OPERATING SYSTEM ABSTRACTIONS OF HARDWARE ACCELERATORS ON FIELD-PROGRAMMABLE GATE ARRAYS

by
Aws Ismail
B.A.Sc. EE (Hons.), University of Windsor, 2005

a Thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Applied Science

in the

School of Engineering Science

Faculty of Applied Science

SIMON FRASER UNIVERSITY

Summer 2011

APPROVAL

Name:	Aws Ismail
Degree:	Master of Applied Science
Title of Thesis:	Operating System Abstractions of Hardware Accelerators on Field-Programmable Gate Arrays
Examining Committee:	**Dr. Carlo Menon** Assistant Professor, School of Engineering Science Chair
	Dr. Lesley Shannon, P.Eng. Assistant Professor, School of Engineering Science Senior Supervisor
	Dr. Arrvindh Shriraman Assistant Professor, School of Computing Science Supervisor
	Dr. Alexandra Fedorova Assistant Professor, School of Computing Science External Examiner
Date Approved:	August 19, 2011

Abstract

Traditionally, one of the main functions of the Operating System (OS) is to abstract the programming model from the low level details of the specific HW platform resources. However, in an FPGA-based SoC with HW accelerators, even with an OS layer, there is no unified HW/SW framework that provides: 1) transparency to the SW designer at the application level; and 2) an interface and OS support for easy HW accelerator integration by the HW designer at the platform level.

This thesis presents a *Front-end USEr* framework, called FUSE, that introduces a set of policies and mechanisms for HW accelerator abstraction. We illustrate FUSE as an API for an embedded Linux OS with POSIX threads on Xilinx's MicroBlaze on a Virtex5 FPGA. For three different applications and HW accelerators, we achieve performance speedups ranging from 5.8-9.0x.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, there has been a rising interest in reconfigurable devices, specifically in *Field Programmable Gate Arrays* (FPGAs) [1]. Initially, this rise in interest was due to the programmability of these devices, which is particularly useful during the development and prototyping stages of the circuit design process.

Due to their increasing density, FPGAs are able to implement complete *Systems-on-Chip* (SoCs), like *Application-Specific Integrated Circuits* (ASICs), for computing solutions (e.g. embedded SoCs [2]). Furthermore, FPGA vendors now provide soft general-purpose processors (soft processors) to facilitate computing solutions that combine custom hardware (HW) *Intellectual Property* (IP) cores on the same FPGA device to create a heterogenous computing system. These custom HW cores are generally used to accelerate application execution time and thus are commonly called HW accelerators [3].

As FPGA-based SoCs increase in complexity and heterogeneity, there is an apparent need for improved system software (SW) support for their user applications. This support must come in the form of a middle layer, which provides transparency to hardware and software designers alike. Traditionally, this layer has been filled by the Operating System (OS), which abstracts the underlying hardware and at the same time enables portability and platform independence [1] [4]. Normally, adding an OS would help simplify the SW development process by hiding the low-level details of HW resources from the SW designer.

However, a traditional OS is challenged by the heterogeneity of computing systems implemented on FPGAs as they can combine one or more soft processor core(s) with dedicated HW processing elements (accelerators) that are intended to be used to achieve a performance speed-up compared to a SW-only solution. There is a need to extend the system software support provided in the the OS to better abstract HW accelerators as additional computing resources. This extension must be transparent to the SW designer, thus enabling efficient use of the HW accelerators.

1.1 Motivation

Initially, FPGA vendors did not support an OS for their soft processors. A rather simple layer of system SW support was usually provided by the vendor to help setup an execution environment enabling user applications to run on the FPGA. However, adding an OS provides a better execution environment that: eases programming, increase portability, and releases software designers from managing and sharing system hardware resources (i.e., processors, memories, input/output peripherals). For FPGA-based SoC computing platforms, this level of abstraction, available in traditional computing workstations, does not exist. Specifically, for FPGA-based SoC computing platforms, traditional OS support is readily available but the additional complexities of abstracting HW accelerators have not been fully incorporated.

Additionally, when targeting FPGA-based SoC computing platforms, software and hardware designers must face the inevitability of dealing with *HW/SW co-design*. *HW/SW co-design* [5] generally implies that an application will have software running on a CPU in conjunction with specialized hardware accelerators (i.e., hardware circuits specifically designed to speed up and parallelize the execution of performance demanding parts of the application). Although HW/SW co-design has been around for more than a decade [5], it still needs to alleviate some open problems, such as the increasing level of heterogeneity in FPGA-based SoCs. Traditional HW/SW co-design techniques normally view hardware accelerators as passive co-processors, thus, further widening the gap between software and hardware designers. This gap hinders design-space exploration and severely impairs the potential for design reuse.

As a motivation, we see that adding an OS can leverage the aforementioned gap between both sets of designers [1]. In its basic form, an OS is a software/hardware abstraction layer that abstracts a computing platform into a single virtual machine. Therefore, an OS is becoming increasingly common as it enables programming model abstractions that simplify programming by abstracting the low-level details of HW peripherals from the SW designer [4]. For an FPGA-based SoC, adding this layer is seen as a positive step, but it still lacks proper mechanisms that enable the software designer to view that computing platform as a general platform with an added benefit of achieving application performance speedup if possible [1].

The lack of proper mechanisms in current OS solutions impose limits on:

1. Communication between a software application and hardware accelerator(s): while writing user software & designing hardware accelerator(s), software and hardware designers have to be aware of specific interfacing details.

2. Available programming paradigms: advanced programming concepts such as multithreading assume unified memory space.

3. Portability of applications across different FPGA-based SoCs: it is burdensome for software designers to maintain their hardware-agnostic, high-level programming approaches, and it is challenging for hardware designers to design HW accelerators that can run across different platforms, without any change in the HDL code.

By extending OS support to include HW accelerators, the low-level HW interaction details can also be masked, facilitating the writing of programs that utilize HW accelerators efficiently.

1.2 Objective

While there has been extensive research investigating operating system support for FPGAs in general, and FPGA-based SoC computing platforms in particular, few have focused on providing an abstraction to leverage their heterogenous aspect; specifically when the system has one or more hardware accelerators. Our objective is to allow the user to customize the operating system at runtime to support existing hardware accelerators as additional computing resources for software designers, allowing the OS to automatically schedule the application(s) to leverage them.

Given an FPGA-based SoC, we show the feasibility of facilitating SW designers use of HW accelerators. Our goal is to provide the same type of abstraction to HW accelerators, that is available to processors. Through a set of policies and mechanisms, the software designer is presented with a run-time framework that leverages the existence of hardware accelerators. We aim to show that programming models such as multitasking can be extended to enable software designers to view HW accelerators as possible HW tasks similar to SW tasks. We also quantify the overhead introduced by this abstraction and contrast it against the expected performance speed-up.

1.3 Contributions

In this thesis, we demonstrate a *F*ront-end *USE*r run-time framework, *FUSE*, for abstracting computing architectures from SW designers creating multithreaded applications. We show that using FUSE is beneficial 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 [6] [7] in the context of a multithreading application. The contributions of this work are as follows:

1. A modular front-end user framework with a customizable data/control communication interface to HW accelerators.

2. OS Kernel support for *on-demand instantiation* of accelerators, similar to a SW dynamically linked library (DLL).

To demonstrate FUSE, we have designed a simple application programming interface (API) based on the POSIX thread standard and integrated it with the PetaLinux operating system [8] as a user library for a MicroBlaze CPU. We use a Xilinx Virtex 5 FPGA and three case studies to illustrate the overhead of loading OS support for an accelerator, along with negligible runtime overhead for the FUSE framework, resulting in performance speedups ranging from 5.8-9x.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of related work on OS support for systems with HW accelerators on FPGAs. Chapter 3 introduces the concepts and key ideas behind FUSE while Chapter 4 describes the implementation of FUSE components and its support for HW accelerator virtualization. Chapter 5 demonstrates the use of the FUSE framework, along with its overhead and performance speedups for a set of case studies. Finally, Chapter 6 summarizes the conclusions of the work and possible areas of future work.

Chapter 2

Background and Related Work

This chapter covers several aspects of reconfigurable computing, specifically focusing on FPGA-based, Systems-on-Chip (SoC), embedded computing platforms as an example. We also give an overview of the available OS support for such computing platforms as background for the FUSE framework presented in Chapter 3. Section 2.1 gives an overview of FPGAs. Section 2.2 gives and overview of FPGA-based SoCs by outlining the use of processors in such systems and then give an overview of hardware acceleration concepts. Finally, Section 2.3 looks at the existing research into operating system support for such systems and Section 2.4 discusses how our proposed solution relates to previous research on OS support for HW Accelerator integration.

2.1 Modern FPGAs

Field-programmable gate arrays (FPGAs) are integrated circuit (IC) devices that consist of an uncommitted array of logic resources, and are reprogrammable in both logic function and interconnect. Xilinx introduced the the symmetrical island-style

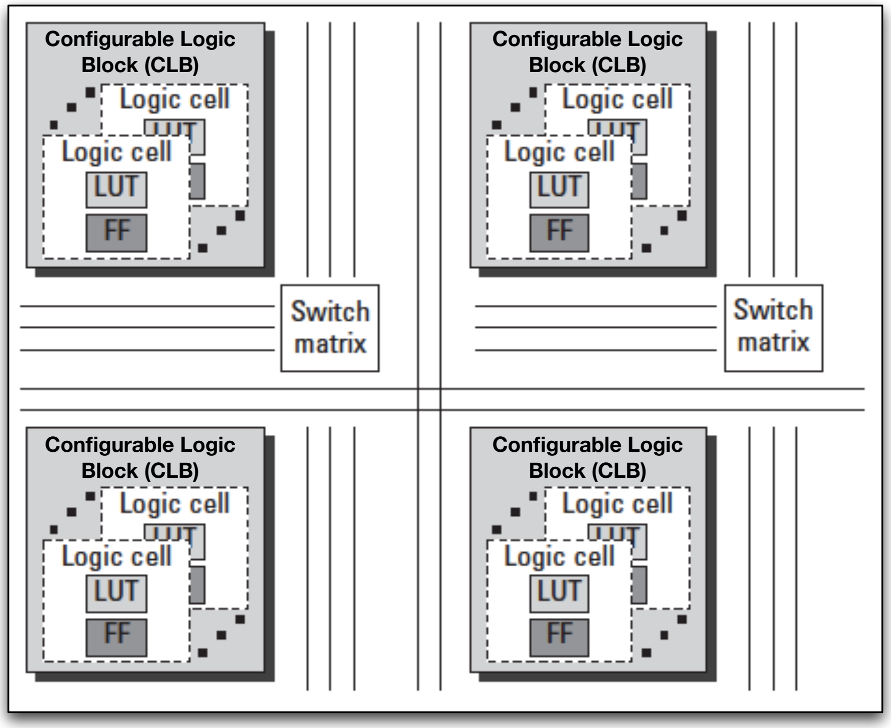


Figure 2.1: A top-level block diagram of an FPGA architecture

FPGA architecture, depicted in Figure 2.1, in the 1980s [9]. It consisted of a regular array of configurable logic blocks (*CLBs*), connected by interconnection networks, which were comprised of wiring channels connected by routing blocks, called *switch matrices*. The CLBs are comprised of *logic cells* containing look-up tables (*LUTs*) and registers or flip-flops (*FFs*). These logic cells allowed for flexible implementation of logic functions within the FPGA fabric, often distributed across several CLBs.

Since their inception, FPGA devices were commonly used for rapid prototyping during the ASIC system design life cycle. Many users of ASIC technologies began adding FPGAs as part of the design life cycle by switching to ASIC prototyping on FPGAs for lower cost functional verification and reduced design risks [10]. FPGAs were, and still are, used for reprogrammable implementations that allow the user to

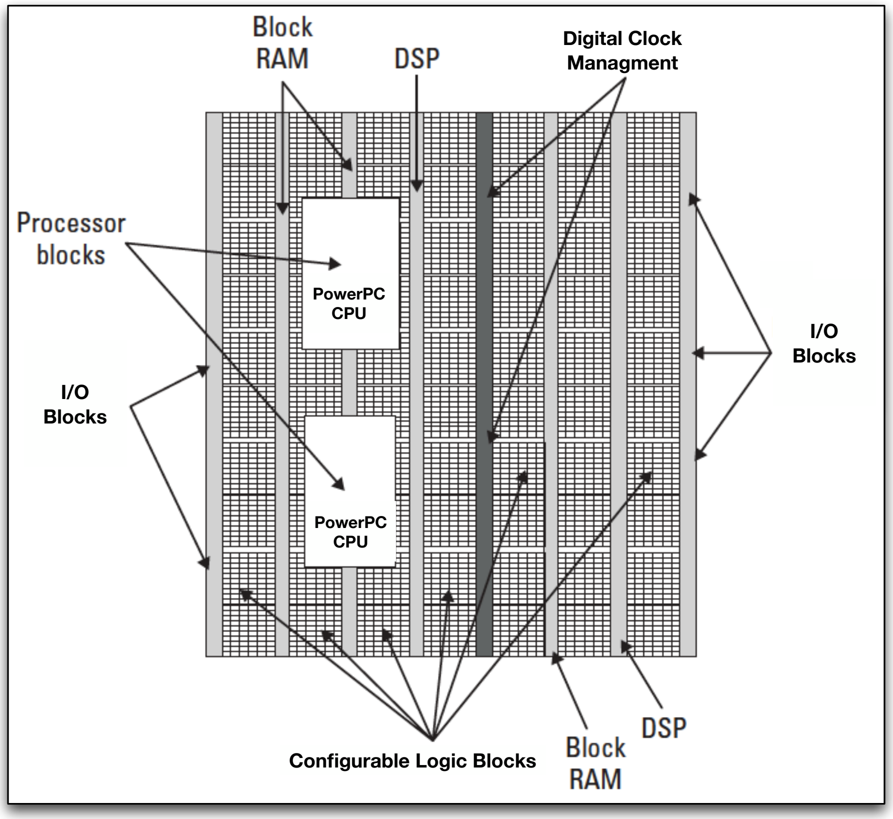


Figure 2.2: Floor plan of Xilinx's Virtex 5 FPGA, consisting of configurable logic blocks, BRAMs, Hard processor Blocks, DSP Blocks, etc.

make quick design changes for faster development time.

As FPGA densities increased, heterogenous components have been added by vendors, facilitating the implementation of complete SoCs on an FPGA similar to ASIC designs [10]. Users and researchers alike started to view FPGAs as a design platform for reconfigurable heterogenous SoCs [11] [12]. A typical SoC design uses these components as functional building blocks to improve the performance and area efficiency of the implemented circuits [2]. Most recent FPGA device families (e.g. Xilinx's Virtex 2 Pro [13], Virtex 5 [14], Virtex 7 [15], and Altera's Stratix [16] and Cyclone [17]) include such functional blocks and are marketed by vendors as implementation targets for complex SoCs.

For example, Figure 2.2 shows the floor plan of a Xilinx Virtex 5 FPGA and highlights the dedicated function blocks included within the reconfigurable fabric. Examples of these blocks include:

- **Multipliers/DSP Blocks:** To speed certain arithmetic functions often found in signal processing applications such as multiply-and-accumulate (MAC).

- **Input/Output Blocks:** The external pins of FPGAs are mapped to regularly distributed I/O Blocks (IOBs), which support a wide set of IO interface standards. Modern FPGA families also include specialized high-speed interfaces for serial communication protocols allowing the creation of multi-FPGA SoC platforms.

- **Memory:** Platform FPGAs now feature dedicated memory blocks called Block RAMs (BRAMs), which provide a higher density and better performance than LUT-based memory. Distributed within the FPGA chip area, these BRAMs can be used to implement read-only memories (ROMs), random-access memories (RAMs), and also storage elements such as First-In First-Out (FIFO) buffers and content-addressable memories (CAMs).

- **Processor Intellectual Property (IP):** They come in two forms, soft processors and hard processors. Soft processors are designed using HW description languages (e.g. VHDL) and can be synthesized onto a FPGA. Hard processors are part of the underlying FPGA fabric. Processors for FPGA-based SoC platforms are discussed in detail in Section 2.2.

In combination with integrated processor IPs and optional external memory (e.g. DDR SDRAM), the FPGA can implement an entire embedded computing system. FPGA-based Systems-on-Chip (SoCs) are now seen as a viable solution for building computing platforms for which hardware abstraction from SW designers (i.e. operating system support) becomes an important consideration [18] [19] [20]. These FPGA-based SoCs are the target implementation of the proposed framework in this thesis (see Chapter 5 for an example target platform).

2.2 FPGA-based SoCs

A major advancement for FPGAs was the ability to include dedicated processors along with specialized custom HW. The combination of these two parts are key to FPGA-based SoC designs. This section deals with FPGA-based SoCs that have soft processor IPs along with custom HW blocks, called HW accelerators. In section 2.2.1 we give discuss the state of the art of processor IPs and soft processors, in particular. Section 2.2.2, gives and overview of using HW acceleration with soft processors to maximize the design performance.

2.2.1 Soft Processors State of the Art

Although hard processors are available on some FPGA families (e.g. PowerPC is only available on certain devices of Xilinx's Virtex 2 Pro [13], Virtex 4, and Virtex 5 [14] families but not on Virtex 6 and Virtex 7 families [15]), it is now possible to define, build and customize a soft processor using any FPGA's available logic primitives [21].

While conventional processors were designed to be built using custom silicon or

ASIC fabrication technologies, soft processors were being built and optimized specifically for efficient implementation on FPGA fabrics. The PicoBlaze [22], for example, is designed for implementing simple state machines, programmed in assembly language on Xilinx FPGAs, while Altera's Inc. The NIOS II family of 16-bit and 32-bit soft processors [23] and Xilinx's MicroBlaze 32-bit soft processor [24] are designed as general-purpose processors, specifically optimized for their perspective vendor's FPGAs. Furthermore, both NIOS and MicroBlaze are RISC architectures designed to be used with general-purpose compiler tools and programmed in high-level languages such as C. Both of these processors are also able to run an embedded version of an operating system, such as embedded Linux [8].

2.2.2 Using HW Accelerators with Soft Processors

FPGA-based SoC computing usually combines one or more processor IP cores along with custom HW function cores, commonly called *HW accelerators*. While current soft processors can reach a maximum frequency of $\sim$125 MHz [24] and are good at running general-purpose applications, many applications contain portions that are computationally and data intensive. Classic examples of such tasks are vector computations, floating point calculations, graphics, and multimedia processing [25]. To achieve better performance, these parts of the application needs to be *accelerated* by running directly on HW.

To accelerate code running on a soft processor using custom HW accelerators, the accelerator cores have to compute the equivalent of several processor instructions in a single clock cycle [26]. As a result, it becomes more attractive to accelerate larger *functions* or *procedures* in an application rather than single instructions, so that the communication overhead between the processor and the HW accelerator is minimized [26]. *Function* acceleration is typically implemented over general-purpose buses, shared memories, coprocessor interfaces, or special point-to-point connections such as Xilinx's fast simplex links (FSLs) [9]. In this thesis, we demonstrate our work with a set of case studies that use HW accelerators implemented over a general-purpose bus interconnect (See Chapter 5 for more details).

2.3 Operating Systems for FPGA-based SoCs

While this section discusses operating system support for SoCs in general and FPGA-based SoCs in particular, it is important, at this point, to distinguish between traditional SoCs and reconfigurable SoCs. Generally, FPGAs are used as platforms for designing SoCs. The term *reconfigurable* is used to denote the ability of the FPGA to be reconfigured at run-time through a technique called run-time reconfiguration (RTR) [27]. When part of the FPGA is reconfigured while the other part remains static, the term dynamic partial reconfiguration (DPR) [28] is used. DPR or RTR is usually used to increase the lifecycle of the design by allowing live updates and the reduction of the design's footprint [27]. If a FPGA-based SoC uses DPR or RTR then it is aptly called a *reconfigurable SoC*.

FPGAs can also be used as a platform for designing static SoCs, where DPR is not needed or used. Regardless of DPR support, all FPGA-based SoCs (i.e. dynamically reconfigurable or static) will have a level of heterogeneity that require abstraction from the SW designers, so they do not have to deal with the underlying details of hardware resources. While the work in this thesis deals with static FPGA-based SoCs, its conclusions can be extrapolated to reconfigurable SoCs where DPR is supported.

The rest of this section will outline the OS support for FPGA-based systems without focusing on whether or not DPR is supported. First, we provide background on the traditional role of the OS, and then discuss how this role is adapted to support the heterogeneity of FPGA-based SoCs.

2.3.1 Traditional Role of the OS

Hansen [29] described an operating system as a set of manual and automatic procedures that enable a group of applications to share computer resources efficiently. Therefore, the function of an operating system can be viewed in two ways [4]:

1. As a resource manager.
2. As an implementor of virtual computers

Acting as the resource manager, the OS enables resource sharing between running applications. Applications will share for the use of physical resources such as processor time, storage space, and peripheral devices. It also means that applications can cooperate by exchanging data. Resource sharing is an economic necessity [29], and the role of an operating system is to make the sharing transparent from user applications.

To achieve resource sharing, the OS virtualizes the underlying resources. Therefore, in its basic form, the OS is merely a layer of software that abstracts a computing system as a unified platform [30]. Operating system calls represent the standard set of operations implemented as libraries for each specific platform. SW designers utilize the operating systems set of virtual operations, with the knowledge that these same operations will be available on all platforms in order to ease the burden on the software designer when migrating the software application in the future.

2.3.2 OS support for FPGA-based SoCs

In order to manage heterogenous HW resources on FPGA-based SoCs, Burns *et al.* [31] recognized the need for an OS. Burns *et al.* [31] concluded that it is more efficient to develop a separate run-time system for HW resources that addresses the common requirements of several applications rather than adding run-time support in every application manually.

Additionally, Nollet *et al.* [32] proposed that one has to provide a management infrastructure in the form of an operating system. This OS must be capable of providing a similar set of services for the heterogenous tasks, as a traditional OS does for software applications in a multi-processor environment. Wigley and Kearney [33] [34] later expanded on the aforementioned previous works [31] [32] and proposed the following axioms for operating systems targetting reconfigurable SoCs in general:

1. the primary focus should be reducing complexity as seen by the user
2. the operating system must form a contract between the systems software and the applications designer.

These axioms are a natural extension of the traditional operating system's role in the reconfigurable SoC domain, essentially advocating a logic resources management interface at the application level (i.e. API).

With potentially many peripheral interfaces and associated heterogeneous HW cores, the increased complexity of FPGA-based SoCs has to be managed by an adequate Operating System (OS). One solution is to use a relatively simple Real Time OS (RTOS). An RTOS is sufficient to support a simple computing systems that does not require peripheral I/O or dedicated controllers. Although RTOSs are sometimes able to manage multitasking [35], these library or microkernel-based operating systems often do not provide memory protection, virtual memory, or process management (e.g. LibXil [36]). Therefore, the heterogeneous nature of FPGA-based SoCs can only be met by a full-featured OS rather than an RTOS, which provides, in addition to process and virtual memory management, a device abstraction layer, advanced I/O services, and hierarchical file systems.

Early research on operating systems for reconfigurable SoCs has assumed or implied a master/slave hardware architecture, whereby a conventional microprocessor based system (often a desktop PC) is interfaced to one or more FPGAs that provided the computationally intensive resources. This has led to a focus on certain requirements such as logic area partitioning [33], management and communication [32] [34]), and dynamic module placement [31] [37]. The master CPU systems execute whatever mainstream operating system is conveniently available, typically Linux or a Unix variant. This split architecture, however, can create a performance bottleneck on the system bus or interconnect, and limit the cohesion possible between the conventional and reconfigurable components of the system.

The advent of processor IP cores for FPGA lead to an increased trend toward the use of operating systems that run on those processors [38] [39]. The main processor, or processors, on the FPGA-based SoC can now host a fully featured OS (e.g. PetaLinux [8]). Consequently, Williams and Bergmann [40] expanded on Wigley and Kearney's axioms [33] [34] by outlining the key requirements of what a *FPGA*-based SoC OS should support:

1. Sequential (processor-based) execution, with a familiar programming paradigm as a starting point for application development. It is unreasonable to expect

that all users will have hardware design capabilities, and therefore, a graceful transition of application code from conventional computing platforms must be offered;

2. Interoperability with existing general purpose computing infrastructure, including networking, file storage and other I/O device interfacing;

3. Scaleable architectures, supporting single-chip, multi-chip and multi-board computing systems within the same operating system architecture.

4. A logic management interface that abstracts operations, such as dynamic partial reconfiguration (DPR), in support of the hardware process model;

5. A process model that seamlessly supports hardware accelerators and software within the same architecture, including support for standard interprocess communication methodologies across homogenous and heterogeneous application implementations; and

6. Integration of hardware components developed in a variety of tool flows, such as VHDL and Verilog flow.

Requirements 1, 2 and 3 are already supported by most available operating systems. Requirement 4 has frequently been addressed by previous reconfigurable operating systems research [41] [42]. Williams and Bergmann [41] have demonstrated a partial dynamic self-reconfiguring linux system. Specifically, they introduced OS support for the internal configuration access port (ICAP) to communicate with the partially reconfigurable area of the FPGA. Santambrogio *et al.* [42] expanded on

Williams and Bergmann's work by integrating the OS support for DPR on a multi-FPGA clustered architecture (MFCA).

Of these six requirements, the work presented in this thesis focuses on 5 and 6 since they were acknowledged by Williams *et al.* [40] as being an open area of research, specially by current embedded OS offerings such as embedded linux [8].

2.4 Previous Work on Accelerator/OS Integration

During the last decade, operating systems for FPGA-based SoCs have been investigated from a number of different angles. Early work on concepts and functions for such operating systems was quickly followed by efforts toward integrating custom hardware circuits as tasks in the mainstream operating system to reflect the heterogenous nature of the design. For an operating system to provide efficient abstraction of communication between user SW and HW accelerators, it requires mechanisms that allow user applications to access the low-level HW in a transparent and safe manner [43]. The remainder of this section discusses the related work and contrasts our contributions within this context.

Researchers have used several methods of encapsulating HW resources in order to provide abstraction for reconfigurable computing. For example, in the *BORPH* project [44], *So et al.* modified and extended a Linux kernel with a hardware interface, providing conventional UNIX inter-process communication (IPC) mechanisms to the hardware using a message passing network. *BORPH* abstracted HW units as UNIX OS processes that have access to OS services and communicate using First-in-First-out (FIFO) buffers. Similarly, Kociuszkiewicz *et al.* [45] built on top of an existing Linux OS kernel and viewed HW tasks as drop-in replacements for SW tasks by mapping them to synthesized coarse-grained PicoBlaze processor cores, which also communicate via FIFOs. Xie *et al.* [46] introduced a heterogenous multiprocessor system consisting of soft processor cores synthesized to an FPGA and running Linux, similar to Kociuszkiewicz *et al.*. The OS integration was limited to FIFO communications as well. Xie *et al.*'s work was a continuation of Williams and Bergmann's work [47] [48], presenting software wrappers to encapsulate access to hardware modules in the form of a so-called "*ghost processes*", which provided a transparent interface for integration from the OS kernel and other processes. The authors considered sharing the same address space between hardware and software execution units as unsuitable since their work was based on processes as units of execution instead of threads or tasks. For communication between SW and HW, Williams and Bergmann used FIFOs mapped to the Linux file system as well as memory that is accessible from both software processes and a hardware process.

Another interesting effort trying to achieve portability and scalability when encapsulating custom HW is Vforce [49]. The authors provide a high-level object-oriented framework based on the VSIPL++ standard, which is a vector and image processing library API. Vforce extended the C++ version of the VSIPL++ API to make reconfigurable hardware implementations available by encapsulating them beneath a standard API so that the application itself needs no hardware-specific implementation code. While their approach does not utilize an operating system layer to leverage this abstraction, they do provide the desired level of separation between SW and HW designers.

Some researchers focused on extending the thread programming model to abstract HW accelerators from user applications. Typically, this extension requires support for synchronized communication between HW and SW tasks. The ReconOS project [6], for example, introduced an execution environment that extends the POSIX multi-threaded programming model from the SW domain to reconfigurable hardware. Also, the "HybridThreads" project [50] 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. [51] 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 [42] [52] [53] [54]. In particular, the *MOLEN/SESAME* project [53] [54] 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. [52] 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. [42] introduced a run-time environment that is able to dynamically place and/or remove HW tasks on demand. They use online partialbitstream manipulation for proper placement of tasks on a multi-FPGA system.

As previously mentioned, adding OS support to virtualize HW resources in FPGA-based SoC is challenging because they have greater heterogeneity and their resources can be dynamically reconfigured at run-time. SW designers targeting these 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). Based on this premise, the *FUSE* framework presented in this thesis gives a unified view of available processing resources as well as communication between processor(s) and HW accelerators. Unlike BORPH [44] and Kociuszkiewicz *et al.* [45], FUSE does not fix the type of physical link used for communication. Furthermore, FUSE virtualizes HW accelerators as tasks, instead of processes like BORPH [44] and Xie *et al.*'s [46] work.

The three closest works to the work presented in this thesis are the ReconOS [6] project, the work by Compton *et al.* [51] [52], and Vforce [49]. We use a shared memory model as recommended in [6] to reduce data communication overhead between SW and HW tasks. As outlined in [6], 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 [6], which uses a statically loaded abstraction layer, our FUSE framework 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. Alternatively, the task is either allowed to run in SW or migrated fully to HW, to be able to run from start to finish. 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 [51] work, where the OS support is registered at OS boot-time, our FUSE framework 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 [51] has been implemented within a simulation environment, our work has been prototyped on a FPGA platform to verify its feasibility.

Vforce's [49] approach, on the other hand, is quite similar to our proposed framework from two perspectives. Firstly, one layer in their framework interacts with the low-level API calls rather than the final users code, insulating the application from hardware changes. Our FUSE framework uses a low-level device driver to access the HW accelerator; the OS system calls used to communicate with this device driver are hidden from the SW designer through the FUSE API. Secondly, Vforce allows HW designers, to create specialized hardware library components that can be interfaced with their framework. We view this approach as complementary to ours as a library of HW accelerators can be created pertaining to a specific application and its FUSE OS support is integrated.

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" [50]. FUSE also incorporates the idea of tasks as units of execution across the HW boundary but, unlike [50] and [6], 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 complementary 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 [42] [52] [53] [54]. We view the work presented in this thesis as complementary to these projects. Instead, the objective of the work presented in this thesis is to provide a framework for OS *abstraction* of the underlying architectural configuration to run hardware tasks, as opposed to a new scheduling algorithm.

Chapter 3

Proposed FUSE Framework

This chapter outlines our proposed Front-end USEr (FUSE) framework. FUSE's main goal is to provide abstraction for HW accelerators from SW designers. FUSE comprises multiple components to achieve such abstraction. In this chapter, we detail the concepts behind the framework's components. We then describe the implementation of FUSE in Chapter 4.

3.1 FUSE Framework Overview

Operating systems typically support concurrency using multithreading programming models. In addition, recent concurrency programming models recommend that the SW designer decompose an application into multiple units of execution called *tasks*, which are defined as an indivisible sequence of operations that can be executed independent of other code. Therefore, a *SW task* implies a task that is designed to be executed in SW on a processor. On the other hand, a hardware accelerator is commonly used to replace a commonly executed function. Hence, we use the term

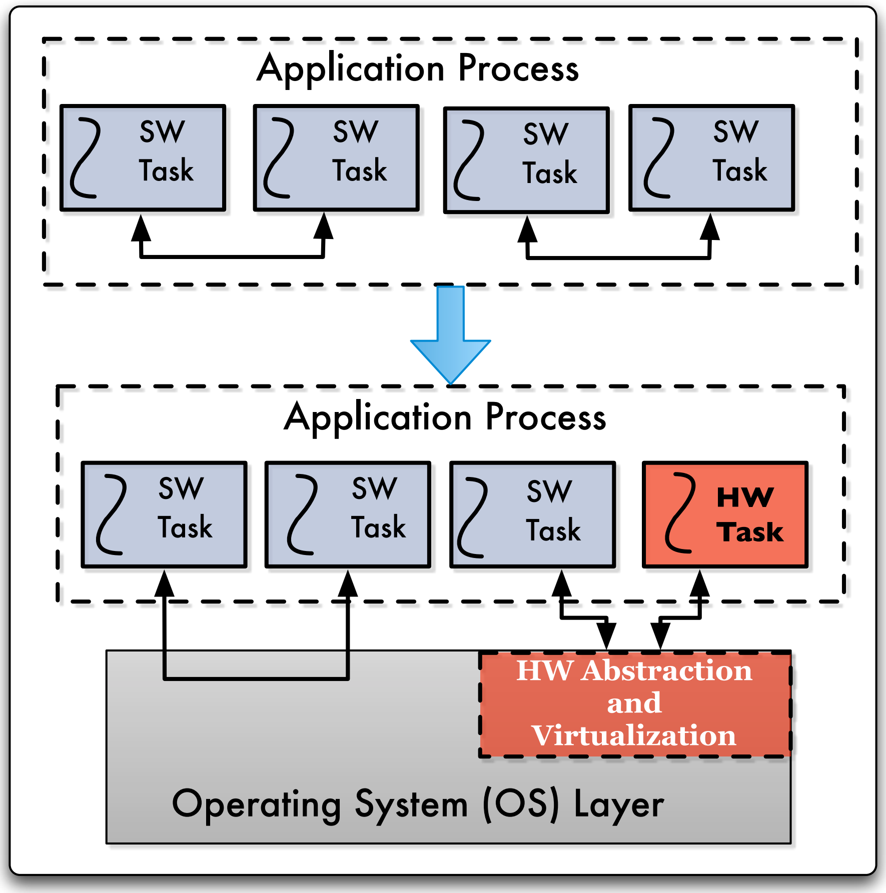


Figure 3.1: A high-level view of HW Accelerators as HW tasks

HW Task to refer to an execution flow implemented using a hardware accelerator.

The key concept behind FUSE is its use of multithreading programming models to provide SW designers with access to HW accelerators, enabling them to be viewed as additional computing resources to the CPU. Our FUSE framework abstracts HW accelerators along with their OS support as *HW tasks*, sharing the resources available to SW tasks, as illustrated in Figure 3.1. FUSE's new higher-level of abstraction also allows SW designers to create their applications with no knowledge of the platform's

HW accelerators (i.e. architecture and availability). This allows SW designers to program using familiar multithreading programming model's API function calls and also allows FUSE to be integrated into systems even when there are no HW accelerators available. This enables fast prototyping and better testing of co-designed applications.

3.2 FUSE Framework Organization

FUSE is comprised of two main components:

1. The *Top-Level FUSE Component* (TLFC): provides an application programming interface (API), which includes function calls to create and destroy SW/HW tasks. They act as wrappers for the chosen multithreading model's "create" and "destroy" functions to augment their abilities to support HW tasks when the FPGA-based SoC contains HW accelerators.

2. The *Low-Level FUSE Component* (LLFC): provides the low-level OS support that abstracts communication between the hardware accelerator and the TLFC.

Figure 3.2 shows FUSE's two main components. They span the OS's user and kernel layers so as to isolate changes to user and kernel SW from each other, and facilitate portability. Sections 3.2.1 and 3.2.2 discuss in detail the concepts behind these components.

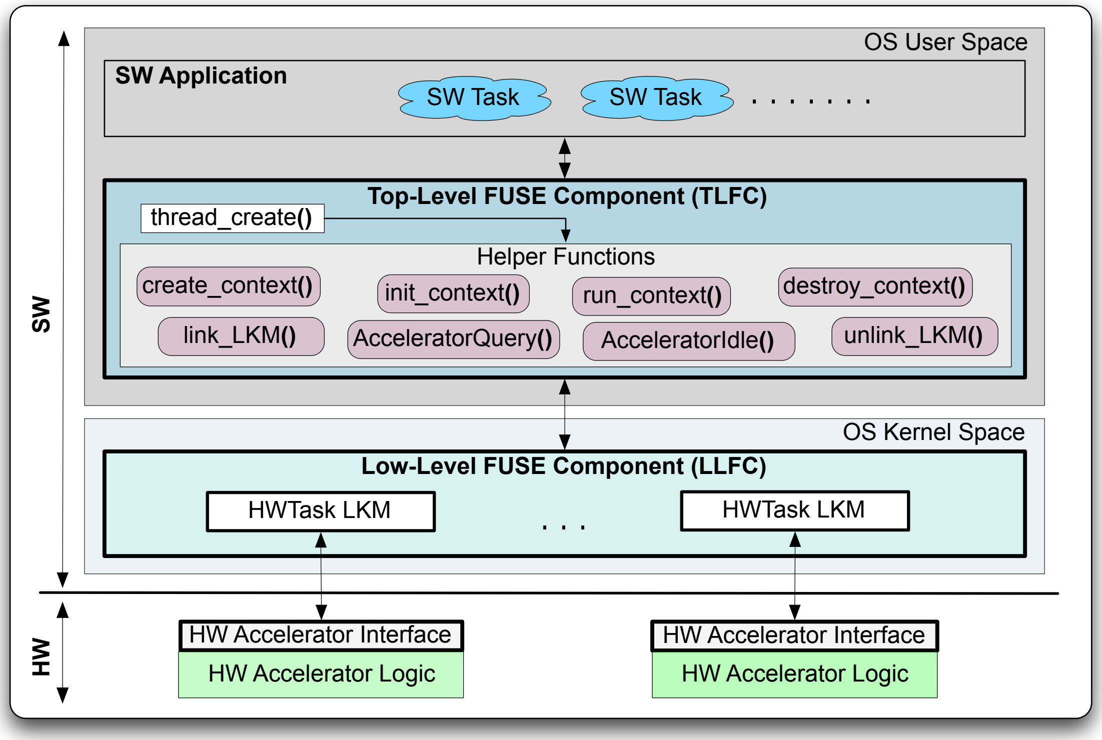


Figure 3.2: FUSE System Architecture

3.2.1 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 SW library that SW designers include in their applications to act as a wrapper for their existing multithreading model's library. In particular, the TLFC user-space library includes equivalent wrapper functions for creating and destroying HW tasks in addition to SW tasks on FUSE-enabled platforms. This provides a clean

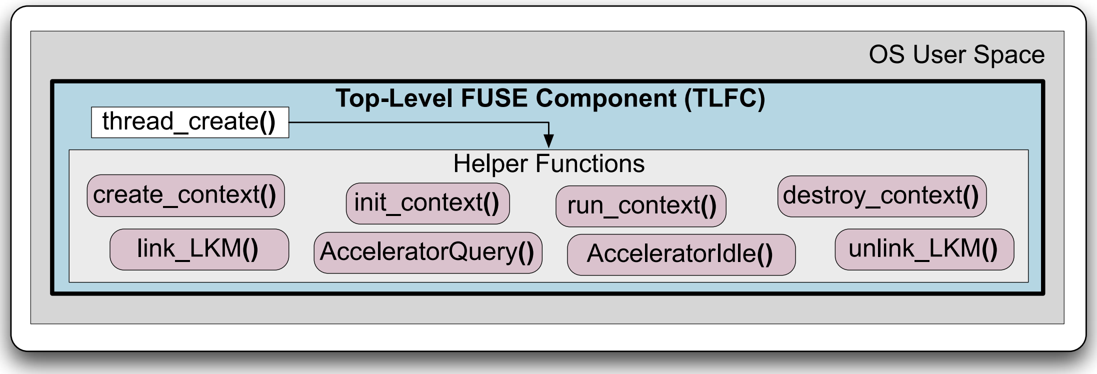


Figure 3.3: FUSE Top-Level Component

and simple way to utilize existing HW accelerators. Additionally, FUSE does not introduce incompatible changes 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.

The TLFC contains “helper functions” as shown in Figure 3.3. They are used to communicate. These helper functions are used to communicate with the *Low-Level FUSE Component* (see Section 3.2.2). 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. They create, initialize, run and destroy contexts; they also dynamically load/unload the necessary low-level OS support for a HW accelerator to the OS kernel. The low-level OS support for a HW accelerator is called a Loadable Kernel Module (LKM), for which we provide an overview in Section 3.2.2 and discuss in detail in Chapter 4.

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 by the SW designer in order to use

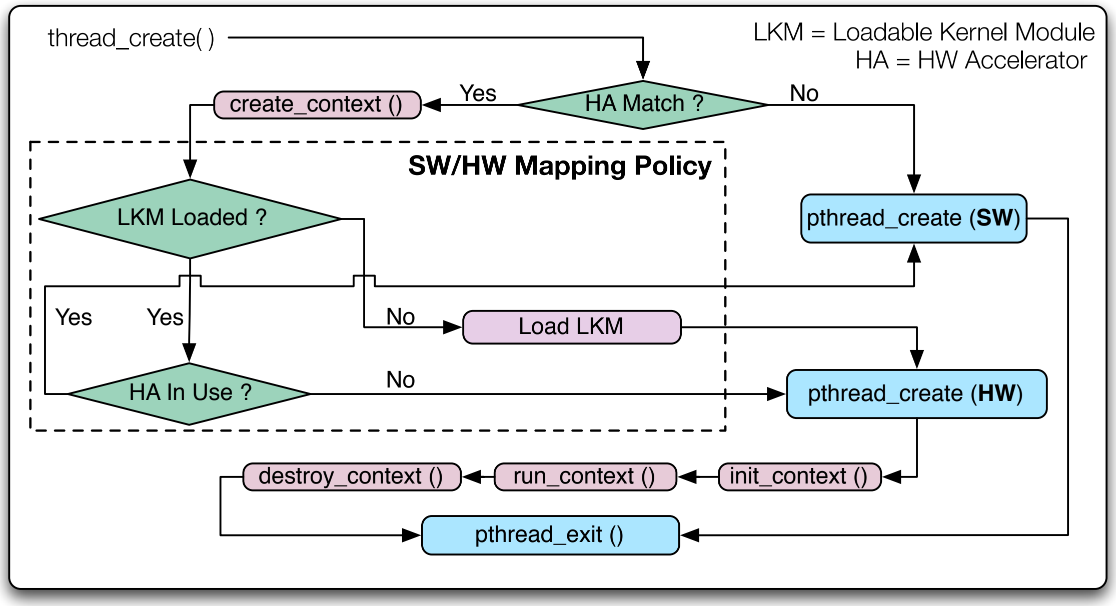


Figure 3.4: Decision Flow for Top-Level FUSE Component

FUSE. We discuss the implementation details of these function calls and outline what changes SW designers are required to perform in Chapter 4.

In a FUSE-enabled application running on platform that contains HW accelerators, FUSE must provide a policy to determine if a task is run in SW, as a SW task, or on a HW accelerator as a HW task. Figure 3.4 illustrates an example of a policy that can be used. However, designers can easily replace the simple mapping policy shown here (see the dashed box in Figure 3.4) with any of the existing algorithms for mapping tasks to SW or HW (e.g. [52] [53]). 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 3.4, 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 (e.g. *pthread_create()*). However, if an accelerator exists, then a check is made to see if its corresponding OS support (i.e. the 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 support for dynamic partial reconfiguration (DPR) of HW accelerators, this mapping policy (See dashed box in Figure 3.4) can be altered to check and see if an additional HW accelerator's logic can be instantiated dynamically using DPR. All the necessary OS support for reconfigurable computing systems using DPR (e.g. [42]) currently exists in FUSE.

3.2.2 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 3.5). It exposes HW accelerators on the FPGA's fabric to the TLFC using *runtime* loadable device drivers (i.e. Loadable Kernel Modules, LKMs) [55]. 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 3.2.1, each HW accelerator attaches to the system via a HW interface (discussed in chapter 4) that is accessible by its own LKM (see

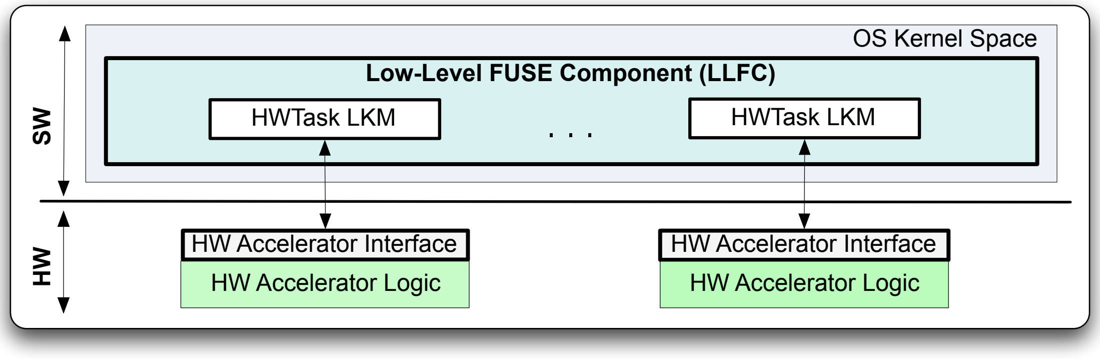


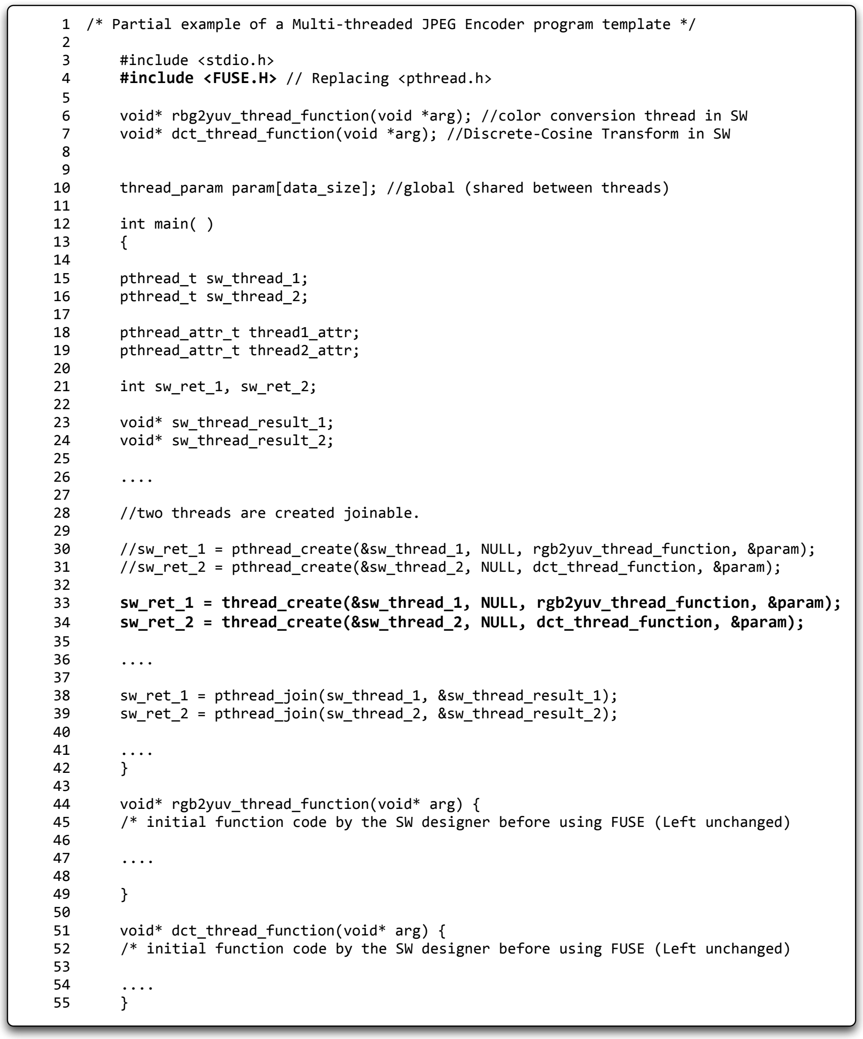
Figure 3.5: FUSE Low-Level Component

Figure 3.5). 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 affect 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 source code can be modified 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.

Chapter 4

FUSE Implementation and Integration

In this chapter, we describe how we implemented 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 [8] as our run-time environment and POSIX threads (Pthreads) [56] [57], a widely used multithreading programming model, as the multithreading model of choice. Using a SoC with three example HW accelerators (See Chapter 5), this chapter provides a top-down discussion of the integration of FUSE API in user SW (Section 4.1) and how it abstracts the underlying architecture (Sections 4.2 and 4.3).



```
/* Partial example of a Multi-threaded JPEG Encoder program template */

    #include <stdio.h>
    #include <FUSE.H> // Replacing <pthread.h>

    void* rbg2yuv_thread_function(void *arg); //color conversion thread in SW
    void* dct_thread_function(void *arg); //Discrete-Cosine Transform in SW


    thread_param param[data_size]; //global (shared between threads)

    int main( )
    {

    pthread_t sw_thread_1;
    pthread_t sw_thread_2;

    pthread_attr_t thread1_attr;
    pthread_attr_t thread2_attr;

    int sw_ret_1, sw_ret_2;

    void* sw_thread_result_1;
    void* sw_thread_result_2;

    ....

    //two threads are created joinable.

    //sw_ret_1 = pthread_create(&sw_thread_1, NULL, rgb2yuv_thread_function, &param);
    //sw_ret_2 = pthread_create(&sw_thread_2, NULL, dct_thread_function, &param);

    sw_ret_1 = thread_create(&sw_thread_1, NULL, rgb2yuv_thread_function, &param);
    sw_ret_2 = thread_create(&sw_thread_2, NULL, dct_thread_function, &param);

    ....

    sw_ret_1 = pthread_join(sw_thread_1, &sw_thread_result_1);
    sw_ret_2 = pthread_join(sw_thread_2, &sw_thread_result_2);

    ....
    }

    void* rgb2yuv_thread_function(void* arg) {
    /* initial function code by the SW designer before using FUSE (Left unchanged)

    ....

    }

    void* dct_thread_function(void* arg) {
    /* initial function code by the SW designer before using FUSE (Left unchanged)

    ....
    }
```

Figure 4.1: Partial source-code of an application using FUSE

4.1 User-space Implementation of TLFC

The top-level FUSE component (TLFC) contains the FUSE API header library. This library is used by the SW designer when writing multithreaded applications. For example, Figure 4.1 provides an code template for an example multithreaded SW application written with the FUSE header library support included. This template code includes the FUSE header library <*fuse.h*> (Line 4) and creates two threads of execution using FUSE's *thread_create()* (Lines 33 and 34), which acts as a wrapper for the original *pthread_create()* (Lines 30 and 31). The four parameters for *pthread_create()* are: the thread identifier (e.g. *sw_thread_1*), the thread attributes object (e.g. *thread1_attr* or NULL is used to use default thread attributes), the thread function name (e.g. *dct_thread_function*), and a single argument for the specified thread function (e.g. the *&param* data structure). The *thread_create()* function utilizes a set of internal helper functions that incorporate additional policies to enable transparent migration of tasks to HW accelerator(s). The rest of this section will describe those helper functions (Section 4.1.1), before discussing the thread creation procedure in detail (Section 4.1.2).

Table 4.1: FUSE Helper Functions

Name	Prototype	OS SysCalls Used
CreateContext()	*void CreateContext(cs_t* cs);*	-
InitContext()	*void InitContext(cs_t* cs);*	*open()*
		mmap()
RunContext()	*void RunContext(cs_t* cs);*	*IOCTL()*
		**ptr = value*
		*value = *ptr*
DestroyContext()	*void DestroyContext(cs_t* cs);*	*munmap()*
		close()
AcceleratorQuery()	*char* AcceleratorQuery(void* fname);*	*popen()*
		fgets()
		fclose()
AcceleratorIDLE()	*bool AcceleratorIdle(char* hw_task);*	*IOCTL()*
LinkLKM()	*int LinkLKM(char* hw_task);*	*modprobe()*
UnlinkLKM()	*int UnlinkLKM(char* hw_task);*	*rmmod()*
LKMLoaded()	*bool LKMLoaded(char* hw_task);*	*lsmod()*

4.1.1 FUSE API Helper Functions

FUSE uses a set of helper functions to manage thread creation in SW or HW. Table 4.1 lists all the helper functions implemented within the FUSE library. Hidden from the user, these helper functions are organized into three categories according to their purpose; context handling functions, accelerator search functions, and LKM handling functions. They are described below:

Context Structure Functions

This category includes *create_context()*, *init_context()*, *run_context()*, and *destroy_context()*. These functions operate on a storage object, defined inside the FUSE library, called a context structure (e.g. *cs_t*). Figure 4.2 shows the definition of the context structure within the FUSE library. This object stores the HW accelerator name (Line 2), the LKM name (Line 3), and the file descriptor (Line 6). It also includes two boolean flags used to indicate if the LKM is loaded (Line 4) and whether or not the accelerator is in use (i.e. IDLE or BUSY/RUNNING) (Line 5). Additionally, the structure keeps an internal buffer for data communications with the HW accelerator (Lines 8 and 9) and also the number of accelerators being used (Line 7).

Accelerator Query/Status Functions

This set of functions is used to match a SW function name with an available accelerator and its LKM. In particular, FUSE uses the *AcceleratorQuery()* function to match a SW task name to a HW Accelerator using the accelerator look-up table stored in the OS. *AcceleratorIdle()* is used two ways; first to check whether or not an available accelerator is currently in use; second, to set the STATUS register in the HW

```
 1  typedef struct {
 2      char* HA_name;
 3      char* LKM_name;
 4      bool LKM_LOADED; //true=loaded, false=not loaded
 5      bool HA_IN_USE; //true=in_use, false= not in_use
 6      int HA_fd;                /* descriptor for the HW Accelerator */
 7      int dev_cnt;               /* device count */
 8      int* cs_bufptr;          /* next unread byte in internal buf */
 9      int cs_buf[CS_BUFSIZE]; /* internal buffer (to be used for data transfer (Read/Write)) */
10  } cs_t; //context structure type used be helper functions
```

Figure 4.2: Partial code showing the context structure definition

accelerator interface (See Section 4.3).

LKM Link/Unlink/Loaded Functions

LinkLKM() and *UnlinkLKM()* are used by FUSE to load and unload the loadable kernel module support for a given HW accelerator. These two helper function are simply wrappers for the *modprobe()* and *rmmod()* system calls in the kernel. The *LKMLoaded()* function uses the *lsmod()* system call to set the *LKM_Loaded* flag in the context structure (see Figure 4.2 (Line 4)) to either true or false and then returns the result.

4.1.2 Thread Creation

The POSIX Pthreads API [56] [57], accessed via <*pthreads.h*>, contains an extensive list of functions that allows SW designers to add concurrency into their application code in addition to providing synchronization. Table 4.2 shows a list of the most important OS objects and the respective API functions as provided by the POSIX standard API [57]. Table 4.2 also shows the wrapper functions provided by the FUSE API for each POSIX API function mentioned. While FUSE provides a wrapper for

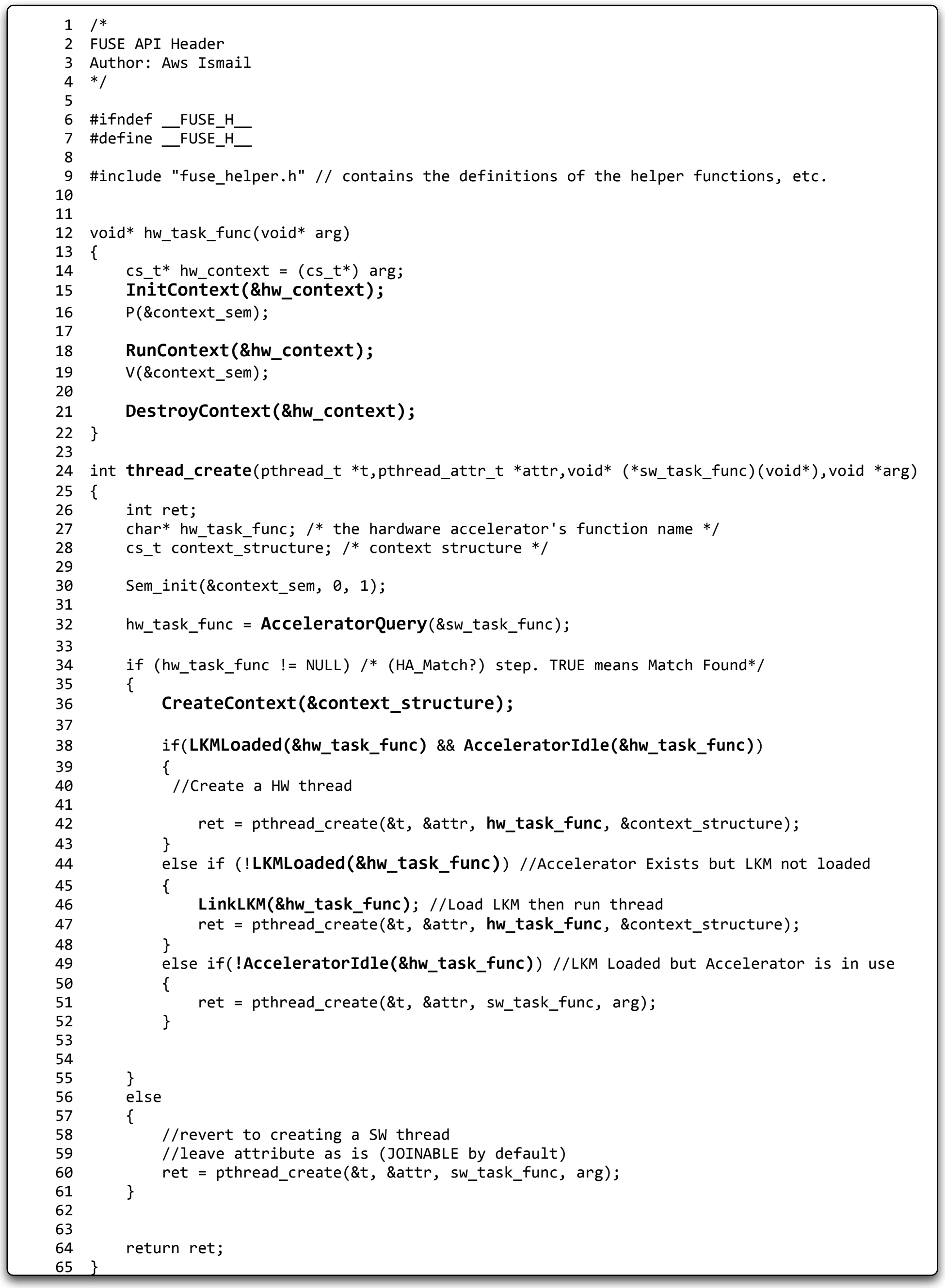
Table 4.2: Overview of the key POSIX API functions vs. FUSE API

OS Object	**POSIX API** [57]	**FUSE API**
Thread Creation	*pthread_create()*	*thread_create()*
Semaphores	*sem_post()*	*P()*
	sem_wait()	*V()*
Mutexes	*pthread_mutex_lock()*	*thread_mutex_lock()*
	pthread_mutex_unlock()	*thread_mutex_unlock()*
Shared Memory	**ptr = value*	**ptr = value*
	*value = *ptr*	*value = *ptr*

most of those functions (e.g. semaphores, mutexes, shared memory), the abstraction of SW/HW thread creation (i.e. *thread_create()*) is the most important function supported by FUSE as it creates new concurrent threads of execution.

During the OS startup, existing HW accelerators are detected by the OS, which registers their names and base addresses for ease of retrieval at runtime. In addition, the low-level OS support (i.e. LKMs) (See Section 4.2) that are statically compiled within the OS image are loaded. New LKMs can be loaded or removed at runtime as desired. Information about existing HW accelerators is saved into a look up table (See Section 4.2) stored within the OS kernel space, 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 designer 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



```
/*
FUSE API Header
Author: Aws Ismail
*/

#ifndef __FUSE_H__
#define __FUSE_H__

#include "fuse_helper.h" // contains the definitions of the helper functions, etc.


void* hw_task_func(void* arg)
{
    cs_t* hw_context = (cs_t*) arg;
    InitContext(&hw_context);
    P(&context_sem);

    RunContext(&hw_context);
    V(&context_sem);

    DestroyContext(&hw_context);
}

int thread_create(pthread_t *t,pthread_attr_t *attr,void* (*sw_task_func)(void*),void *arg)
{
    int ret;
    char* hw_task_func; /* the hardware accelerator's function name */
    cs_t context_structure; /* context structure */

    Sem_init(&context_sem, 0, 1);

    hw_task_func = AcceleratorQuery(&sw_task_func);

    if (hw_task_func != NULL) /* (HA_Match?) step. TRUE means Match Found*/
    {
        CreateContext(&context_structure);

        if(LKMLoaded(&hw_task_func) && AcceleratorIdle(&hw_task_func))
        {
         //Create a HW thread

            ret = pthread_create(&t, &attr, hw_task_func, &context_structure);
        }
        else if (!LKMLoaded(&hw_task_func)) //Accelerator Exists but LKM not loaded
        {
            LinkLKM(&hw_task_func); //Load LKM then run thread
            ret = pthread_create(&t, &attr, hw_task_func, &context_structure);
        }
        else if(!AcceleratorIdle(&hw_task_func)) //LKM Loaded but Accelerator is in use
        {
            ret = pthread_create(&t, &attr, sw_task_func, arg);
        }


    }
    else
    {
        //revert to creating a SW thread
        //leave attribute as is (JOINABLE by default)
        ret = pthread_create(&t, &attr, sw_task_func, arg);
    }


    return ret;
}
```

Figure 4.3: *thread_create()* definition inside <*fuse.h*> header file.

of <*pthread.h*>.

Figure 4.3 shows part of the API implementation for the *thread_create()* function in the FUSE header library. As part of the *thread_create()* function, FUSE uses several internal "helper functions," that exist in the *Top-Level FUSE Component* (See Table 4.1), to manage thread creation. This implementation of *thread_create()* 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 36, 15, 18 and 21 respectively in Figure 4.3).

When creating a thread, the aforementioned accelerator look-up table is checked via *AcceleratorQuery()* helper function (Line 32) to find a matching accelerator name. Then FUSE uses *create_context()* (Line 36) to allocate a context structure (Line 28) to hold information related to the thread, such as the function name, user data, and current state. The *LKMLoaded()* and *AcceleratorIdle()* (Line 38) functions assert whether the matched accelerator can be locked to this context. If an accelerator exists, FUSE checks whether or not its low-level support (i.e. LKM) is loaded and also whether or not the accelerator is currently in use by other applications. These checks mimic the mapping policy shown in Chapter 3 (see Figure 3.4). Ideally, the accelerator will be idle with its LKM already loaded. FUSE then creates a thread that performs HW task initialization and execution (Line 42). If the accelerator is idle but its LKM is not loaded then FUSE first loads the LKM support through calling *LinkLKM()* function (Line 46) and then proceeds with the HW task initialization (Line 47). *LinkLKM()* loads the corresponding LKM for that particular accelerator using the *modprobe()* system call provided by the OS [55]. This system call is made only once when the accelerator is first used.

This newly created thread will use *init_context()* (Line 15) and *run_context()* (Line 18) to run the HW accelerator. *init_context()* fills the context structure with the related information (fetched from the accelerator look-up table). 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 18) 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 *ioctl()* system call for sending/receiving control signals (e.g. set the STATUS register within the HW accelerator interface to RUNNING or BUSY) in addition to simple array dereferencing of the shared-memory space when sending/receiving data values. Finally, FUSE calls *destroy_context()* (Line 21) to deallocate the context structure, set the STATUS register within the HW accelerator to IDLE, release the shared memory, and close the device file handler. This is done after the HW accelerator is finished processing any remaining data.

4.2 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. Kernel modules are pieces of code that extend the functionality of the kernel without the need to reboot the system (i.e.

while the kernel is already in memory and executing). For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. The *Low-Level FUSE Component* (LLFC), is a collection of Loadable Kernel Modules (LKMs) that add support for HW accelerators to the OS. The 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 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 bus. The rest of this section will give an overview of the LKM's architecture (Section 4.2.1), the set of mechanisms it exports to enable abstraction (Section 4.2.2), and the way it binds to a HW accelerator device when initially loaded (Section 4.2.3).

4.2.1 LKM Architecture

As mentioned in Section 3.2.2, each LKM can be compiled in such a way that it is either loaded automatically during OS startup, or is loaded by FUSE at run-time after the OS boots. During thread creation, FUSE handles both cases by offering to check whether or not the LKM is loaded and also to load or unload the LKM on demand. (See *LKMLoaded()*, *LinkLKM()* and *UnlinkLKM()*, in Section 4.1.1). A HW accelerator's LKM implements miscellaneous device driver functionality to treat 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.

```
// Module information.
MODULE_LICENSE("SFU");
MODULE_DESCRIPTION("hwtask_dct_adapter");
#define DRIVER_NAME "hwtask_dct_adapter"
#define DRIVER_NAME_ID(id) DRIVER_NAME#id

// Version information.
int hwthread_dct_major = MISC_MAJOR;
module_param(hwtask_dct_major, int, 0);
static int dev_id = 0;

#define                BUFSIZE 10  // Slave registers in HWtask_DCT PLB IP core.
static LIST_HEAD       (inst_list);
static DECLARE_RWSEM   (inst_list_sem);

// Structures.
struct         of_platform_driver  hwtask_dct_driver;
static struct  file_operations     hwtask_dct_fops;
struct         hwtask_dct_local;
static struct  of_device_id        __devinitdata hwtask_dct_match[];

struct of_platform_driver hwtask_dct_driver = {
    .driver = {
       .name =      DRIVER_NAME,
       .owner =     THIS_MODULE,
       .of_match_table =    hwtask_dct_match,
    },
    .probe =        hwtask_dct_probe,
    .remove =       __devexit_p(hwtask_dct_remove),
};

static struct file_operations hwtask_dct_fops = {
    .open =         hwtask_dct_open,
    .close =        hwtask_dct_close,
    .read =         hwtask_dct_read,
    .write =        hwtask_dct_write,
    .mmap =         hwtask_dct_mmap,
    .ioctl =        hwtask_dct_ioctl,
};

struct hwtask_dct_local {
    struct list_head    link;       // For the linked list of instances.
    unsigned long       base_phys;  // IP core base address.
    unsigned int        base_addr;  // Virtual address for the reg file IO MEM resource.
    unsigned long       remap_size;
    u32         device_id; //ID counter for multiple instances of the same accelerator
    int         is_inuse; //flag used by get_instance()
    unsigned int        buf[BUFSIZE];   // The register file (10 slave regs, 32-bit each).
    struct miscdevice   *miscdev;
    wait_queue_head_t   wait;       // Wait queue for blocking I/O read or write.
};

static struct of_device_id __devinitdata hwtask_dct_match[] = {
    { .compatible = "xlnx,hwtask-dct-1.00.a", },
    { /* end of list */ },
};
MODULE_DEVICE_TABLE(of, hwtask_dct_match);
```

Figure 4.4: Partial code showing part of the DCT LKM implementation

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 manipulated using the traditional file API (e.g. *open()*, *close()*, *read()*, *write()*, etc.). Specifically, the LKM exports its own implementation of these file operations (discussed in Section 4.2.2) that enables the TLFC to communicate with the HW accelerator using common file API system calls. The LKM uses a local data structure which contains the physical base address of the memory-mapped device. For example, Figure 4.4 shows part of the LKM code for a DCT HW Accelerator. The data structure is called *hwtask_dct_local* (Line 41). This structure allows the kernel-space side of FUSE, the LLFC, to recognize each HW accelerator by its base physical address (Line 43) in the address space map.

The LKM is designed so that it can handle multiple instances of the same HW accelerator. To achieve this, the LKM uses a linked list, called *inst_list* (Line 13) to store multiple instances of the local data structure (*hwtask_dct_local*); one for each accelerator. Also, each local data structure has a device ID (Line 46), which assigns a unique ID number, starting from 0, for each accelerator handled by that LKM. This ID number is appended to the *DRIVER_NAME* (Line 5) during the registration and initialization step (discussed in Section 4.2.3). Additionally, the LKM can handle different architectural versions of a given HW accelerator. This is done by keeping a list of matching versions of the HW accelerator (Line 53).

4.2.2 LKM File Operations

The bulk of the LKM implementation involves the definitions of the file operation functions, which the LKM exports to the user space and are in turn used by the helper functions inside the top-level FUSE component. As shown in Figure 4.4, each LKM provides services for *open*, *close*, *read*, *write*, *mmap*, and *ioctl* system calls (Lines 33 to 38). These calls are used in the implementation of the helper functions, *init_context()*, *run_context()* and *destroy_context()*, in the user-space part of the FUSE framework (i.e. TLFC) (See Table 4.1).

4.2.3 LKM Registration/Initialization Procedure

Whether or not the LKM is loaded at run-time or during the OS startup phase, the LKM follows a specific initialization procedure upon loading. The LKM uses internal kernel macros to notify the kernel of its entry point and exit point functions. For example, in Figure 4.5, the DCT LKM specifies *hwtask_dct_init()* as the LKM's entry point (Line 26) via the *module_init()* kernel macro and *hwtask_dct_exit()* as its exit point (Line 27) using the *module_init()* macro. If the LKM were compiled statically into the kernel image, the *hwtask_dct_init()* function would be stored in the kernel image and run during OS boot. Otherwise, the kernel invokes *hwtask_dct_init()* when the module is dynamically loaded (e.g. using *LinkLKM()* in the TLFC).

The *hwtask_dct_init()* function (Line 2) calls the *of_register_platform_driver()* kernel function (Line 7) to perform the registration process. *of_register_platform_driver()* uses the *hwtask_dct_driver* structure, defined at the beginning of the LKM (See Figure 4.4 (Line 22)), as a parameter. This structure notifies the kernel of the LKM name, and the probe function to invoke. The probe function *hwtask_dct_probe* (See

```

static int __init hwtask_dct_init(void) {

    int res;
    printk(KERN_ALERT "Loading module... ***********************\n");

    res = of_register_platform_driver(&hwtask_dct_driver);

    if (res != 0)
    printk(KERN_ALERT
        "***Loading failed... ********************************\n\n");
    else printk(KERN_ALERT
        "Loading was successful... ***************************\n\n");
    return res;
}

static void __exit hwtask_dct_exit(void) {

    printk(KERN_ALERT "Unloading module... *********************\n");

    of_unregister_platform_driver(&hwtask_dct_driver);

    printk(KERN_ALERT "Unloading module finished... ************\n");
}

module_init(hwtask_dct_init);
module_exit(hwtask_dct_exit);
```

Figure 4.5: The DCT LKM's entry/exit points functions

Figure 4.4 (Line 28)) performs the bulk of the registration procedure. Figure 4.6 illustrates in detail the steps taken by the probe function within an LKM, using the DCT LKM as an example. The sequence of steps illustrated in Figure 4.6 are detailed below:

1. Allocate the local data structures, find the matching HW accelerator(s), and set the device name.

2. (a) Evaluate the I/O address space for the device registers using the *platform_get_resource()* kernel function.

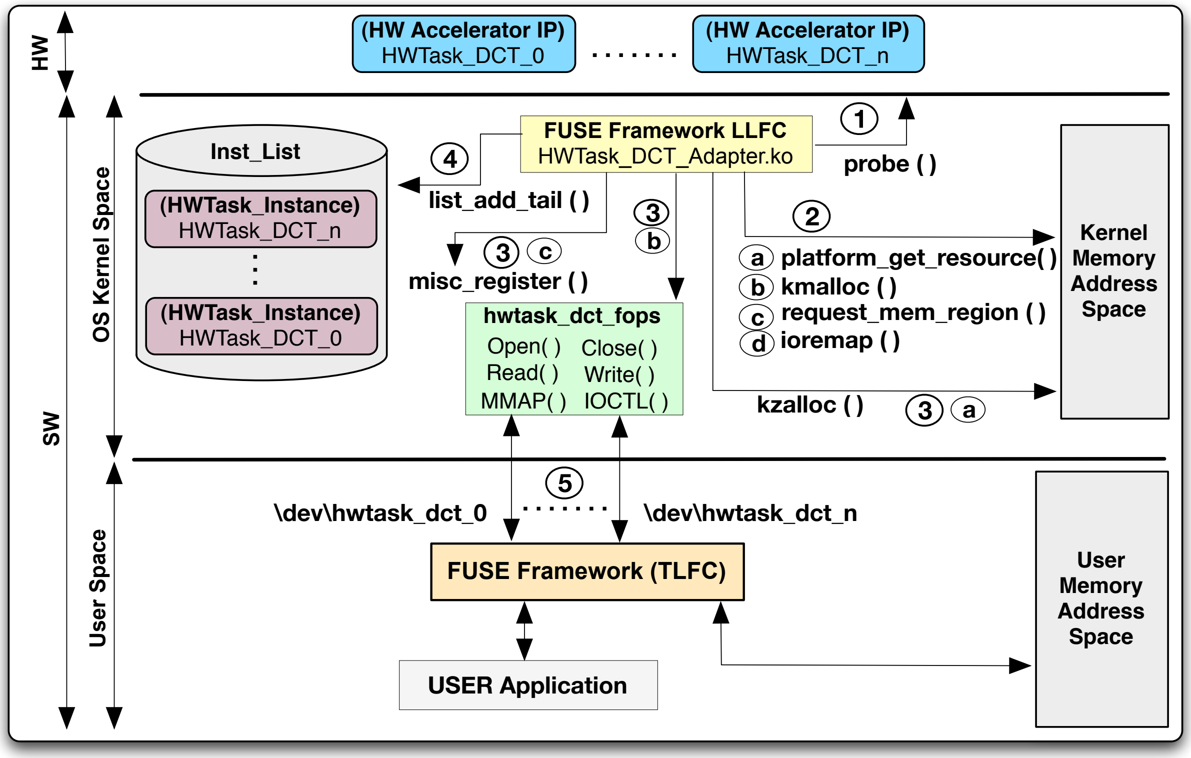

HW
SW
OS Kernel Space
User Space
(HW Accelerator IP) HWTask_DCT_0
(HW Accelerator IP) HWTask_DCT_n
Inst_List
(HWTask_Instance) HWTask_DCT_n
(HWTask_Instance) HWTask_DCT_0
FUSE Framework LLFC HWTask_DCT_Adapter.ko
1 probe ( )
4 list_add_tail ( )
3 c misc_register ( )
3 b
2
a platform_get_resource( )
b kmalloc ( )
c request_mem_region ( )
d ioremap ( )
Kernel Memory Address Space
hwtask_dct_fops
Open( ) Close( ) Read( ) Write( ) MMAP( ) IOCTL( )
kzalloc ( ) 3 a
5
\dev\hwtask_dct_0
\dev\hwtask_dct_n
FUSE Framework (TLFC)
USER Application
User Memory Address Space

Figure 4.6: The DCT LKM's Probe() function execution procedure

(b) Allocate kernel memory for an instance of the local data structure (i.e. hwtask_dct_local) using the *kmalloc()* kernel function.

(c) Lock I/O address space into kernel memory for the HW accelerator registers using *request_mem_region()* kernel function.

(d) Get the virtual base address of the device using *ioremap()* kernel function.

3. (a) Register the new miscellaneous device by first Allocating memory for it (*kzalloc()*).

   (b) Fill in the name and the list of file operation functions (i.e. hwthread_dct_fops).

   (c) Call the *misc_register()* function to register the device as a miscellaneous platform device in the kernel.

4. Add the new device local structure to the local device list using *list_add_tail()* kernel function.

5. Publish the device as a device file entry to the user space (i.e. */dev/hwtask_dct_0*).

After successful registration, the TLFC's helper functions (e.g. *InitContext()*, etc.) can use the file operations exported by this LKM (e.g. *open()*, etc.) to establish communication with the HW accelerator.

4.3 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 3.2). To demonstrate how FUSE enables HW accelerators with LKM support to mimic SW threads, we used the example HW accelerator interface shown in Figure 4.7 for the shared-bus used in the experimental system discussed in Chapter 5.

Our example HW accelerator interface is comprised of a *Bus Interface*, which performs the appropriate address decoding and signal control for the system communication network, and the accelerator's *Function Interface* (see Figure 4.7). 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

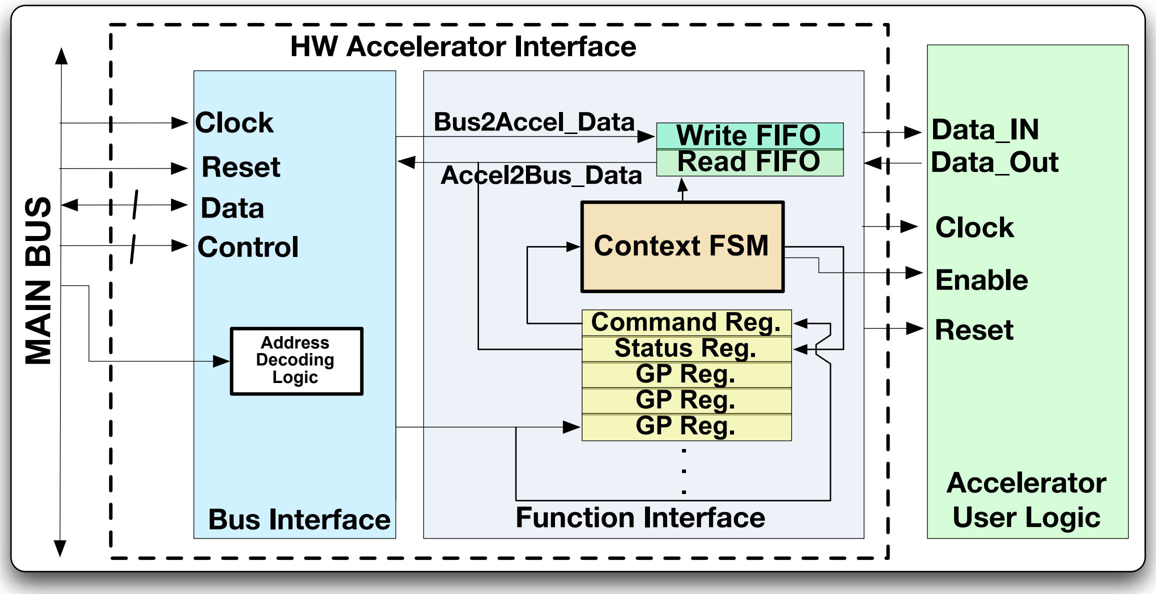


Figure 4.7: HW Accelerator Interface

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 uses the *AcceleratorIdle()* helper function to read the *Status Register*. In the TLFC implementation described in Section 4.1, the *run_context()* helper function uses LKM-supported system calls (e.g. *IOCTL()*) 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; however, our current experimental system does not use DMA, which has been left for future work (See Chapter 6).

Chapter 5

Evaluation and Experimental Results

This chapter outlines the experimental system used to demonstrate and evaluate the FUSE framework. We discuss three SW case studies that utilize HW accelerators accessed via FUSE. In Section 5.1, we describe the experimental system architecture and quantify the resource utilization the HW accelerators with respect to the entire system. In Sections 5.3.1 and 5.3.2, we discuss the run-time overhead incurred from the Low-Level FUSE Component (LLFC) and the Top-Level FUSE Component (TLFC), which were both outlined in Chapter 4. In Section 5.2, we first explain our testing methodology and discuss the overhead incurred from the top-level FUSE component (TLFC) when migrating a SW thread to a HW accelerator. Finally, we outline the performance speedups obtained from using FUSE in conjunction with HW accelerators for each case study.

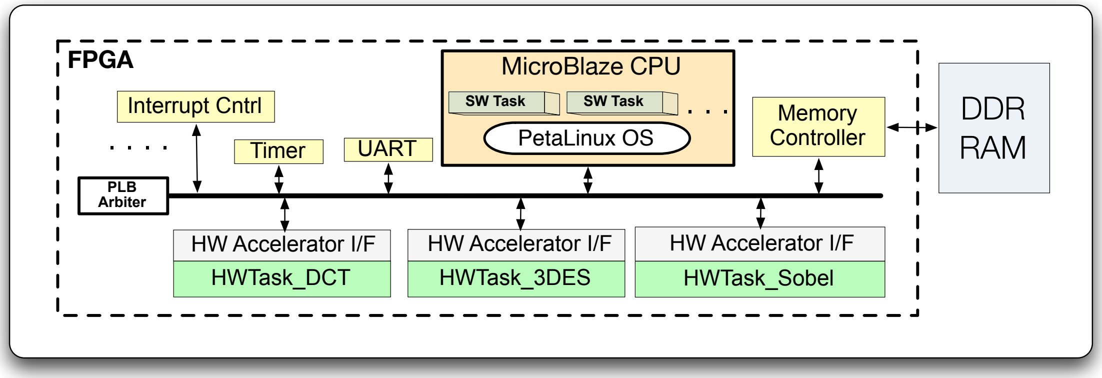


Figure 5.1: Experimental System Architecture

5.1 Hardware Experimental Setup

Figure 5.1 highlights the key components of our experimental HW system, adapted from a PetaLogix example built on a Xilinx Virtex 5 FPGA [8] using Xilinx's EDK FPGA CAD tool. 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 example HW accelerators (i.e. *HWTask_DCT*, *HWTask_3DES*, *HWTask_Sobel*) that are connected to a shared system bus through HW accelerator interfaces. The MicroBlaze CPU is configured with an Memory Management Unit (MMU), a 8KB data cache and a 8KB 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.

Three multithreaded application examples are implemented: JPEG image compression, Triple-DES encryption–and–decryption, and an image filter. Also, Three

Table 5.1: System Resource Utilization on a Virtex5 FPGA

App(HWTask)	LKM Size	F/F	LUTs	DSP48E	BRAMs
JPEG Enc.(DCT)	5.5 KB	2293 (4%)	1819 (3%)	8 (13%)	8 (5%)
Encryption (3DES)	5.0 KB	1698 (3%)	2042 (4%)	0 (0%)	0 (0%)
Image Filter (Sobel)	5.0 KB	960 (1%)	1620 (3%)	0 (0%)	0 (0%)
Overall System		20509 (29%)	18750 (27%)	54 (84%)	22(15%)

Table 5.2: Utilization on a Virtex5 FPGA for the example HW Accelerator Interfaces

HW Accelerator Interface	F/F	LUTs
HWTask_DCT I/F	803 (1%)	817 (1%)
HWTask_3DES I/F	560 (<1%)	402 (<1%)
HWTask_Sobel I/F	560 (<1%)	401 (<1%)

HW accelerators are integrated into our system via the example HW accelerator interface we discussed previously in Chapter 4. Each accelerator provides specific functionality for a given application: the JPEG image compression application uses a Discrete Cosine Transform (DCT) accelerator (*HWTask_DCT*); the Triple-DES (3DES) application uses an encryption accelerator (*HWTask_3DES*); and the image filtering application uses a SOBEL edge detection accelerator (*HWTask_Sobel*).

5.1.1 Resource Utilization

Table 5.1 outlines the resource utilization for the accelerators and their LKMs, along with the overall experimental system, on the Virtex 5 FPGA. These 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. Table 5.2, on the other hand, shows the resource utilization of the example HW accelerator interface itself (i.e. the bus interface and function interface discussed in Section 4.3). It can been seen that the example HW accelerator interface requires a maximum of 803 Flip-Flops and 817 LUTs to implement. This interface can be customized, as previously mentioned in Chapter 4, to include BRAMs and more registers to satisfy the application requirements.

5.1.2 PetaLinux OS Versions

We have demonstrated FUSE with two versions of the PetaLinux OS, versions 0.4 and 2.1 [8]. Differences specific to these versions are as follows:

PetaLinux Version 0.4:

This older version has experimental MMU support for FPGA-based systems with a MicroBlaze soft processor. However, its integration with Xilinx's EDK FPGA CAD Tool was incompatible after versions 11.x of that tool. Therefore, version 0.4 of the OS was being used with EDK version 10.1.3. This OS version also lacked support for a standardized way of adding LKMs to the PetaLinux code. Hence, when using it, we opted to integrate FUSE's LLFC within the kernel source tree directly. Furthermore, when using version 0.4 of PetaLinux, we used software-based timing measurements to quantify the impact of FUSE's overhead on our case studies.

PetaLinux Version 2.1:

This version provides an enhanced device driver support for common peripherals on the FPGA-based system as well as support for the latest FPGA CAD tools from Xilinx (version 13.1). The MMU and cache memory support are also improved. Additionally, this version now provides a standardized way to include add-on packages and LKMs to the system during the design process. This eliminated the need for changes to the kernel source tree itself. Version 2.1 of PetaLinux is used in conjunction with Xilinx's EDK version 13.1. Additionally, we opted to use hardware-based timing measurements to quantify the impact of FUSE's overhead on our case studies.

5.2 Example Case Studies

5.2.1 Testing Methodology

Our objective is to quantify the overhead of the runtime abstraction of HW accelerator(s) in the OS and its impact of the performance speedup due to acceleration. 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, when DPR is supported, which has been quantified by the vendor [28].

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 3.4); each check takes ∼400us. For these three applications, using FUSE without HW accelerators increases the runtime negligibly (<1%). When HW accelerator(s) exist, however, the overhead incurred from using

FUSE is divided into:

1. Loading the LKM the first time a HW accelerator is used (~12 ms when using PetaLinux version 0.4, and 14–18 ms when using PetaLinux version 2.1).
2. Calling the *open()*, *mmap()*, *munmap()*, and *close()* functions to initialize/uninitialize communications with the HW accelerator (~0.91ms for both versions of PetaLinux).
3. The potential unloading of the LKM when the execution finishes (~12 ms when using PetaLinux version 0.4, and ~14 ms when using PetaLinux version 2.1).

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.

5.2.2 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 task into HW.

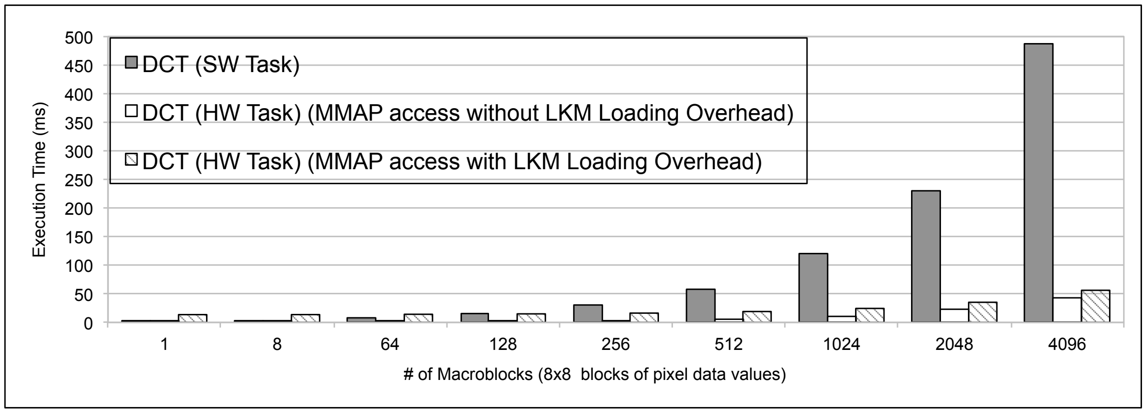


Figure 5.2: Execution Time of the JPEG Encoder application with different implementations for the DCT task (PetaLinux Ver. 0.4, SW-based timing measurement)

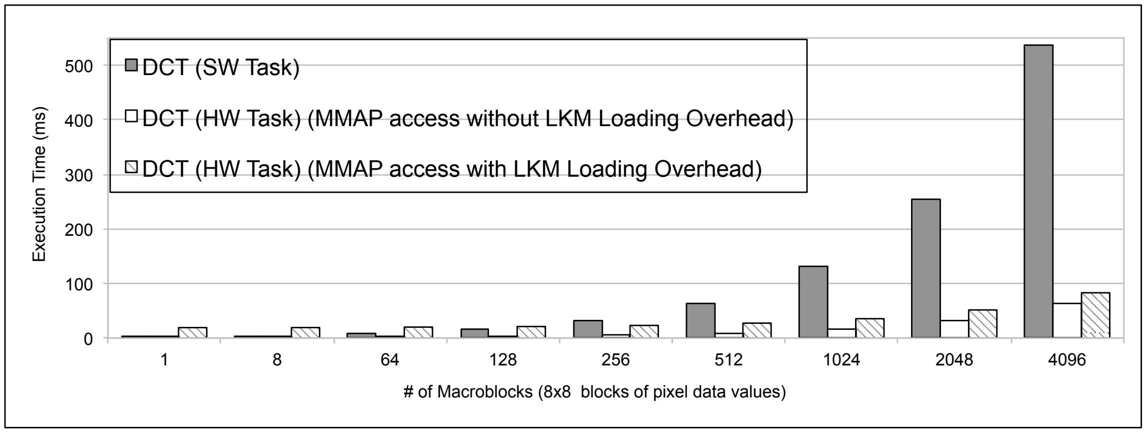


Figure 5.3: Execution Time of the JPEG Encoder application with different implementations for the DCT task (PetaLinux Ver. 2.1, HW-based timing measurement)

Figures 5.2 and 5.3 show the execution time of the JPEG encoder application when implemented using PetaLinux versions 0.4 and 2.1, respectively. Different DCT implementations (i.e. SW Task and HW Task) are shown in each figure with increasing image sizes and thus, an increased number of macroblocks. The DCT accelerator's

LKM uses a MMAP-based method of data transfer as it was found to incur the least overhead (which will be discussed in Section 5.3.2 in detail). This method establishes a direct memory map to the accelerator's address space and enables arrays of data to be copied in a single access. Both figures summarize the execution times of the application using only SW threads compared to using the DCT HW accelerator for the MMAP access method including and excluding runtime LKM loading overhead.

When using PetaLinux version 0.4 with software-based timing measurements (recall Figure 5.2), 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. One the other hand, while using PetaLinux version 2.1 with hardware-based timing measurements, as shown in Figure 5.3, performance speedup compared to the SW implementation only increases to a maximum of 8.3x for larger image data sizes (e.g. 4096 macroblocks) for our static SoC configuration. Including the LKM loading overhead that would be incurred for DPR systems reduces the maximum speedup to 6.4x.

5.2.3 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.

Using versions 0.4 and 2.1 of PetaLinux, Figures 5.4 and 5.5 show the execution

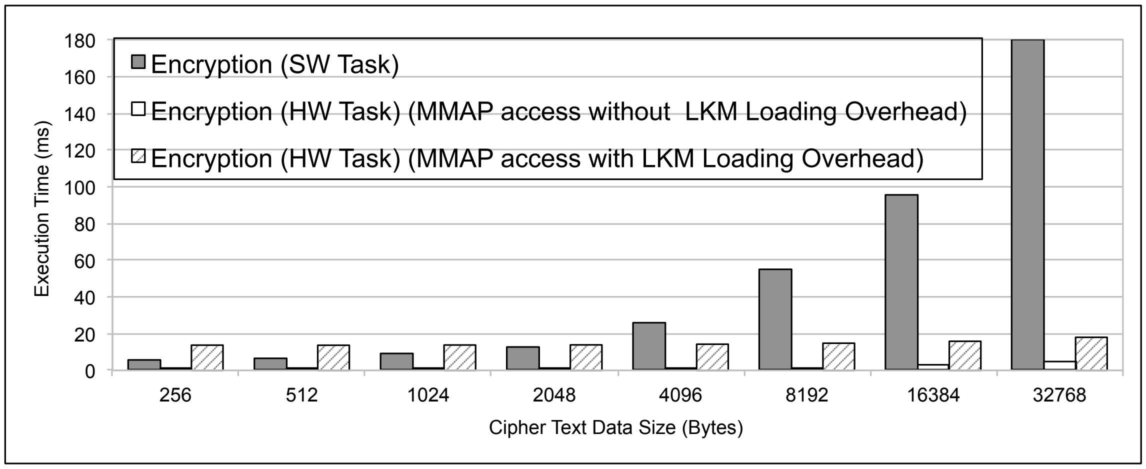


Figure 5.4: Execution Time of 3DES application with different implementations for the encryption task (PetaLinux Ver. 0.4, SW-based timing measurement)

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 only occur for larger cipher texts (>4KB) where the execution time is >25ms. For PetaLinux version 0.4 (Figure 5.4), a maximum performance speedup of 37x without LKM loading overhead, which is reduced to 11x when it is included. On the other hand, using PetaLinux version 2.1 (see Figure 5.5), leads to a maximum performance speedup of 9.0x is achieved without LKM loading overhead, which is reduced to 5.2x when it is included.

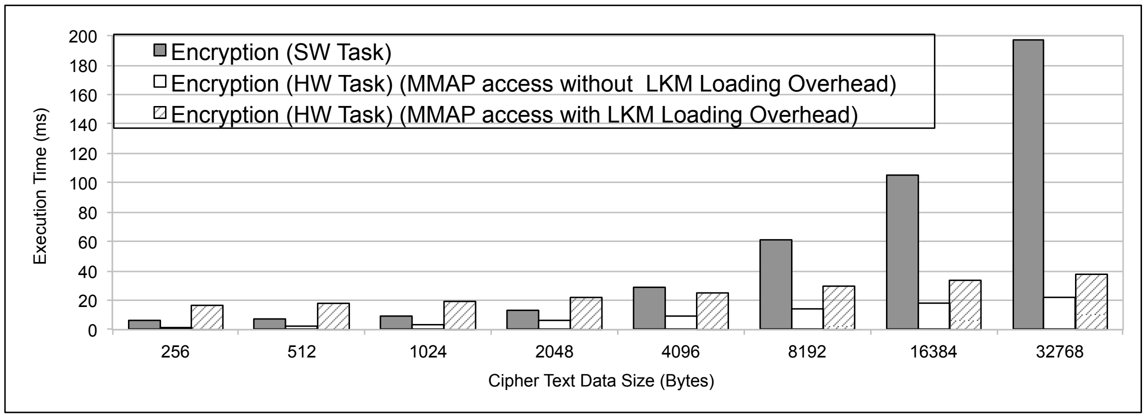


Figure 5.5: Execution Time of 3DES application with different implementations for the encryption task (PetaLinux Ver. 2.1, HW-based timing measurement)

5.2.4 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 pixel window size. This thread is used to detect horizontal and vertical edges of objects in an image. The second thread sharpens the image contents 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.

Figures 5.6 and 5.7 show the execution times of the image filtering application when implemented using PetaLinux version 0.4 and 2.1, respectively. These execution times show the use of 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. For PetaLinux version 0.4 (Figure 5.6), only images containing at least 1024 pixels achieve a visible performance speedup with

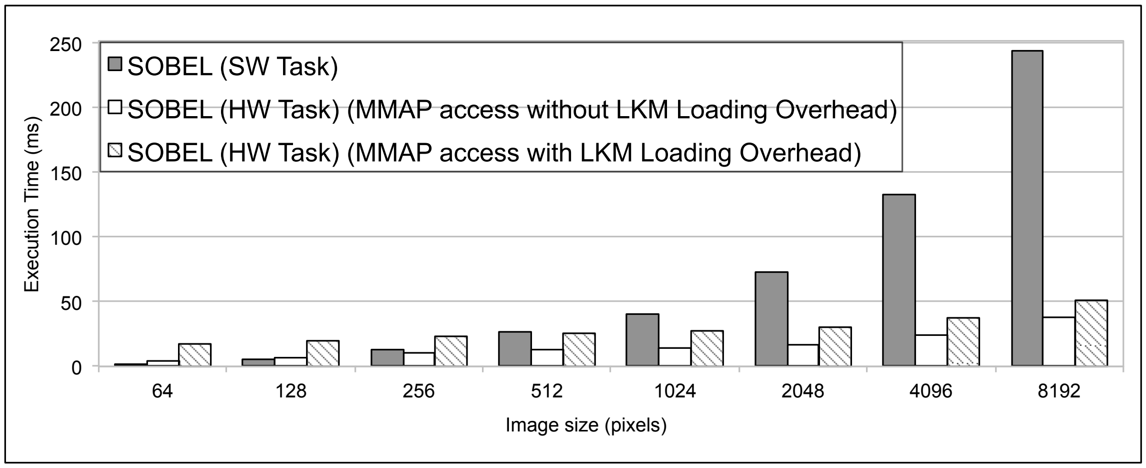


Figure 5.6: Execution Time of the Image Filter application with different implementations of the SOBEL task (PetaLinux Ver. 0.4, SW-based timing measurement)

HW acceleration when LKM loading overhead is included, compared to the SW-only implementation. The performance speedup, with PetaLinux version 0.4, 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. On the other hand, using PetaLinux version 2.1 (Figure 5.7), the performance speedup for SOBEL edge detection using an image with 8192 8-bit pixels in HW is 5.8x excluding LKM loading overhead, which reduces the speedup to 4.1x of the SW version.

5.2.5 Discussion of Case Studies Results

It is clear that the performance speedups for the three example case studies is reduced to some extent when switching from PetaLinux version 0.4 to version 2.1. This could be due to the fact that since the case studies have relatively short runtimes (<500 ms), the SW-based method of timing measurement for PetaLinux version 0.4 does not produce sufficiently precise results, unlike the HW-based method of timing

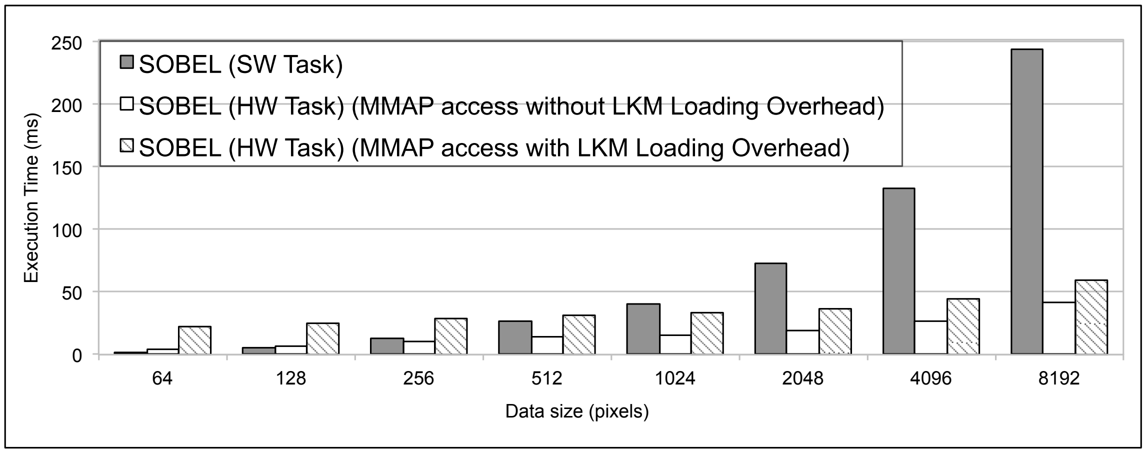


Figure 5.7: Execution Time of the Image Filter application with different implementations of the SOBEL task (PetaLinux Ver. 2.1, HW-based timing measurement)

measurement used for version 2.1 of PetaLinux. To better determine if the difference in these performance speedups is due to the method of measuring execution time, we will discuss in detail the overall overhead from using the FUSE framework in the following section.

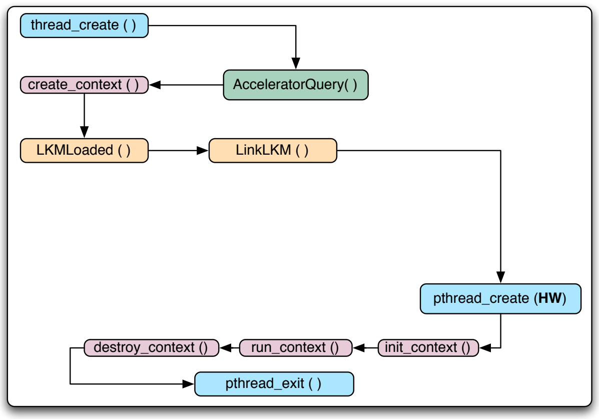
5.3 FUSE Overhead

5.3.1 LKM loading/unloading overhead

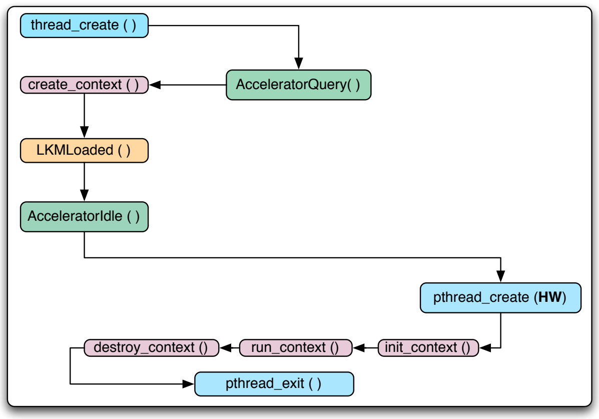
FUSE implements three helper functions, which are used to handle the dynamic load/unloading and checking for LKMs in the OS. Specifically, the *LinkLKM()* uses the *modprobe()* system call to dynamically load an LKM at run-time while *UnlinkLKM()* uses the *rmmod()* system call to dynamically remove the LKM while the system is running. The *LKMLoaded()* helper function uses the *lsmod()* system call to check if an LKM is currently loaded or not. In this section, we quantify the overhead incurred from dynamically loading or unloading the LKM at run-time.

According to the matching policy shown previously in Chapter 3 (see Figure 3.4), FUSE uses these helper functions when the LKMs are compiled to be dynamically loaded or unloaded at run-time. However, LKM loading, when needed, will happen only once; before the HW accelerator is first utilized. For example, Figure 5.8 shows the two main cases in the mapping policy of the TLFC's *thread_create()* function (recall Section 4.1.2). If the matching HW accelerator's LKM is not loaded, it implies that this is the first time this accelerator is being used by FUSE. Therefore, a call to the *LinkLKM()* function is made (see Figure 5.8a). During subsequent uses of that particular accelerator, FUSE continues to check if that LKM is still loaded and, if so, whether the accelerator is being used by another application (see Figure 5.8b).

To illustrate the impact of when FUSE uses these helper functions, Figure 5.9 shows the the number of clock cycles used when calling these functions for each LKM with PetaLinux version 2.1 as measured with a HW cycle counter. It is seen that these functions introduce most of the overhead in the system when used. For example, calling LinkLKM() for the *hwtask_3DES* LKM takes about 18 ms to execute on a system running at clock speed of 125 MHz. When using PetaLinux version 0.4 the overhead of the LKM loading/unloading was measured as a fixed value of 12 ms, with the SW-based method of timing measurement, which is relatively similar to the results seen in Figure 5.9 (14–18ms).



(a) Initial Run



(b) Further Runs

Figure 5.8: TLFC mapping policy cases when loading an LKM

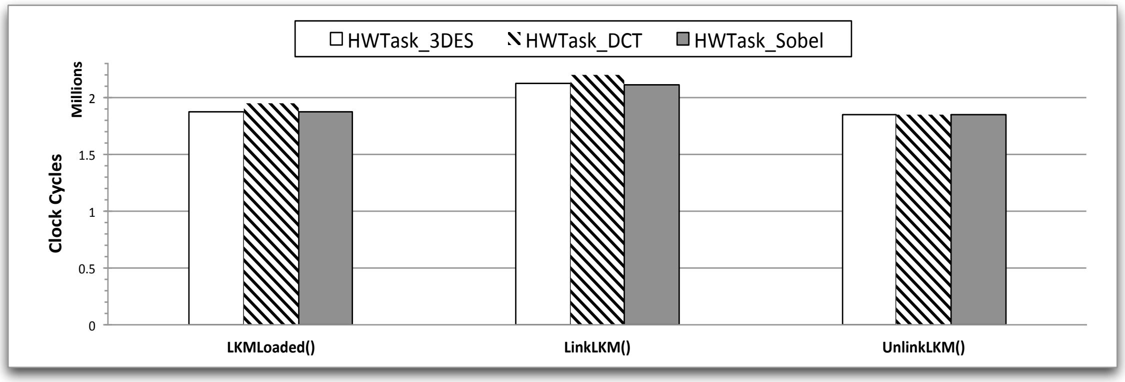


Figure 5.9: Number of clock cycles incurred by the *LKMLoaded()*, *LinkLKM()* and *UnlinkLKM()* FUSE API helper functions (PetaLinux Ver. 2.1, HW-based timing measurement)

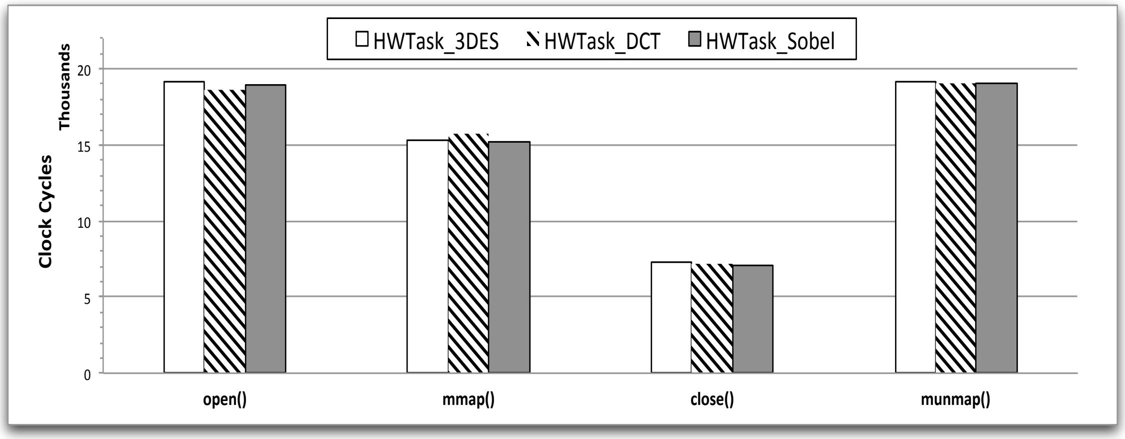


Figure 5.10: Number of clock cycles when using the system calls within the *InitContext()* (i.e. *open()*, *mmap()*) and *DestroyContext()* (i.e. *close()*, *munmap()*) FUSE API helper functions (PetaLinux Ver. 2.1, HW-based timing measurement, CPU operating frequency of 125 MHz)

5.3.2 Run-time Overhead

As shown in Chapter 4, FUSE introduces a layer of abstraction between user applications and available HW accelerators. This often comes at the cost of introducing run-time overhead that impacts performance in certain cases. Therefore, it is the goal of the HW designer to decide on a reasonable trade-off between choosing which part of the application to accelerate in HW and the design time of the HW accelerators themselves. In order to achieve this trade-off, we aim to quantify this overhead based on simple benchmark applications run on the experimental setup. In this section, we present the overhead incurred from the using the file operations provided by the LKM for each HW accelerator.

FUSE's LLFC abstracts HW accelerators as memory-mapped I/O devices from the user-space and presents them as device files that can be manipulated using the traditional file API (e.g. *open()*, *close()*, etc.). Figure 5.10 shows the average number of clock cycles used to execute the *open()*, *close()*, *mmap()*, and *munmap()* file operations exported by the LLFC (See Section 4.2.2) for PetaLinux version 2.1. The time required is very similar to what was measured for PetaLinux version 0.4 using SW-based timing measurements (∼0.91 ms). These file operations are organized according to which of the HW accelerator's LKM they belong. The impact of using these specific operations is considered minimal for two reasons: firstly, they are only called once each time a *thread_create()* call is migrated to run in HW; secondly, the number of clock cycles they consume is negligible compared to the rest of the file operation methods that are used for data communications (i.e. thousands of clock cycles compared to millions of clock cycles).

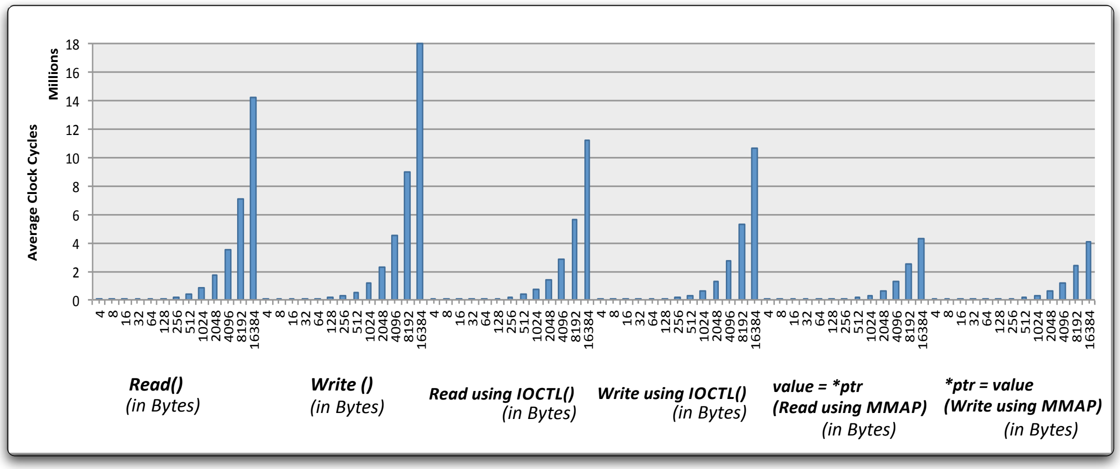


Figure 5.11: Average number of clock cycles for system calls for data communication when used within the *RunContext()* FUSE API helper function, when using PetaLinux version 2.1 with HW-based time measurements.

To better illustrate the impact of data communication between the TLFC and the HW accelerator via its LKM, Figure 5.11 shows the average number of clock cycles (over the 3 LKMs) taken to read and write bytes of data with varying sizes when using PetaLinux version 2.1 with HW-based time measurement. Three methods of data access, supported in a LKM, are investigated:

1. Using the *read()* and *write()* system calls, which reads/writes a buffer of data words to the the HW accelerator via its interface. The *read()* and *write()* calls both perform a similar task, that is, copying data from and to user space. Each one of these calls specifies the size of the requested data transfer and uses a buffer pointer, which points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. The LKM accesses that user buffer by using two special architecture-independent kernel-supplied functions, called *copy_to_user()* and *copy_from_user()*. Those two functions are used to copy each data word from the user space to kernel space and vice versa;

2. Using the *IOCTL()* system call, which reads/writes a data buffer to the HW accelerator via its interface, similar to *read()/write()*. The *IOCTL()* call (short for I/O ConTroL), in general, offers a way to issue accelerator-specific commands (such as setting the accelerator's interface STATUS and COMMAND registers). We have added new commands to the IOCTL method that will enable generic read and write functionality as well, similar to *read()* and *write()*. However, these new IOCTL commands use optimized macros that are architecture-dependent (i.e. MicroBlaze-optimized) to copy data between user space and kernel space (e.g. *inb()* and *outb()*);

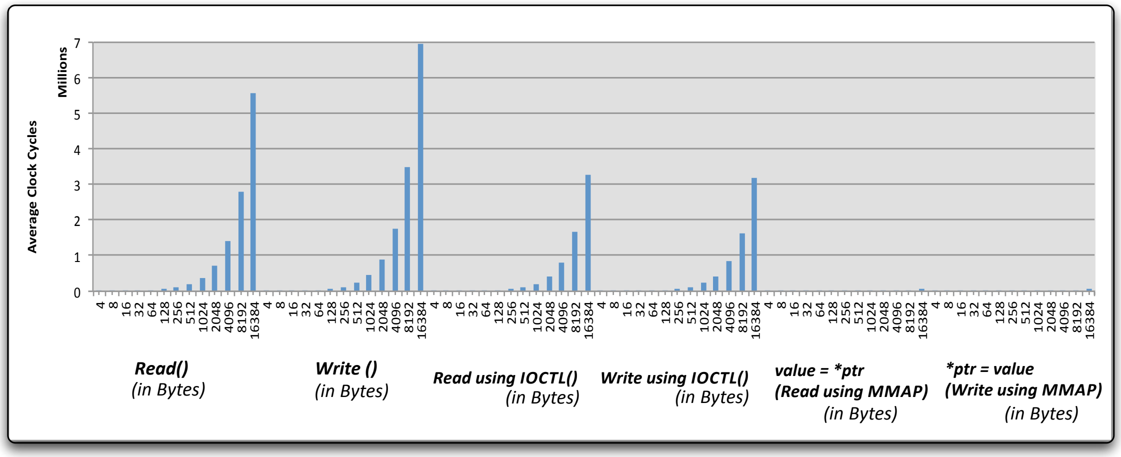
3. Using *memory mapped dereferencing* via the *mmap()* file operation, which establishes a direct memory map to the accelerator's address space in the kernel and the user address space. MMAP enables arrays of data to be copied in a

single access, thus minimizing the number of transitions between user and kernel spaces.

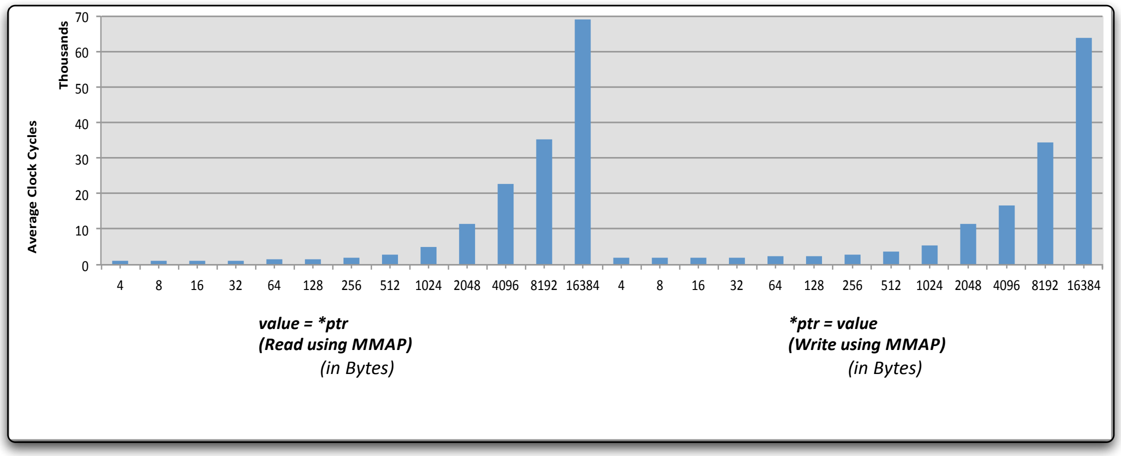
These three methods were considered for the implementation of the *RunContext()* helper function in the TLFC. According to Figure 5.11, using the *read()* and *write()* calls for data communications requires the most clock cycles to execute as the data set increases in size. The IOCTL approach generally requires a longer execution time as well. However, it is less compared to the execution time for the *read()* and *write()* calls. Both *read()/write()* and *IOCTL()* methods perform multiple transitions between the user space and kernel space to transfer data from a user space buffer. However, IOCTL requires fewer clock cycles due to its use of an architecture-dependent set of optimized macros to perform the data transfer. This is in comparison to the kernel-specific architecture-independent *copy_to_user()* and *copy_from_user()* kernel function calls used by *read()* and *write()*, respectively.

The apparent overhead from both of these methods inhibits the potential performance speedup to be gained from HW acceleration. Conversely, the MMAP approach is a far more efficient way to provide bulk data transfer between the TLFC and the HW accelerator; individual *IOCTL()* or *read()/write()* calls inflict overhead due to repeated transitions between the user and kernel space of the OS, where as the MMAP approach allows entire data sets to be transferred 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, only using IOCTL calls for sending and receiving individual control values to the HW accelerator interface's FSM or STATUS/COMMAND registers. We decided not to use the *read()* and *write()* calls due to their large overhead, even though they are still implemented by each LKM in FUSE.

Figure 5.12a shows the average number of clock cycles taken to read and write bytes of data with varying sizes when using PetaLinux version 2.1, obtained using SW-based time measurements. This is a repetition of the same experiment performed with HW-based timing measurements (recall Figure 5.11). We see that both Figure 5.11 and Figure 5.12a demonstrate the same trends for the three data transfer methods (i.e. *read()/write()*, *IOCTL()*, and MMAP-based pointer dereferencing), with the MMAP-based method as the ideal choice to transfer data between the TLFC and the HW Accelerator (see Figure 5.12b). However, it can be clearly seen that using a SW-based method for time measurement results in a lower average number of clock cycles used (i.e. millions for the HW-based method vs. thousands for the SW-based method), which clearly explains why the performance speedup was higher for the three example applications when PetaLinux 0.4 was used. Consequently, we see that using a SW-based method of obtaining measurements does not provide sufficient precision compared to the more precise HW-based method to obtain timing and clock cycle measurements.



(a) Average number of clock cycles for system calls for data communication when used within the *RunContext()* FUSE API helper function, when using PetaLinux version 2.1 with SW-based time measurements



(b) Average number of clock cycles for MMAP-based system call for data communication when used within the *RunContext()* FUSE API helper function, when using PetaLinux version 2.1 with SW-based time measurements (enlarged MMAP region of 5.12a)

Figure 5.12: Average number of clock cycles for system calls for data communication when used within the *RunContext()* FUSE API helper function, when using PetaLinux version 2.1 with SW-based time measurements

Chapter 6

Conclusion and Future Work

In this chapter, we summarize the contributions of this work, draw conclusions from the results, and outline future research directions.

6.1 Contributions

Although the concept of operating system support for FPGA-based SoC computing platforms is not new, most research approaches have focused on the management of the FPGA area and few have focused on the integration of hardware accelerators into the programming and execution environment of the operating system. The purpose of this integration is to provide an abstraction to leverage their heterogenous aspect; specifically when the system has one or more hardware accelerators.

In this thesis, we have presented FUSE, a framework for abstracting 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. In particular, we have provided the following contributions:

- We have devised and implemented a user-level software library that acts as a wrapper for a widely used multithreading programming model and also incorporates new policies to enable applications to use HW accelerators in the system.
- We have devised and implemented low-level OS support for HW accelerators that tie into the user-level software library. This low-level support is both modular and can be added dynamically to a running system.
- We implemented a customizable HW Accelerator Interface, which enables HW designers to easily port HW accelerators to be used in the system.
- We have integrated our framework within an established embedded operating system, Linux, thus leveraging a wide range of software applications and libraries.

6.2 Conclusions

By demonstrating our FUSE framework on a FPGA-based SoC platform running PetaLinux OS for three different applications, we showed that performance speedup can be achieved when using HW accelerators. However, this performance speedup is directly dependent on the applications execution time and also on the run-time overhead if HW accelerators are being used by the application.

We can conclude that an application's potential performance speedup is greatly affected by its execution time. In order for HW accelerator(s) to achieve visible speedup within an application, the run-time overhead must be minimal when compared to the total application execution time. This is clearly suitable for applications that operate on large amounts of data rather than a smaller data sets. We have quantified the run-time overhead incurred from using this framework with the MMAP approach and corresponding kernel support for both static and DPR-based systems. For our case studies, performance speedups (measured via HW-based cycle counters) for static SoC platforms range from 5.8x-9.0x for PetaLinux OS version 2.1, dropping to 4.1x-6.4x when LKM linking overhead is included. When using PetaLinux OS version 0.4, performance speedups (measured via SW-based timers) ranged from 37x-6.4x and dropped to 11x-4.7x when LKM linking overhead is included.

6.3 Future Work

Based on the work presented in this thesis, we see several major directions of future research:

Accelerator Communications: To enable HW accelerators to independently marshall data *during* their execution, an investigation into using a DMA path to the physical memory from the accelerator's interface is a possible future research direction for this thesis. Supporting DMA would reduce the overhead of data transfer between SW tasks and HW tasks. Research related to this thesis is currently investigating direct memory path from the HW accelerator's interface to the Level 2 (L2) cache on a MicroBlaze-based SMP SoC supporting cache coherency.

Communication Networks: Another possible future work is using communication network structures other than the shared-bus example we illustrated in this work and compare the impact on the overhead. For example, a Network-on-Chip (NoC) structure could be used to connect multiple HW accelerators and multiple processors with more efficiency and scalability than a shared-bus structure.

Partial Reconfiguration of Accelerators: How the bitstreams for reconfiguring HW accelerators can be loaded in parallel with their LKMs to mask some of the runtime overhead. This area of research is still new and the possibilities for having a run-time reconfigurable SoC to adapt its HW accelerators to changing application requirements are many-fold.

Bibliography

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

[2] N. Ohba and K. Takano, "An SoC design methodology using FPGAs and embedded microprocessors," in *Proceedings of the 41st annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: ACM, 2004, pp. 747–752. [Online]. Available: http://doi.acm.org/10.1145/996566.996769

[3] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2006, pp. 13–13, January 2006. [Online]. Available: http://dx.doi.org/10.1155/ES/2006/56320

[4] C. Crowley, *Operating Systems: A Design-Oriented Approach.* McGraw-Hill Professional, 1996.

[5] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, April 2003.

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

[7] 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.

[8] Petalogix Inc. (2009, February) Linux solutions for a reconfigurable world. [Online]. Available: http://www.petalogix.com

[9] Xilinx Inc. (2011, June) FPGA, CPLD, and EPP solutions from Xilinx, Inc. [Online]. Available: http://www.xilinx.com/

[10] D. Brasen, “ASIC Prototyping with Reprogrammable Implementations of Large ASICs,” in *Proceedings of the 7th IEEE International Workshop on Rapid System Prototyping (RSP ’96)*, ser. RSP ’96. Washington, DC, USA: IEEE Computer Society, 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=827259.828061

[11] R. Hartenstein, “Trends in reconfigurable logic and reconfigurable computing,” in *Electronics, Circuits and Systems, 2002. 9th International Conference on*, vol. 2, 2002, pp. 801–808.

[12] W.-T. Zhang, L.-F. Geng, D.-L. Zhang, G.-M. Du, M.-L. Gao, W. Zhang, N. Hou, and Y.-H. Tang, “Design of heterogeneous MPSoC on FPGA,” in *Proceedings of the 7th International Conference on ASIC, 2007. ASICON ’07.*, oct. 2007, pp. 102–105.

[13] Xilinx Inc. (2011, June) Virtex 2 Platform FPGA datasheet. [Online]. Available: http://www.xilinx.com/support/ documentation/data_sheets/ds083.pdf

[14] ——. (2011, June) Virtex 5 Platform FPGA Family Overview. [Online]. Available: http://www.xilinx.com/support/ documentation/data_sheets/ds100.pdf

[15] ——. (2011, June) 7 Series FPGA Overview. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[16] Altera Inc. (2011, June) Stratix V Device Family Overview. [Online]. Available: http://www.altera.com/literature/ hb/stratix-v/stx5_51001.pdf

[17] ——. (2011, June) Cyclone VI Device Family Overview. [Online]. Available: http://www.altera.com/literature/ hb/cyclone-iv/cyiv-51001.pdf

[18] A. DeHon, “The density advantage of configurable computing,” *Computer*, vol. 33, no. 4, pp. 41–49, April 2000.

[19] M. Abramovici, C. Stroud, and M. Emmert, “Using Embedded FPGAs for SoC Yield Improvement,” in *Proc. ACM/IEEE Design Automation Conference*, 2002, pp. 713–724.

[20] D. Pellerin and S. Thibault, *Practical FPGA programming in C*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2005.

[21] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen, “Application-specific customization of parameterized FPGA soft-core processors,” in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided*

design, ser. ICCAD '06. New York, NY, USA: ACM, 2006, pp. 261–268. [Online]. Available: http://doi.acm.org/10.1145/1233501.1233553

[22] Xilinx Inc. (2010, February) Picoblaze Soft Processor User Guide. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf

[23] Altera Inc. (2011, May) The NIOS Soft CPU Family. [Online]. Available: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[24] Xilinx Inc. (2008, January) The Microblaze processor reference guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[25] T. Marescaux, V. Nollet, J.-Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Run-time support for heterogeneous multitasking on reconfigurable SoCs," *The VLSI Journal on Integration*, vol. 38, pp. 107–130, October 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1056481.1056488

[26] P. Ienne and R. Leupers, *Customizable Embedded Processors–Design Technologies and Applications*. Morgan Kaufmann, 2006.

[27] A. Upegui and E. Sanchez, "Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs," in *International Conference on Evolvable Systems*, 2005, pp. 56–65.

[28] 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.

[29] P. B. Hansen, *Operating system principles*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1973.

[30] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[31] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit, "A Dynamic Reconfiguration Run-Time System," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, April 1997.

[32] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *International Proceedings of the Parallel and Distributed Processing Symposium*, april 2003, p. 7 pp.

[33] G. Wigley and D. Kearney, "The Development of an Operating System for Reconfigurable Computing," in *In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM). IEEE CS*, 2001.

[34] ——, "Research Issues in Operating Systems for Reconfigurable Computing," in *in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2002, pp. 10–16.

[35] QNX Software Systems. (2011, August) Middleware, development tools, realtime operating system software and services for superior embedded design. [Online]. Available: http://www.qnx.com

[36] Xilinx Inc. (2011, June) OS and Libraries Document Collection. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/edk10_oslib_rm.pdf

[37] C. Patterson, "A Dynamic Module Server for Embedded Platform FPGAs," in *Engineering of Reconfigurable Systems and Algorithms*, ser. ERSA '03, 2003, pp. 31–40.

[38] J. Tong, I. Anderson, and M. Khalid, "Soft-core processors for embedded systems," in *Microelectronics, 2006. ICM '06. International Conference on*, December 2006, pp. 170–173.

[39] H. Calderon, C. Elena, and S. Vassiliadis, "Soft Core Processors and Embedded Processing: a survey and analysis," in *Conference Proceedings*, 2005, pp. 483–488.

[40] J. Williams and N. Bergmann, "Reconfigurable Linux for Spaceflight Applications," in *In Proceedings of Military and Aerospace Programmable Logic Devices*, 2004.

[41] ——, "Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip," in *In Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, ser. ERSA '04, 2004.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] X. Xie, J. Williams, and N. Bergmann, “Asymmetric multiprocessor architecture for reconfigurable system-on-chip and operating system abstractions,” in *International Conference on Field-Programmable Technology.*, Dec 2007, pp. 41–48.

[47] N. W. Bergmann, J. A. Williams, J. Han, and Y. Chen, “A Process Model for Hardware Modules in Reconfigurable System-on-Chip,” in *ARCS Workshops*, 2006, pp. 205–214. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings81/article4365.html

[48] J. A. Williams, N. W. Bergmann, and X. Xie, “FIFO Communication Models in Operating Systems for Reconfigurable Computing,” in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 277–278. [Online]. Available: http://portal.acm.org/citation.cfm?id=1090947.1091332

[49] N. Moore, A. Conti, M. Leeser, and L. S. King, “Vforce: An Extensible Framework for Reconfigurable Supercomputing,” *Computer*, vol. 40, pp. 39–49, March 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1251558.1251715

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

[51] 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.

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

[53] 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.

[54] 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.

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

[56] B. Lewis and D. J. Berg, *Multithreaded programming with Pthreads.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[57] The IEEE and The Open Group, *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition.* New York, NY, USA: IEEE, 2004. [Online]. Available: http://www.opengroup.org/onlinepubs/009695399/