression Application: The multithreaded image compression application implements the JPEG compression standard on sample images of varying sizes. The compression algorithm is divided into multiple threads: colour-conversion, 2D-DCT transformation, quantization, Huffman encoding, and a main thread that handles other operations such as read and write. The image is divided into macroblocks that are processed concurrently; each macroblock is an 8x8 pixel set, where each pixel has a 16 bit value. When the DCT HW accelerator is present, FUSE detects the accelerator's availability and migrates the SW thread into HW. Figure 8 shows the execution time of the JPEG encoder application for different DCT implementations with increasing image sizes and thus, an increased number of macroblocks. Two methods of data access supported in the DCT accelerator's LKM are investigated: 1) using the IOCTL system call, which reads/writes *individual* data words to the HW accelerator via its interface; and 2) the MMAP system call, which establishes a direct memory map to the accelerator's address space and enables arrays of data to be copied in a single access. Figure 8 summarizes the execution times of the application using only SW threads compared to using the DCT HW accelerator for both of these access methods excluding runtime LKM loading overhead.

The IOCTL approach generally requires a longer execution time than even the SW approach. This is due to the overhead of the individual system calls required to read/write *each* data word, which increases as the data set increases. This overhead counteracts the potential performance speedup to be gained from HW acceleration. Conversely, the MMAP approach allows entire data sets to be copied with only one system call, greatly reducing the overhead and enabling significant performance gains. Consequently, we opted to use MMAP to set up memory-mapped direct-access to the accelerator for data communications, while using the IOCTL call only for sending and receiving individual control values to the HW accelerator interface's FSM.

Performance speedup compared to the SW implementation increases, for larger data sizes, to a maximum of 11x for our static SoC configuration when the image has 4096 macroblocks. Including the LKM loading overhead that would be incurred for DPR systems reduces the maximum speedup to 8.7x.

3DES Encryption/Decryption Application: The 3DES SW application has three threads; the main thread handles cipher text data read/write from a file, while the two remaining threads perform encryption and decryption on the data. When the encryption thread is created, FUSE will migrate it to the HWTask_3DES accelerator. In this example, only the MMAP approach is used due to its reduced impact on execution time. Figure 9 shows the execution time for the application with SW and HW versions (with and without LKM loading overhead) of the 3DES encryption thread for varied sizes of the cipher text data.

The HW accelerated versions have better performance than the SW thread version for smaller data sizes when only the overhead of the system function calls is included. However, when including LKM loading overhead as well, visible speedups occur for larger cipher texts (>4KB) where the execution time is >25ms. A maximum performance speedup of 37x is achieved without LKM loading overhead, which is reduced to 10x when it is included.

Image Filtering Application: The third application is a multithreaded image filtering application that has two threads performing image edge detection and image sharpening. The image edge detection thread uses a SOBEL operator with a 3x3 window size. This thread is used to detect horizontal and vertical edges of objects in an image. The second thread sharpens the image contents

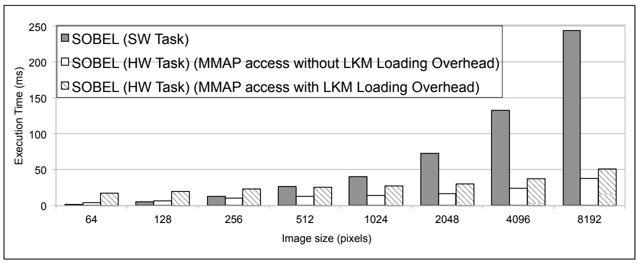

Figure 10. Execution Time of the Image Filter application with different implementations of the SOBEL task

using a Laplacian operator. The edge detection SW thread is migrated by FUSE to the HWTask_SOBEL accelerator. Data processed by the HW accelerator is sent to the Laplacian thread in SW to complete the filtering process. Figure 10 shows the execution times of the image filtering application using a SOBEL task HW accelerator compared to its original SW task using different image sizes. The image size is given in terms of the number of pixels in the image where each pixel has an 8-bit value.

When LKM loading overhead is included, only images containing at least 1024 pixels achieve a visible performance speedup with HW acceleration compared to the SW-only implementation. The performance speedup for SOBEL edge detection using an image with 8192 8-bit pixels in HW is 6.4x excluding LKM loading overhead, which reduces the speedup to 4.7x of the SW version.

VI. Conclusion and Future Work

In this paper, we have presented FUSE, a framework that abstracts embedded computing architectures from SW designers creating multithreaded applications. FUSE provides transparent integration of HW accelerators into the design by virtualizing HW accelerators from SW designers so they can be treated as "HW tasks" of execution. We have demonstrated FUSE on a FPGA-based SoC platform running PetaLinux with FUSE for three different applications to achieve performance speedups with HW accelerators. We have also quantified the execution time overhead incurred from using this framework with the MMAP approach and corresponding kernel support for both static and DPR-based systems. An application's potential performance speedup is greatly affected by its execution time. For our case studies, performance speedups for static SoC platforms range from 6.4x-37x, dropping to 4.7x-10x when LKM loading overhead is included. To investigate how much performance improvement can be obtained from enabling HW tasks to independently marshal data during their execution, we are investigating a HW accelerator interface that includes a DMA path to the physical memory. We also plan to investigate communication network structures other than the shared-bus example we illustrated in this work and compare the impact on the overhead. Finally, we are investigating how the bitstreams for reconfiguring HW accelerators can be loaded in parallel with their LKMs to mask some of the runtime overhead.

Acknowledgments

The authors thank the National Science and Engineering Research Council, the Canadian Microelectronics Corporation, and Petalogix for their support of this project and Dr. Katherine Compton and Dr. Mehrdad Moallem for their advice.

References

[1] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Burlington, MA: Morgan Kauffman, 2007.

[2] D. Andrews *et al.*, "Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link," *IEEE Micro*, vol. 24, pp. 42–53, Apr. 2004.

[3] E. Lübbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, pp. 1–33, 2009.

[4] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A Computational Model for Reconfigurable Devices," in *Proc. of the IEEE Intl. Conf. on Field Programmable Logic and Applications*, 2006.

[5] (2009, February). [Online]. Available: http://www.petalogix.com

[6] K. Kosciuszkiewicz, F. Morgan, and K. Kepa, "Transparent management of reconfigurable hardware in embedded operating systems," in *IEEE Computer Symposium on Emerging VLSI Technologies and Architectures*, vol. 00, Mar 2006, pp. 432–433.

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

[8] K. Kosciuszkiewicz, F. Morgan, and K. Kepa, "Run-Time Management of Reconfigurable Hardware Tasks Using Embedded Linux," in *International Conference on Field-Programmable Technology.*, Dec 2007, pp. 209–215.

[9] P. Garcia and K. Compton, "A reconfigurable hardware interface for a modern computing system," in *IEEE Symposium on Field-Programmable Custom Computing Machines.*, April 2007, pp. 73 –84.

[10] V. Rana, M. Santambrogio, D. Sciuto, B. Kettelhoit, M. Koester, M. Porrmann, and U. Ruckert, "Partial dynamic reconfiguration in a multi-FPGA clustered architecture based on Linux," in *IEEE International Symposium on Parallel and Distributed Processing.*, Mar 2007, pp. 1–8.

[11] K. Rupnow, W. Fu, and K. Compton, "Block, Drop or Roll(back): Alternative preemption methods for RH multitasking," in *IEEE Symposium on Field-Programmable Custom Computing Machines.*, 2009, pp. 63–70.

[12] V.-M. Sima and K. Bertels, "Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform," in *IEEE International Symposium on Parallel Distributed Processing.*, May 2009, pp. 1–6.

[13] K. Sigdel, M. Thompson, A. Pimentel, C. Galuzzi, and K. Bertels, "System-level runtime mapping exploration of reconfigurable architectures," in *IEEE Intl. Symposium on Parallel Distributed Processing.*, May 2009, pp. 1–8.

[14] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[15] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgeford, "Enhanced Architecture, Design Methodologies and CAD Tools for Dynamic Reconfiguration for Xilinx FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2006, pp. 1–6.