TERRAE: A Framework for Adaptive Hardware Concurrent Systems

by

Victor Gusev Lesau

B.Eng. (Electrical), McMaster University, 2006

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 Sciences

SIMON FRASER UNIVERSITY

Fall 2012

Approval

Name: **Victor Gusev Lesau**

Degree: **Master of Applied Science**

Title of Thesis: ***TERRAE: A Framework for Adaptive Hardware Concurrent Systems***

Examining Committee:

Chair: **Dr. Carlo Menon**
Assistant Professor,
School of Engineering Science

Dr. William A. Gruver
Senior Co-Supervisor
Professor Emeritus,
School of Engineering Science

Dorian Sabaz
Senior Supervisor
CTO, Holonic Technologies Inc.

Dr. Richard Hobson
Co-Supervisor
Professor Emeritus,
School of Engineering Science

Dr. Craig Scratchley
Internal Examiner
Senior Lecturer,
School of Engineering Science

Date Defended/Approved: October 3, 2012

Partial Copyright Licence

Abstract

Dynamic Partial Reconfiguration (DPR) of Field Programmable Gate Arrays (FPGAs) is a technology that enables the development of embedded systems with hot swappable logic on the FPGA fabric. The advantage is that hardware logic can be swapped in and out “on-the-fly” while the rest of the system is operational. Since DPR is relatively new, tool support is still evolving. This thesis introduces new FPGA architectural tools and Linux OS modifications that aid in supporting DPR on FPGAs for concurrent control. It shows that control systems benefit from hardware concurrency, meaning that by moving the control intelligence into hardware, the negative effects inherent to threads and their scheduler are minimized. This leaves software with the role of a high-level administrator rather than an executor, thereby eliminating unnecessary bottlenecks. The tools described in this thesis enable the hardware engineer to develop DPR-FPGA systems more effectively for rapid control system development. Furthermore, the introduced Adaptive Hardware Concurrent System (AHCS) architecture illustrates how a designer can take operating systems to a new level of concurrency resulting in true deterministic concurrency implemented on DPR-enabled hardware platforms.

Keywords: Adaptive Hardware; Embedded Linux; Hardware Concurrency; Field Programmable Gate Arrays; Dynamic Partial Reconfiguration; SoC Design

To my family

Acknowledgements

This thesis would not have been possible without the support of many people. First of all, I would like to thank the examining committee members for dedicating their time and energy in reviewing the thesis and providing the invaluable feedback.

Also, I wish to express my gratitude to Dr. William A. Gruver for his thoughtful guidance, sincere advice and constructive feedback throughout my journey towards Masters. His wisdom and inspiration made a significant impact on this project.

As well, a special thanks to Dorian Sabaz for being an unlimited fountain of ideas, which he shared with an open heart and without reservation, making it a truly joyful and inspirational learning experience. The long whiteboard sessions, countless Skype calls, and heated debates not only about technology, but also history, psychology, economics, and management played a key role in diversifying my thinking, and establishing a lifelong friendship.

Furthermore, I would like to thank PMC-Sierra Inc. for providing support in completing this thesis, as well as the great people working there who shared my excitement and encouraged me throughout this project.

I wish to thank George Babut for “infecting me with a technology bug” and fueling my initial curiosity in reconfigurable hardware and communication systems, which defined my career and academic paths to this point.

Thanks to my grandfather, Victor Lesau, for influencing me from the childhood, and being a living example of a passionate researcher with an open sincere character.

Finally, I wish to thank my family for giving me the opportunity getting to this point, supporting my values, and providing unconditional love. We made it!

Table of Contents

List of Tables

List of Figures

Glossary

AHCS	Adaptive Hardware Concurrent System
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BOM	Bill of Materials
BRAM	Xilinx Block RAM
BSP	Board Support Package
CBD	Component-Based Design
CF	Compact Flash
CLT	Command Line Tool
CPU	Central Processing Unit
DCM	Digital Clock Manager
DP	Daemon Process
DDR	Double Data Rate synchronous dynamic random access memory
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DPR_SS	DPR Subsystem
ENET	Ethernet
EDK	Xilinx Embedded Development Kit
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
GPIO	General Purpose Input/Output
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HA	Hardware Administrator
HA_CPU	CPU attached to Hardware Administrator
HCApp	HA_CPU Application
HCC	Hardware Communication Controller
HLC_SS	High-Level Control Subsystem
HPC	High Performance Computing
HRC	Hardware Reconfiguration Controller

IP	Intellectual Property
ICAP	Xilinx Internal Configuration Access Port
INTC	Interrupt Controller
IPC	Inter-Process Communication
IRQ	Interrupt Request
ISE	Xilinx Integrated Software Environment
LIBPR	Supporting library for Partial Reconfiguration
LINUX_CPU	CPU hosting Linux OS
LUT	Lookup Table
MBOX	Mailbox
NoC	Network-on-Chip
NRE	Non-Recurring
OS	Operating System
OPB	On-chip Peripheral Bus
OTN	Optical Transport Network
PAR	Place and Route
PLB	Processor Local Bus
PR	Partial Reconfiguration
PRM	Partial Reconfiguration Module
PRR	Partial Reconfiguration Region
RISC	Reduced Instruction Set Computing
RTL	Register-Transfer Level
RX	Receive
SDR	Software Defined Radio
SoC	System-on-Chip
SOP	Start of Packet
TERRAE	InTERfRAmEwork
TX	Transmit
UART	Universal Asynchronous Receiver/Transmitter
uB	MicroBlaze processor
uP	Microprocessor
UML	Unified Modeling Language
VM	Virtual Machine

WS	Workstation
XMD	Xilinx Microprocessor Debugger
XPS	Xilinx Platform Studio
XUP	Xilinx University Program

1. Introduction

1.1. Motivation

Until recently, embedded operating system (OS) environments on FPGAs were designed to support either hardware accelerated threaded software, usually done for high performance computing, or generally provide integration support for onboard sensor monitoring and actuator control systems. With the advent of Dynamic Partially Reconfigurable FPGAs (DPR-FPGAs) that allow individual bitstreams to be swapped onto specific regions of the FPGA fabric, high-level system support that focused on supporting the highly concurrent characteristics of these DPR regions was largely lacking.

As DPR-FPGAs become larger and more capable to provide greater partitioning of the FPGA fabric, including increasing the number of possible bitstreams that can be loaded into these DPR regions, on-board OS support for these concurrently running hardware regions is more demanding. Until recently, operating systems were primarily temporal schedulers of software code segments. DPR-FPGAs demand operating systems to have the added capability of spatial managers of DPRs, in addition to the scheduling of off-FPGA stored hardware bitstreams. In fact, as DPR-FPGAs increase in size, it can be argued that the temporal schedulers for threaded architectures become less important, while spatial scheduling, which is highly essential for supporting concurrency, becomes more of an issue.

One area where concurrency plays an important role is the discipline of control systems targeting applications that require hardware implementations of core functionality due to area, power, cost and response time constraints [1]-[2]. For example, consider a 100-Gbit Optical Transport Network (OTN) muxponder line card for multiplexing lower data rate clients into a single 100-Gbit optical transport signal [3]. Such control systems usually have been implemented with ASICs, merchant silicon, or

statically configured FPGAs having fast data paths and control planes, avoiding software implementations at the cost of design time, flexibility and dynamic partial reconfigurability [1]-[2]. One reason for this situation is that control systems had more sophisticated system requirements than FPGAs were able to provide until recently. For example, considering the combination of a system's physical constraints, adaptability to a dynamically changing environment and remote upgradability, to date processors have provided the most flexibility. Yet for applications such as the muxponder card, CPU implementations are not sufficient to meet the latter requirements. This situation has changed with the advent of DPR and with available sizes of fabric that are now offered with the latest FPGAs. DPR-FPGAs answer the above requirements and can now be partitioned to offer enough concurrency fabric to control systems, which makes them an appropriate technology for implementing commercial and industrial systems. The early examples of such control systems are implementations of network elements by Altera [3] and Xilinx [4].

Another reason for the lack of adoption of DPR-FPGAs for concurrent control systems is that tools for both statically-configured and DPR-based FPGAs have focused on implementing computer architectures that are mostly founded on threaded architectures [5]-[11], which make them solely processor centric. On the other hand, the needs of concurrent systems oppose this architectural methodology. It should be remembered that threads and their scheduling were originally created to imitate concurrency due to the processors being serial instruction execution machines. However, as their numbers are increased, threads introduce non-determinism. As Lee [12] has described, inherent problems with threaded systems pull these systems further from their ideal just-in-time execution requirements that are necessary for control applications.

As already stated, control systems would benefit from control intelligence residing in hardware. Notwithstanding the benefits of hardware acceleration, the demands on reconfigurable fabric can be reduced, without introducing the limitations and non-deterministic demands imposed by threads and their scheduling [12]. Instead, OS can take the responsibility of a high-level administrator and an interface to the outside world.

1.2. Contributions

This thesis describes kernel and user space additions to Linux to provide support for the DPR hardware subsystem, along with a highly concurrent FPGA computer architecture realized on a Xilinx ML505 development system with the Virtex-5 FPGA. It examines the requirements of DPR demand on FPGA hardware design and how the use of Linux can extract maximum advantage for developing concurrent control systems. It also introduces an Adaptive Hardware Concurrent System (AHCS), which has four main attributes: (1) reconfigurable user logic for adaptive behavior; (2) changeable interfaces for decoupling the logic from the system architecture; (3) adaptive architecture to which modules can bind and rebind via the interfaces, and; (4) a hardware operating system, that does not use a processor, and, thereby does not restrict concurrency. This thesis presents the following significant results:

- Implements a framework to integrate a highly concurrent framework previously developed at the iDEA Laboratory of Simon Fraser University as part of a Ph.D. thesis [13]-[14] with the Linux OS on a Xilinx Virtex-5 DPR-FPGA;
- Implements an architecture for a concurrent control system removing the impact of the FPGA's internal partial reconfiguration on the processor with the OS and its system bus;
- Implements a Linux mailbox driver for a Xilinx Mailbox Intellectual Property (IP) core to enable connectivity between frameworks;
- Implements a software application programming interface (API), allowing Linux command line communication with DPR hardware modules;
- Introduces the AHCS architecture and a path towards the migration of an OS, such as Linux, to hardware. The goal is to realize a fully concurrent control system, without the utilization of a processor.

1.3. Organization

This thesis is structured in the following manner. Chapter 2 discusses the current state of the art in the area of Linux and DPR-FPGAs for adaptive hardware concurrent systems. Chapter 3 states the requirements and proposes a framework (TERRAE) with hardware and software architecture that minimizes the effect of threads on such systems. Chapter 4 discusses implementation details, and Chapter 5 analyzes

the results. Finally, the thesis concludes with a discussion of the contributions and future work in Chapter 6.

2. Background

In the last few decades, technology has been evolving at a fast pace, which is observed by progressively complex system-on-chip (SoC) designs. Thus, the demands being placed on hardware are becoming increasingly sophisticated, especially for engineering control solutions. In many ways, one can make the compelling affirmation that hardware needs become similar to software needs: adaptable, reconfigurable, and relocatable.

This section investigates topics in the core areas that pertain to the above supposition and to the underlining ideas of this thesis. The background information will put into perspective how FPGAs and Linux have been positioned over time, especially with regard to an existing bias towards developing solutions for high-performance computing (HPC) and computer science, as opposed to developing systems for control engineering that require support for full hardware-level concurrency.

2.1. Field Programmable Gate Arrays

Since their introduction in the mid-1980s [15], FPGAs have played an increasingly important role in building hardware systems due to their reconfigurability, lower non-recurring (NRE) costs and shorter times to market when compared to ASICs. They have been used in applications for automotive, industrial, medical, networking, aerospace, defense, software-defined radio, digital signal processing and others.

The major advantages of FPGAs arise from their reconfigurable architecture. This is achieved by an array of logic blocks and routing channels (Figure 1). Each typical logic block contains several lookup tables (LUT) for implementing multi-input/multi-output functions, a register for capturing the result and a multiplexor for bypassing it [16]. More complex FPGAs integrate multipliers and other circuits for additional functionality. The routing channels have programmable switches allowing

interconnection of logic blocks. The configuration of both LUTs and switches is sent to the FPGA during reconfiguration in the form of a bitstream generated by development tools. As a result, the fabric can be reconfigured by an external device with a different circuit representation on demand, depending on the application.

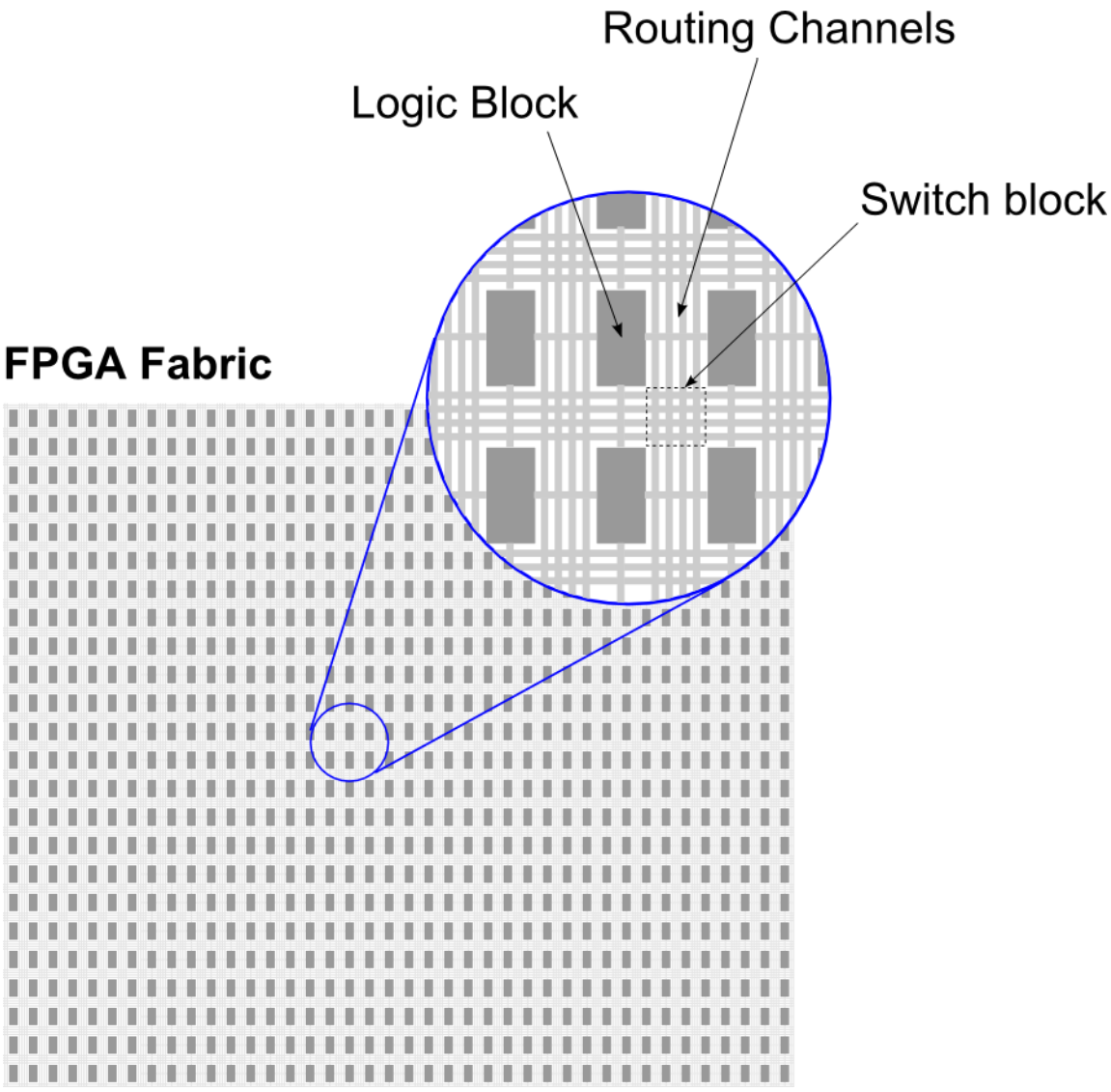
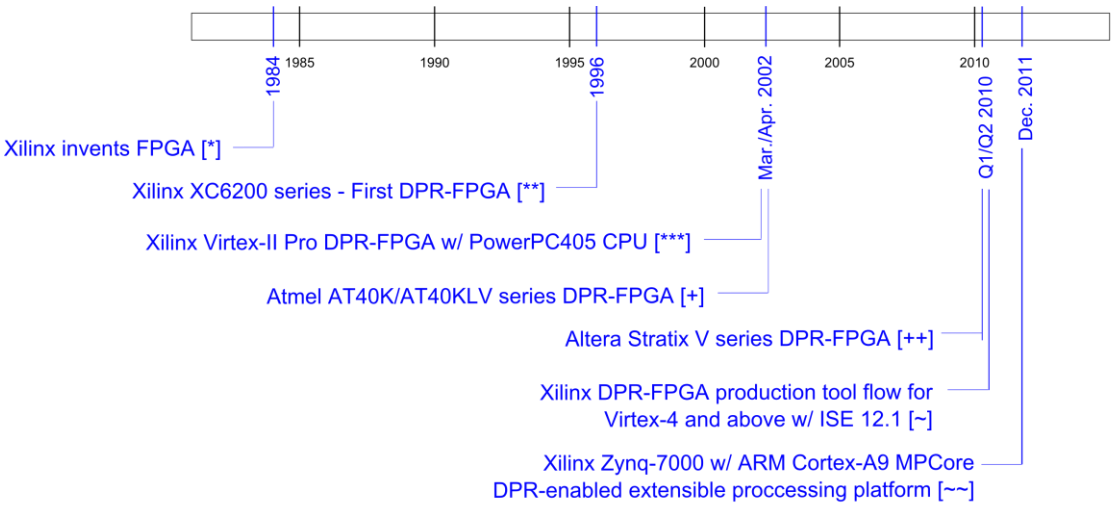


Figure 1: FPGA fabric

These advantages do not come without drawbacks. Generally, FPGAs are slower than custom ICs of the same technology node, draw more power and provide smaller densities due to reconfiguration overhead, and are more costly when compared with ASICs at a higher volume of production.

2.2. Dynamic Partial Reconfiguration

With the introduction of the Xilinx XC6200 in the 1990s and later with the Xilinx Virtex families of devices, it became possible to reconfigure parts of FPGAs dynamically without affecting the operation of the rest of the logic [17]-[18]. A similar functionality was later introduced in the AT40K family by Atmel Corporation [19] and Stratix V from Altera Corporation [20]. Applications of DPR-FPGA technologies can be found in implementations of network elements [3]-[4], waveform processing for software defined radio (SDR) [21], fast PCIe configuration [22], and asymmetric key encryption [23]. The timeline of major events in DPR-FPGA technology development history is summarized in Figure 2.



DPR-FPGA references:
- [*] D.W. Page, "Dynamic Data Re-Programmable PLA," U.S. Patent 4524430, June 18, 1985.
- [**] *XC6200 FPGA Advance Product Specification*, product specification, Xilinx Inc., June 1996.
- [***] *Xilinx Introduces Breakthrough Virtex-II Pro FPGAs to Enable New Era of Programmable System Design*, Xilinx Press Release #0204, Xilinx Inc., Mar. 4, 2002.
- [+] *AT40K/AT40KLV Series FPGA*, datasheet (0896CS-FPGA), Atmel Corp., Apr. 2002.
- [++] Stratix V Device Family Overview, datasheet (SV51001-2.3), Altera Corp., Feb. 2012.
- [~] *Partial Reconfiguration User Guide*, UG702 (v12.1), Xilinx Inc., May 3, 2010.
- [~~] (2011, Dec. 8) *Xilinx Ships F.irst Zynq-7000 Devices, the World's First Extensible Processing Platform* [Online]. Available: http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1637670

Figure 2: Timeline of DPR-FPGAs [15], [18]-[19], [23]-[27]

The early devices, however, had severe DPR constraints. For example, looking at the widely used devices from Xilinx, the Virtex-II Pro family with early tool flows restricted the size and placement of Partial Reconfigurable Regions (PRRs) to entire columns of the FPGA [28]. This restriction was changed with the advent of the Virtex-4

and Virtex-5 families and their flexible-sized PRRs allowing a tile-based frame architecture [28]-[29]. New bus macros (BMs) in these devices, physical interfaces between static and dynamic parts of the design, allowed for vertical and horizontal connectivity between the PRRs. This enabled the construction of 2D structures [30]. The situation improved when BMs became obsolete and interfaces were abstracted by an automated flow since the introduction of Xilinx ISE 12.1 tools, leaving a reasonable requirement for registering of signals at PRR boundaries and gating them off during reconfiguration [23]. Figure 3 below shows an FPGA fabric with three PRRs of different dimensions.

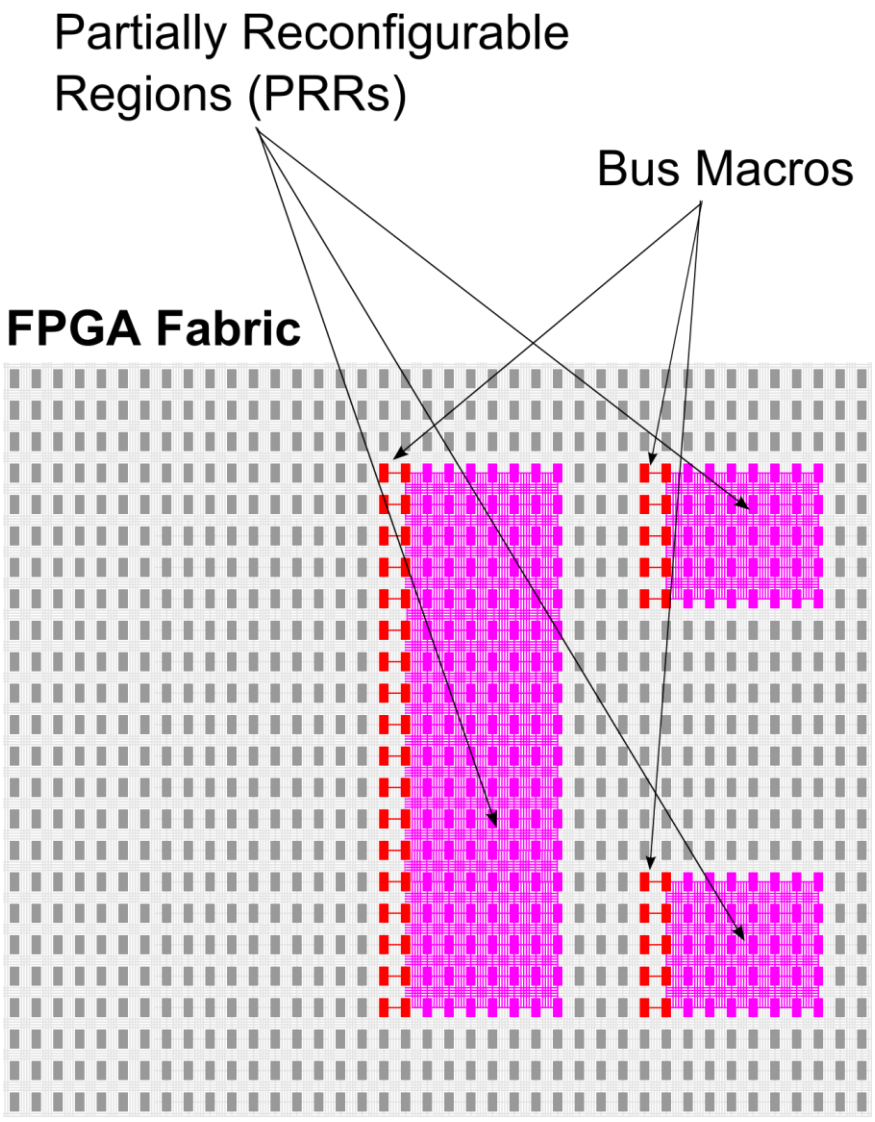


Figure 3: Dynamic partially reconfigurable FPGA fabric

2.3. Component-Based Design

These advancements made DPR technology more attractive for architectures with Partial Reconfiguration Modules (PRMs). By analogy, it made FPGAs for hardware seem similar to memory for software. In particular, blocks of hardware logic responsible for certain functionality can now be swapped like software libraries, and can be loaded and unloaded while the system is operational. This development enabled the engineering community to develop tools for reconfigurable configuration of hardware. There were two approaches for developing these tools. The designer would treat DPR-FPGAs as a new paradigm for development and thereby construct entire systems specifically for DPR hardware alone, or abstract the hardware DPR into a hardware form of memory, analogous to that used in software development today. In other words, a designer could benefit from the use of technological methodologies that software has developed during the last twenty years.

One such software methodology is Component-Based Design (CBD), where software code modules are placed behind interfaces, allowing the code to be reconfigured without breaking any API dependencies. This technique is most commonly used for distributed systems development such as Internet applications. These methodologies allow for development of scalable modular software frameworks and can now be applied for DPR-based architectures of the recent FPGAs [13]-[14]. In essence, this creates a possibility for hardware cores being viewed and managed like hard versions of software agents. These hardware agents would provide all the benefits of software agents: decoupling of logic from system architecture, and establishing a higher level of hardware encapsulation.

2.4. Dynamically Reconfigurable Architecture

DPR-FPGA technology and CBD together enable not only encapsulation of hardware modules and their runtime reconfigurability, but also the ability to change a system's architecture "on-the-fly." This ability is important especially for sophisticated control systems. We can get inspiration for why this is so if we look to one of the most

advanced control systems—the human cerebellum. The human cerebral cortex is constantly “rewiring” its neural architecture as a person’s environment changes [31]-[32]. Thus, a strong argument can be made that for electronic control systems to become more sophisticated and adaptable, their computer architectures and logic running on these architectures need to possess a similar level of adaptability. Looking at dendrites and axons, the neural network implementation of input and output interfaces, we note that they are not only changing themselves, but also changing in association with their neural network architecture. Thus, we could also mirror this result with electronic hardware. This concept admittedly is not new; the field of Cybernetics is founded on similar ideas [33]. The novelty is that this hardware adaptability can now be accomplished with DPR-FPGAs.

With the use of DPR on FPGAs, there is now the potential to design electronic systems on FPGAs that will have functional flexibility similar to the human brain. Of course, compared with an FPGA, there is still a large difference in size between what is available as a neural network and its associated complexity. Nevertheless, an opportunity exists now to develop new control structures and architectures that provide large scale concurrency. This leads to a conclusion that if we are to learn from nature in building massively concurrent control systems such as the human brain, we have to emphasize the importance of not only dynamically adaptable modules and their interfaces, but also architectures, which are now practical with the help of emerging paradigms in DPR hardware.

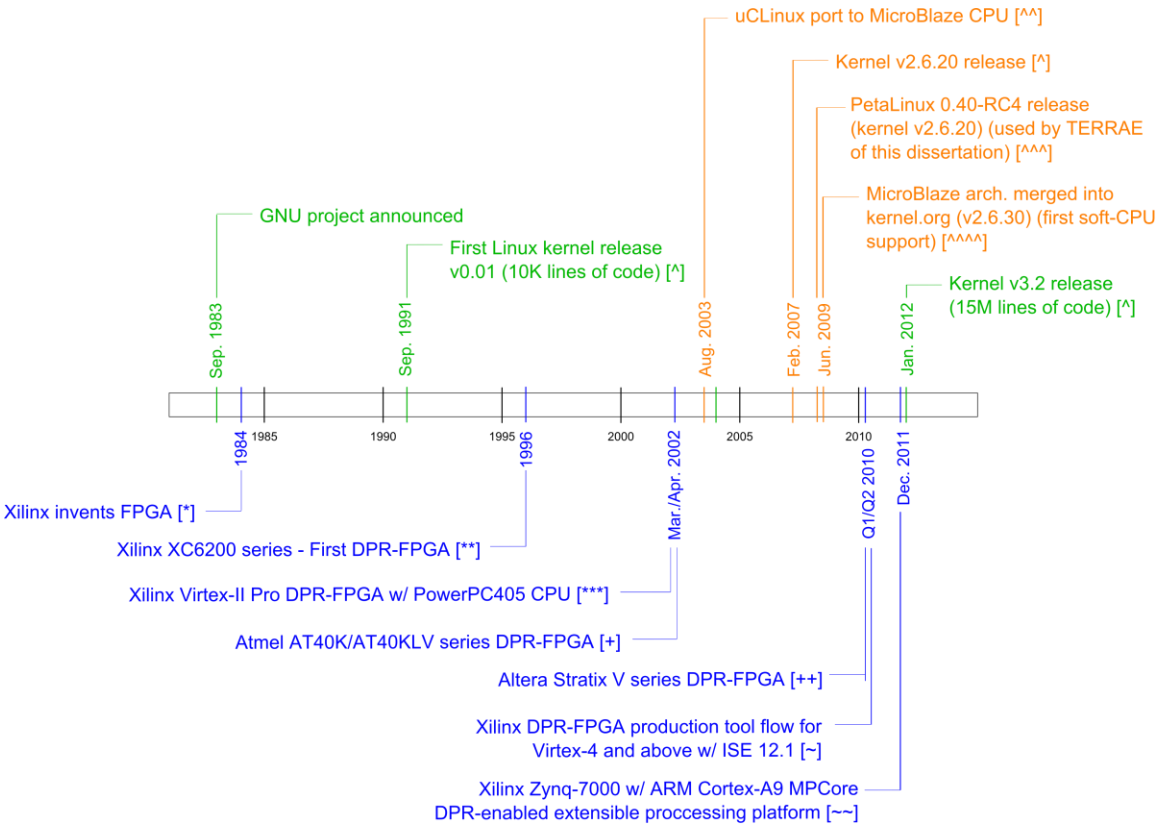
2.5. Linux and Embedded Devices

An embedded operating system is a software system that is deployed on an embedded processor, which in turn is a part of a device that is not a general purpose computer [34]. Currently, a variety of open source and commercial embedded operating systems are available on the market [35]-[37]. Embedded Linux is one the most widely used operating systems in devices ranging from mobile phones and set-top boxes to networking equipment and robotics [37]-[40]. Linux began its history in 1991 as a personal project and grew in size with its kernel core currently exceeding 15 million lines of code (v3.2), stable releases made every 2-3 months, and over a thousand developers

from nearly 200 companies contributing to each release [41]. It is available without royalties or licensing fees, permits modification and redistribution of its source code, and allows for off-the-shelf communication and other services. Consequentially, the advantages are reduction of initial development time and on-going support costs, release stability, security and customizability.

The architecture of the Linux kernel is monolithic, meaning that the core functionality shares the same address space with networking, drivers and other services [42]. It is also modular with the help of kernel modules, which have the ability to be either compiled into the kernel or dynamically linked at runtime [43]. The advantage of this architecture over the microkernel, which runs only a minimal number of services needed to implement an OS such as address space management, thread management and inter-process communication, is increased performance and simpler debugging. The disadvantages are code bloating and also reliability issues due to a higher chance of system crash if a part of kernel space code causes an error [44]. Major events in Linux overlaid with DPR-FPGA history that are relevant to this thesis are highlighted in Figure 4.

With the increase in FPGA fabric size and the introduction of hard processor cores, Linux expanded its reach into this new platform. In the 1990s it was ported to one of the major architectures, PowerPC [45]-[46], which later emerged as a hard IP core in the Virtex-II Pro FPGA in 2002 [24]. Also, in 2003, the uCLinux distribution was ported to a popular MicroBlaze soft processor targeting Xilinx FPGAs by Williams [47]. With the release of kernel 2.6.30 on June 2009, MicroBlaze architecture was merged into the mainline kernel tree [48]. This thesis uses the PetaLinux toolchain by PetaLogix [49], which is based on the uCLinux port for MicroBlaze.



DPR-FPGA references:
- [*] D.W. Page, "Dynamic Data Re-Programmable PLA," U.S. Patent 4524430, June 18, 1985.
- [**] *XC6200 FPGA Advance Product Specification*, product specification, Xilinx Inc., June 1996.
- [***] *Xilinx Introduces Breakthrough Virtex-II Pro FPGAs to Enable New Era of Programmable System Design*, Xilinx Press Release #0204, Xilinx Inc., Mar. 4, 2002.
- [+] *AT40K/AT40KLV Series FPGA*, datasheet (0896CS-FPGA), Atmel Corp., Apr. 2002.
- [++] Stratix V Device Family Overview, datasheet (SV51001-2.3), Altera Corp., Feb. 2012.
- [~] *Partial Reconfiguration User Guide*, UG702 (v12.1), Xilinx Inc., May 3, 2010.
- [~~] (2011, Dec. 8) *Xilinx Ships F.irst Zynq-7000 Devices, the World's First Extensible Processing Platform* [Online]. Available: http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1637670

Linux references:
- [^] J. Corbet, G .Kroah-Hartman and A. McPherson, "Linux Kernel Development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It," white paper, The Linux Foundation, Mar. 2012.

MicroBlaze and TERRAE project related Linux references:
- [^^] (2003, Aug. 25). *[microblaze-uclinux] [ANN] Microblaze uClinux demo released* [Online]. Available: http://itee.uq.edu.au/~listarch/microblaze-uclinux/archive/2003/08/msg00054.html
- [^^^] (2009, June 22). *Re: Corrections for download URL link and website* [Online]. Available: http://permalink.gmane.org/gmane.linux.uclinux.microblaze/8997
- [^^^^] (2009, June 9). *MicroBlaze CPU architecture merged into kernel.org* [Online]. Available: http://www.petalogix.com/news/microblaze-architecture-merged-into-kernel.org

Figure 4: Timeline of Linux and DPR-FPGAs [15], [18]-[19], [23]-[27], [41], [45]-[51]

2.6. Embedded Systems from Static FPGA to DPR-FPGA SoC

As it has been stated, FPGAs have advantages over ASICs and merchant silicon for applications requiring reconfigurability, remote upgradability and shorter time-to-market. With the Linux OS being a popular choice for embedded devices, it is no surprise that both technologies have been combined together by engineering hands for building complex embedded reconfigurable systems. Targeted market segments include wireless and wired communications, consumer, automotive, and financial analytics [38], [52]-[56].

Traditionally such FPGA-assisted systems have been built with a static FPGA being used as an I/O controller, datapath element or for acceleration of computational algorithms that can benefit from parallelization techniques due to their ability to realize parallel structures on top of the underlying reconfigurable fabric [53]. Figure 5 shows a typical architecture for this use-case [57]. In such systems the external processor is a master, which uses on-board peripherals such as DDR RAM, Flash, USB controller, various I/O's and others; as well as peripherals or accelerators implemented on the FPGA, such as an Ethernet controller and custom user logic. With these features and an OS such as Linux, such a reference system can target the development of applications such as a network card, remote video display, and networked security camera.

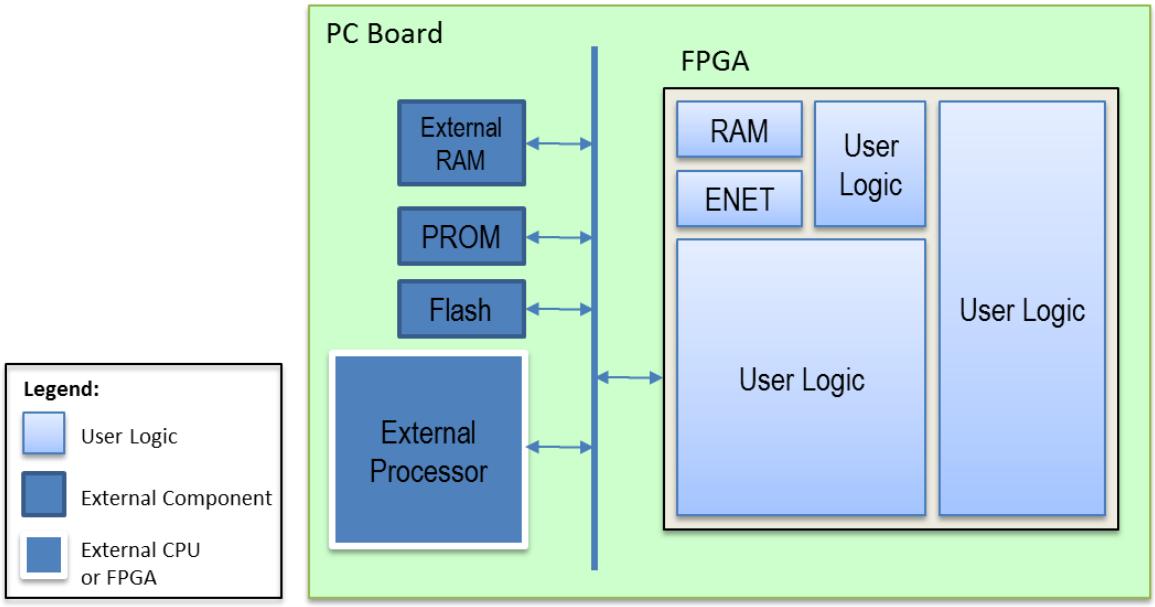


Figure 5: FPGA with external support and without DPR (past)

With the increase of FPGA capacity and development of the new IP, they began absorbing more elements of embedded systems on the fabric including a processor, thereby becoming an integral part of a SoC. Such elements were realized in the form of the hard IP (e.g., PowerPC440, Ethernet MAC, gigabit transceivers, PCIe, DDR interfaces, and fractional PLL blocks) and soft IP blocks including soft processor cores from multiple vendors [25]-[27], [58]-[69].

However, to reconfigure such an SoC, an external entity needs to erase everything from the FPGA and update it with a new bitstream. During this time, the functionality implemented on the reconfigurable fabric is consequentially lost. Thus, with external reconfiguration the state of the system needs to be outside of the FPGA, keeping it a secondary role and increasing the system size and cost due to the requirement for an external controller. This is not acceptable for many applications that need system adaptability to a changing environment.

Motivation to provide a solution for the latter problem led FPGA manufacturers to introduce dynamic partial reconfiguration and an internal hardware reconfiguration controller (HRC) IP, thereby allowing the new DPR-FPGAs to take control of their own reconfiguration [17]-[20]. Figure 6 presents a general embedded system representation that uses a DPR-FPGA SoC capable of runtime hardware adaptability. Such disruptions

in FPGA technology have spawned a number of projects adding support for runtime management of DPR resources [5]-[11], [70]-[74].

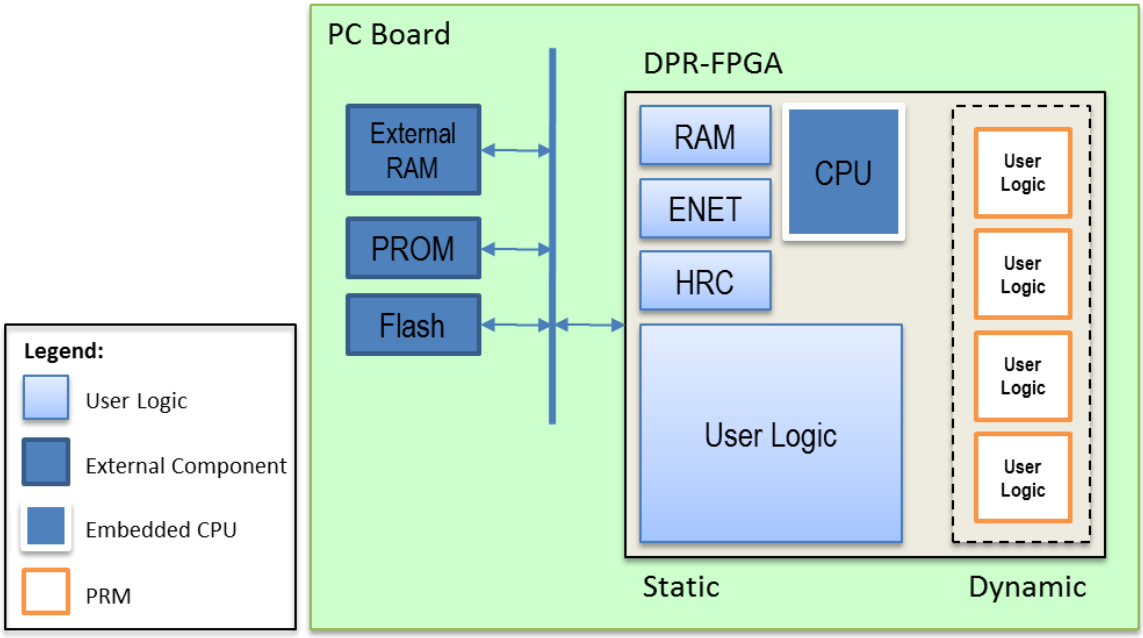


Figure 6: FPGA with DPR and external support (present)

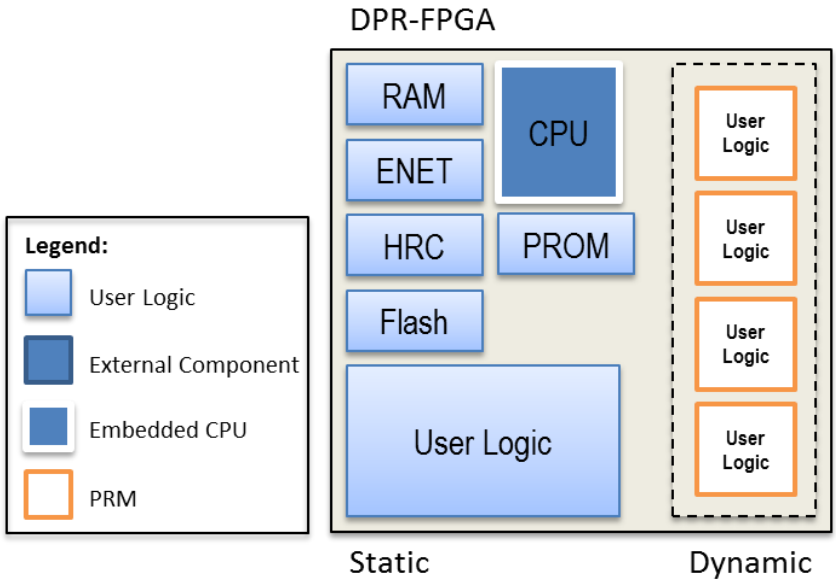


Figure 7: FPGA with DPR and no external support (future)

In the future, the integration of a complete system-on-chip will be driven by the benefits of reducing the bill of materials (BOM), miniaturization and maturation of technology. An example of a DPR-FPGA SoC is illustrated in Figure 7.

With increased system complexity, where boundaries between software and hardware begin to merge, there is a need for more advanced tools and methodologies in order to accelerate development and adaptation of technology, and promote its reuse. A number of developments in this area were done in recent years. However, their motivation, typically coming from acceleration of computation and computer science, resulted in the majority of projects being concentrated around well-established computer science processor centric models, such as threaded architectures.

2.7. Current Operating System Support for DPR-FPGAs

Lee [12] has described that threads and their scheduler impose limitations on ideal just-in-time execution requirements of concurrent control systems. Indeterminism, reliability, power and cost limit the choice of architectures. This is why the concurrency potential of DPR technology and threaded architectures are essentially counterproductive. According to Koopman [34], features of a general purpose processor (GPP) that allow it to be used in a number of applications, oppose the control system principles, especially when considering real-time constraints.

It is important to consider an important issue arising from communication bottlenecks between hardware and a typical operating system such as Linux. Threads communicate with each other via OS-supported objects residing in the address space of a process. These objects are not readily available to hardware. Bergman, et al. [5] and Lübbers, et al. [6] proposed processes and threads, respectively, which run on behalf of DPR modules and can make system calls. However, there is a noticeable slowdown due to communication. For example, a semaphore-based posting methodology, with

data caching enabled, slows down the hardware threads[1] communicating via the OS, as compared to two software threads using the same method [6]. Within Linux, this problem is further exacerbated if delegate threads are implemented in user space due to the additional overhead of copying data between kernel and user space. There are projects that optimized this aspect of the system by mapping main OS primitives to hardware [7], adding direct FIFO based communication between hardware threads [8], and further abstracting hardware interfaces and communication protocols from the software developer within a thread [9] or hardware process context [10]-[11].

2.8. Issues with Threaded Architectures

The real issue is that processor-based architectures utilize threads as an approximation of concurrency. Referring to a typical implementation of a threaded architecture on DPR-FPGA with two hardware threads occupying two partially reconfigurable regions (PRRs), as shown in Figure 8, the following issues are listed when considering their application to adaptive hardware concurrent systems:

- Threads are non-deterministic [12];
- CPU creates a bottleneck as threads are still communicating with/via software;
- The architecture is tightly coupled with CPU/threads; thus, the system and interfaces are built with the CPU in mind.

[1] Hardware thread is an IP core, which can perform a specific task, and has a compatible programming model with software threads, such that an OS can execute either the hard- or software thread using the same OS services.

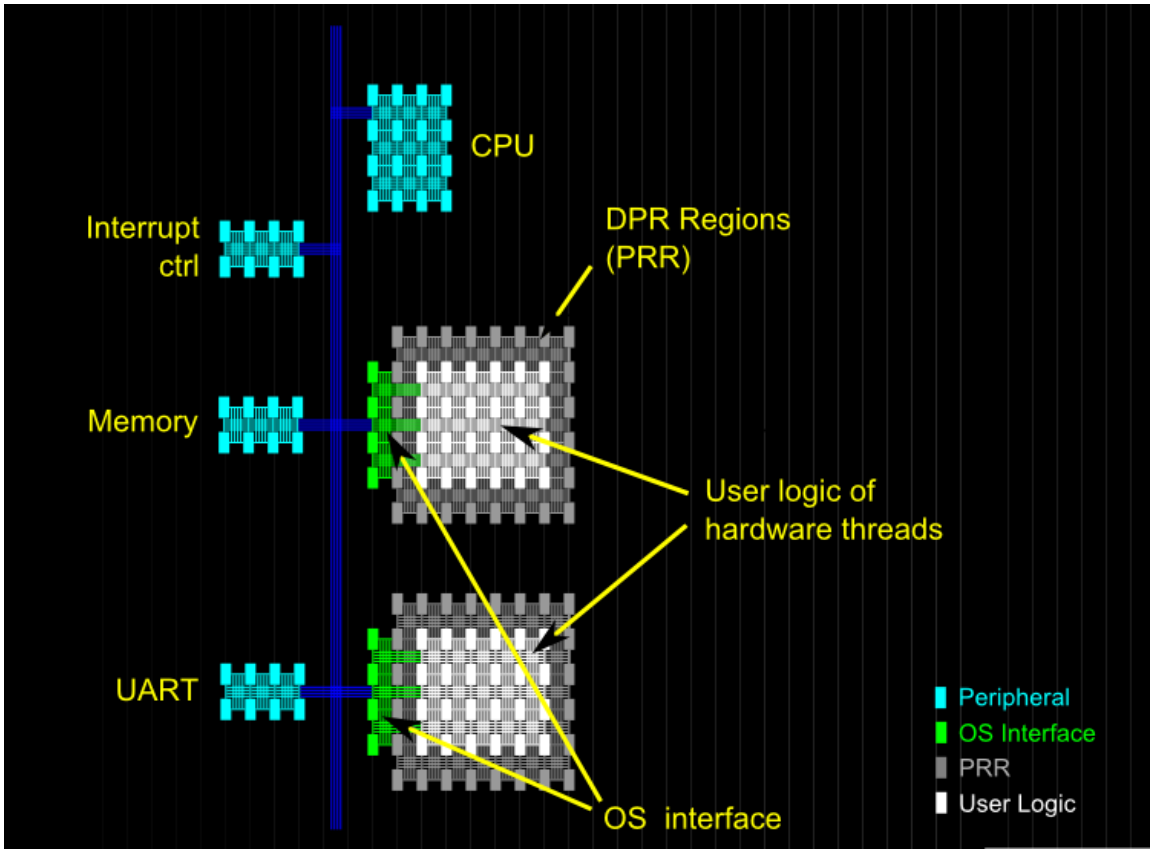


Figure 8: Typical thread architecture on DPR-FPGA

2.9. Adaptive Hardware Concurrent Control and DPR-FPGAs

As has been stated, just-in-time execution requirements make threaded architectures an undesirable choice for adaptive concurrent control applications that have tight power, area, and response time constraints. So far little work has been done in this area. However, the interest is increasing especially in space applications such as described by Michel, et al. [72], where reconfigurability is used among other aspects for better management of power and physical resources, including those on the FPGA. Their work has the closest relationship to this project; however, it differs in several areas that can be improved upon. Firstly, it presents a control system architecture that allows for implementation of processing pipelines in hardware that do not rely on software. It implements modular wrappers for user logic of each PRM. However, it does not decouple the logic from control and data interfaces, as is done in software-based CBD.

This results in less effective integration with higher level systems. Secondly, the system uses embedded software to steer high-level control, but it does not leverage the benefits of using an OS such as Linux with standard well-known facilities. Finally, due to the specific requirements for space applications such as enhanced fault-tolerance, the system is implemented with multiple chips (CPU, static FPGA and Virtex-4 DPR-FPGA). By contrast, this thesis integrates the system on a single Virtex-5 chip, making it a standalone system-on-chip solution that is also implemented with a newer FPGA family.

This thesis does not propose another method for abstracting hardware threads. Instead, it extends features of Linux for DPR support, but without the use of threaded architectures for its core control functionality. The InTERfRAmEwork (TERRAE) [75] presented in this thesis is a solution for connectivity between the hardware version of CBD for DPR IP cores that was previously developed by our team [13]-[14] (e.g., Intraframework), and the external environment. TERRAE is another layer of hierarchy of the scalable adaptive system that abstracts the low-level complexities of the DPR and provides high-level software APIs for application development, without limiting concurrency. It consists of a hardware subsystem and software components. The latter is further subdivided into high-level APIs, drivers and embedded system C code. Thus, the “inter” part of the framework’s name refers to the DPR subsystem connectivity with its external environment and other similar DPR subsystems [75].

3. System Architecture and Design

This chapter examines the architectural requirements of implementing control systems with concurrency. It begins with analysing the requirements for such systems, and then develops possible architectural approaches. It then further examines the design elements for the hardware and software components of such a system and supporting useful tools to assist in this development.

The emphasis of this chapter is on requirements placed upon Linux OS for supporting a fully concurrent system implemented with DPR-FPGAs, while realizing that full concurrency requires unique attributes that threaded systems are not architecturally able to provide.

3.1. System Requirements

Generally, individual control systems have requirements that are specific to their respective application domains. Complicating this matter is the fact that once an implementation technology methodology and supporting components have been settled upon, the resulting solution package places further considerations for additional requirements. Before detailing the general architecture, some of the issues that lead to its choice are examined. While doing this, this section takes into account not only the overall system requirements, but also examines how the underlying potential architectural and design solutions have their own strategic and logical issues when implementing a highly concurrent solution with a DPR-FPGA.

An environment with multiple simultaneous events requires a servicing control system to rely on some degree of concurrency for servicing them. However, for a number of reasons, ranging from legacy methodologies that were very limiting in their day, and the typical constraints of engineering and cost, concurrency has been approximated to various degrees—for instance, utilizing hard or soft real-time systems,

which through stringent control of constraints allowed satisfying application requirements. In cases where more sophisticated requirements for area, power, cost and response time constraints require hardware solutions, these systems have been implemented with custom ASICs, merchant silicon or static FPGAs [1]-[2].

With the advent of DPR technology we can now overcome these pseudo concurrency techniques, such as threads, providing flexibility of software-like reconfigurability and the speeds and power efficiency of hardware. Although DPR-FPGAs do offer this promising ability, these benefits are not readily available due to the lack of tool or tool-chain support. In other words, there is a large gap that needs to be filled before the full potential of DPR-FPGAs can be realized. It is here that additional finer grained requirements are required, so as to provide support for the application's needs for management of dynamically reconfigurable resources such as loading and unloading PRMs, establishing their connectivity and abstracting the PRM management itself from the upper control layers.

Consider the example of an avionic on-board control system and how it can be improved with the help of DPR-FPGAs. Avionics systems for satellites, military aircrafts, inertial guidance systems and even commercial planes have a variety of redundant and backup systems. Extra weight and space comes at a costly premium. For instance, NASA's Stardust deep space probe carries over 100 non-reprogrammable FPGAs [76]. Reduction of components in this case while taking advantage of DPR-FPGA technology can prove to be extremely beneficial. One area where DPR-FPGAs would show enormous assistance is component efficiency for the cases when independent control systems can be staged and do not need to be run concurrently. These components can be loaded as needed, thereby removing the need for physical systems that would have to be either shut-down or put to idle. As a result, DPR-FPGAs could decrease the system's weight, power and increase space utilization by having fewer components. They make the overall system design simpler and more cost effective.

Continuing with the avionics system example, some components might require a pipelined processing hardware architecture in order to meet the timing at each of its stages, such as in high bandwidth acquisition of data that can be sampled, filtered, decoded and processed independently of the CPU. With DPR-FPGAs, it is possible to

provide adaptive digital filtering capabilities that can tune to their environment. For example, the sampling bit-depth can be altered based upon signal bandwidth or noise. DPR-FPGAs answer these requirements by providing a concurrent dynamically reconfigurable hardware fabric. However, they also introduce additional demands such as the development of embedded architectures and tools that enable seamless PRM management for power, better resource utilization, and higher level abstraction.

Arguing for multicore processor solutions, as the complexity of the system increases we reach the limitations imposed by threaded architectures, whether in its operational throughput, response time, power, or area due to the increase in clock frequency or the number of cores and processors in the system [12]. It is not hard to conclude that concurrent control systems, as the avionic controller mentioned previously, would benefit from true hardware-based concurrency.

Finally, in order to integrate a concurrent system with the environment and other systems, and to provide a remote administrator and inter-system access, the availability of a high-level interface and OS support becomes necessary—the latter preferably with a well-known set of connectivity options and system services. Ideally, the OS should not be concerned with managing low-level functions and their administrative overhead. Instead, an intelligent hardware system should be responsible for managing control subsystem communications, while leaving the OS to perform system decisions of configuration, high-level control and interfacing with ambient influences: load/unload hardware, start, stop, reset, request status and send/receive data. Also, the system needs to be able to scale and be flexible enough for reuse in multiple projects. In view of the latter considerations, the requirements for an Adaptive Hardware Concurrent control System (AHCS) that takes advantage of DPR technology are summarised as follows:

- Architecture that minimizes controller response times for control tasks;
- High-level software front-end with familiar interface;
- Network connectivity;
- Scalability;
- Rapid development and reuse;

- Autonomous management of DPR resources independent of processor and threads;
- Independent operation during reconfiguration;
- Ability to connect PRMs with various topologies, including pipeline.

Also, three key architectural features of an AHCS are summarised:

- DPR user logic;
- DPR interfaces for decoupling user logic from architecture;
- DPR architecture.

3.2. Architecture and Design Discourse

This section introduces computer architecture approaches for meeting the above requirements. It begins by describing the desired hardware computer architecture to support concurrency. Software reuse methodologies are introduced and a parallel is shown how they can be applied to building scalable modular control systems with the main control functionality realized in hardware, and how it promotes reuse and rapid prototyping. The discussion concludes with OS support and software architecture.

3.2.1. Single Bus Computer Architecture

Figure 9 demonstrates a potential computer architecture. There are two dynamic PRMs connected to a Hardware Communication Controller (HCC) gateway responsible for intra-module communication. The purpose of the HCC is to enable command and data routing between multiple PRMs and a potential user application while minimizing the contention on the system bus. The HCC can be connected to other similar controllers allowing various PRM connectivity topologies and promoting scalability. Next, the internal reconfiguration port is responsible for internal FPGA partial reconfiguration. External storage is used for keeping partial bitstreams, binary files that represent each PRM configuration. The hardware reconfiguration controller (HRC) takes care of the management of the actual reconfiguration. These blocks and a framework enabling their connectivity should satisfy the basic requirements of a concurrent control system that does not rely on a high-level OS for DPR related tasks.

Next, to enable connectivity with the outside world and provide high-level software front-end for system administration, a processor and DDR RAM are added for hosting and executing an OS and universal asynchronous receiver/transmitter (UART) for console I/O. Also, an interrupt controller connected to the processor is included to enable event driven communication and control initiated by PRM. The communication between different PRMs and between the PR and processor can be message-based in order to pass the control information in-band, promote scalability and allow for the same format of communication between the following: a pair of PRMs; PRM and HCC; and PRM and processor.

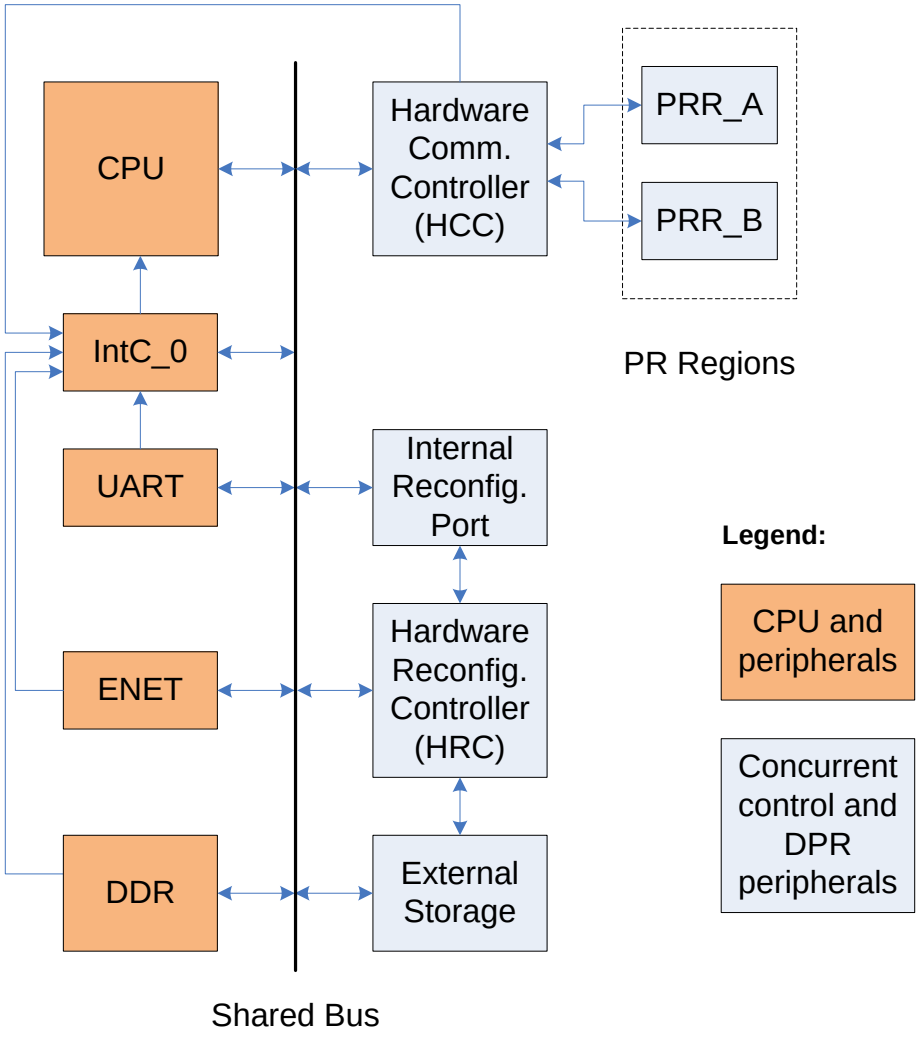


Figure 9: General computer architecture block diagram (shared bus)

Even though the connectivity between components is straightforward, the proposed shared bus architecture has several drawbacks such as HRC complexity and limited scalability. Figure 9 shows HRC connected to both the internal reconfiguration port and the external storage. This is to enable PRR reconfiguration without affecting the shared system bus and the operation of processor and its peripherals. By reading a bitstream directly from the external storage and writing it to the internal reconfiguration port, the HRC reduces traffic on the system bus. However, the complexity of the HRC is increased. The second issue is system scalability. The HCC will have a limited number of PRM ports. When more PRMs are required, multiple HCCs can be instantiated and connected to the system bus. In this case, a pair of PRMs managed by two separate HCCs will be able to communicate with each other, if needed, via the bus. However, traffic contention on the system bus would again be increased, thereby affecting the main processor and its peripherals. Section 3.2.2 addresses these issues.

3.2.2. Dual Bus Computer Architecture

A dynamically reconfigurable subsystem should be both self-sufficient during reconfiguration and self-contained, meaning that it does not adversely affect utilization of available resources such as the system bus. The HCC gateway satisfies this requirement by keeping the intra-PRM traffic away from the bus. The second problem is the contention during reconfiguration when a bitstream file is transferred from the external storage to the internal configuration port. A second path other than the system bus needs to be established, which can be achieved if HRC has direct connections to the external storage and configuration controllers, as seen in Figure 9 above. Alternatively, the processor can be decoupled from the DPR subsystem (DPR_SS) by introducing a second bus as seen in Figure 10. In this scenario two subsystems for DPR management (DPR_SS) and high-level control (HLC_SS) can be defined. The inter-communication between the subsystems can be enabled via a memory mapped bridge and/or a message FIFO. This architecture allows for expansion of each subsystem with additional peripherals without affecting the other [57]. Also, this promotes scalability as multiple HCCs can be connected to the DPR_SS bus without negatively impacting the operation of the HLC_SS. In view of the benefits of scalability and decreased complexity of the HRC, this latter computer architecture has been chosen for this research with details to be presented in Section 3.3.

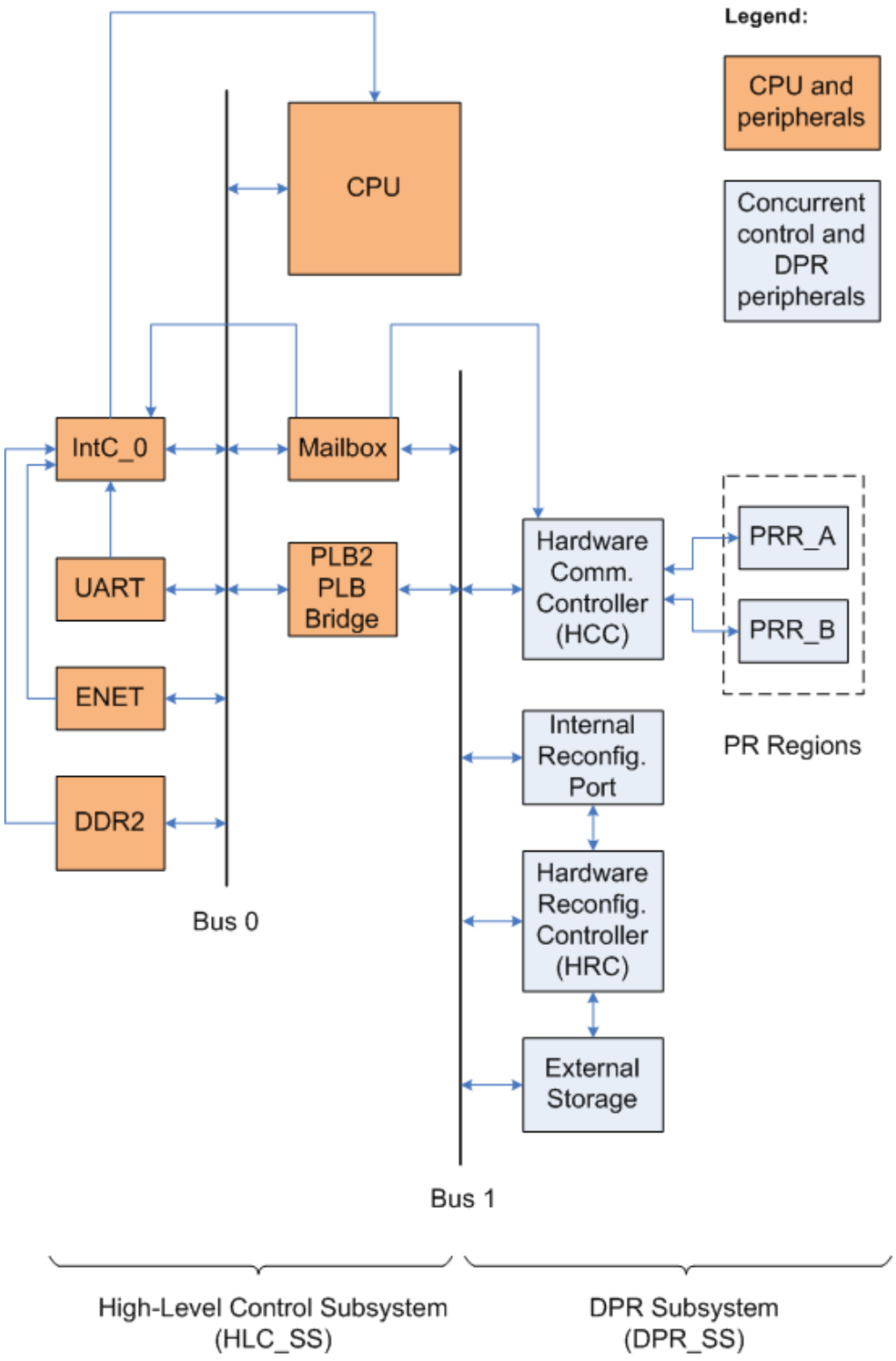


Figure 10: General computer architecture block diagram (dual system bus)

3.2.3. PRM Connectivity and Component-Based Design

In a control system, hardware can be used for many purposes, such as monitoring sensors, acquisition of data, implementation of processing pipeline, or as accelerators. For this reason each PRM can have custom external port requirements including direction and bus widths. Figure 11 shows two external devices employing three PRM modules. The first device uses two PRMs as a processing pipeline, while the second uses a single PRM for sensor monitoring. With such architecture, a PRM can communicate with another PRM via a direct connection or the HCC. The communication with OS occurs via the HCC. Thus, the HCC abstracts the PRM connectivity from the rest of the system.

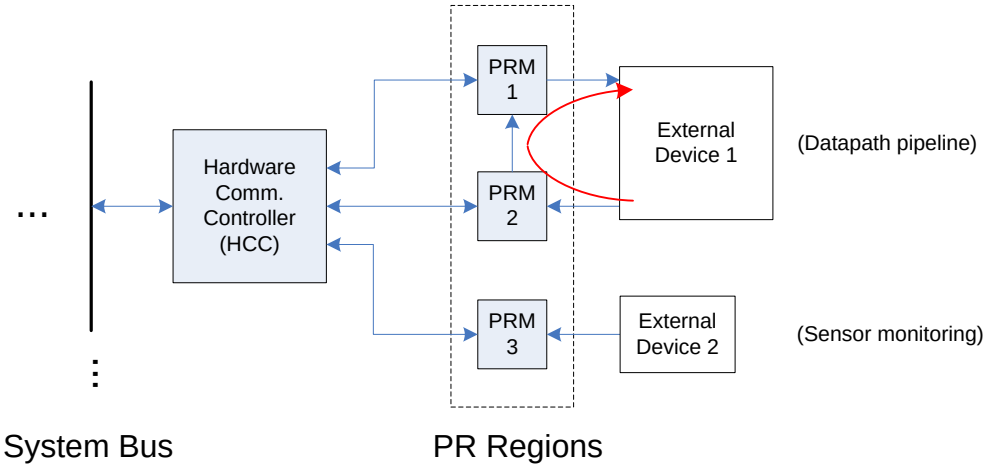


Figure 11: PRM connectivity (pipeline vs. sensor processing)

When considering module connectivity, it is important to examine interfaces and component reuse. Historically, due to a more flexible nature when comparing with hardware, software has established and benefited from software engineering principles such as Component-Based Design (CBD), re-use, inheritance, and other aspects of object oriented programming and similar paradigms. Hardware, however, has lagged due to obvious difficulties of the inflexibility of hardware. Since the introduction of DPR-FPGA technology, hardware has become more flexible. It can now be dynamically and partially reconfigured, similar to loading and unloading software libraries or modules in an operating system. This new dimension of flexibility and thus, complexity, creates a need for a well-structured, hierarchical and more agile representation of hardware. With

numerous similarities between software and modern hardware, it is possible to apply already existing software principles such as CBD to hardware development.

Figure 12 defines a generic PRM with input control/data and output status/data interfaces represented with UML, while Figure 13 demonstrates two PRMs connected to the HCC. The CBD for PRM design and reuse, and a framework providing intra-module communication structure have been introduced and analysed in parallel research [13]-[14].

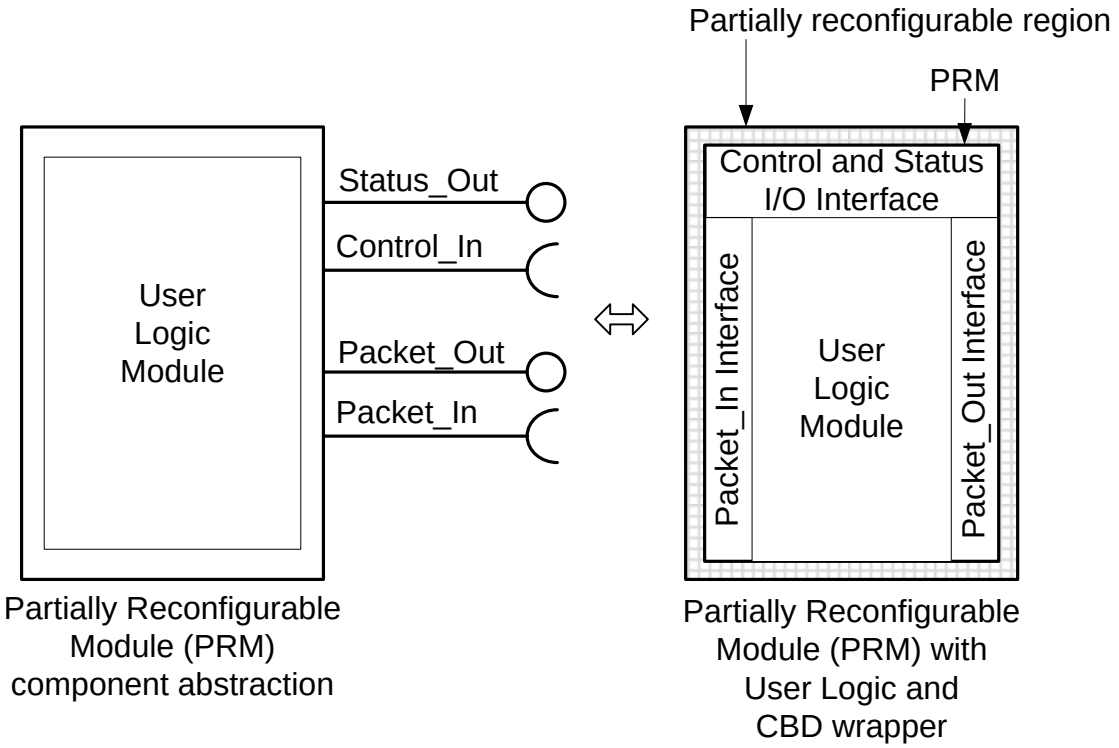


Figure 12: Hardware PRM with standard interface UML

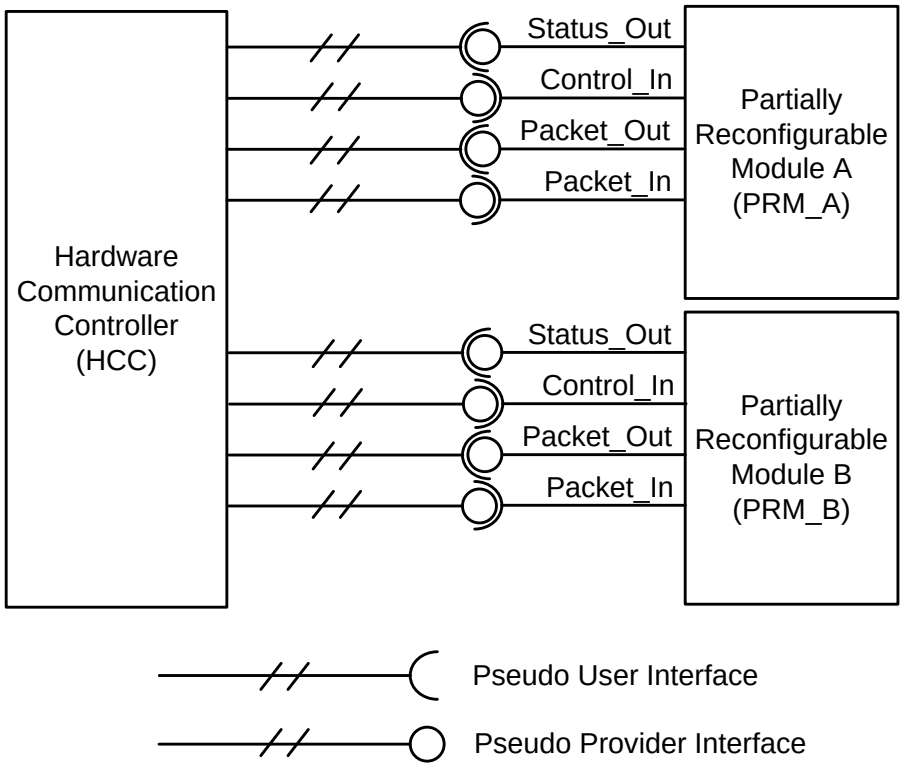


Figure 13: Interface between HCC and user logic UML diagram

3.2.4. Processing Support

3.2.4.1. Processor for Operating System

To support an OS, the processor described in Section 3.2.2 must be capable of providing enough resources to enable the desired system front-end and connectivity to the outside world. Its choice depends on the selected FPGA vendor, throughput requirements, and availability of hardware IP and OS kernel. Some FPGA vendors integrate hard IP processor cores such as PowerPC405/440 and ARM dual-core Cortex-A9 MPCore in some Xilinx FPGA families [24], [26], [60]; ARM dual-core Cortex-A9 and Intel Atom E6x5C with some Altera's families [61]-[64]. It is also possible to use soft IP cores. These typically run slower, use reconfigurable logic resources and occupy a larger area. However, since they are not hardened, there can be multiple instances only limited by the amount of resources in the system. They are also supported by a wider range of FPGA families. Examples are MicroBlaze and PicoBlaze from Xilinx [65]-[66], Nios II and MIPS from Altera [67]-[68], and a number of independent processors including OpenRISC 1200 [69] and Leon [77].

Our lab's inventory included the Xilinx ML505 board with a Virtex-5 LX50T that does not have a hard processor. Also, we did not want to limit the architecture by the number of available hardened processor cores for scalability purposes. Considering a number of factors including performance, size, community support, as well as the popularity and maturity of the OS port, MicroBlaze emerged at the top of the list.

3.2.4.2. Processor for DPR Subsystem

A combination of multiple requirements suggested consideration of a secondary microprocessor uP_1 for the DPR_SS connected to the secondary bus similar to the parallel Intraframework project [13]-[14]. As a reminder, the purpose of the HCC block shown in Figure 10 is to act as a gateway providing hardware-level connectivity between PRMs connected to the same HCC, and also other HCCs when the system is scaled up. The role of the HRC is to enable internal reconfiguration. Neither HCC nor HRC should affect the operation of the main processor with OS uP_0 or its system bus. Also, there is a need to establish a communication channel between the OS of the HLC_SS and HCC/HRC of the DPR_SS. The above features can be realized with the help of a secondary processor uP_1 as described next.

Beginning with HCC, one possible implementation of its functionality is to split it into a hardware controller and a secondary processor uP_1, with the latter serving as glue between the controller and the rest of the system while providing an API. In this case the concurrency of the system must be contained within the reach of the controller, meaning the uP_1 should not be in the path of PRM-to-PRM communication. In this way, the bottlenecks related to hardware-software correspondence can be avoided. Such architecture was chosen and implemented in a parallel Intraframework project [13]-[14], Figure 15. The hardware controller IP core was named Hardware Administrator (HA) and the processor of choice became MicroBlaze. We refer to this processor as uP_1 or HA_CPU, as shown in Figure 16.

Next, the standard way of implementing internal partial reconfiguration functionality of the HRC on Xilinx FPGAs uses the Internal Configuration Access Port (ICAP) IP core and a processor containing a reconfiguration driver that is able to stream a bitstream from the off-chip storage to ICAP, thereby reconfiguring the device internally. Alternatively, there are also a number of other approaches for enhanced hard controllers

that do not use a processor during reconfiguration, free up system bus, and claim faster reconfiguration specs [73]-[74]. The standard approach was utilized in this thesis because the optimization of reconfiguration speed was not in its main focus.

Finally, there are a number of choices for connecting two processors. The most obvious ones that would fit the selected dual bus architecture are based on a bridge or a message FIFO. The Xilinx Mailbox core is one of the available IPs enabling inter-processor communication (IPC) via a pair of FIFOs, and creating a duplex communication channel with configurable interrupts at each end. Due to the out-of-the-box availability of the above-mentioned IP, and since the first goal of the project was the proof of concept implementation of a concurrent control system without speed optimization, it was decided to adopt the standard Xilinx IP component.

It is important to note that a processor is a sequential device that inherently opposes the principles of concurrency defined earlier. However, in this case the concurrency and communication between PRMs is abstracted by the HA, so the potential bottleneck is not the PRM-to-PRM communication, which is the main interest, but rather PRM-to-OS communication via the secondary UP_1 and the Mailbox. We consider PRM-to-OS communication to be of lower importance as we are not trying to accelerate software in DPR hardware and thus did not optimize this path in the system.

3.2.5. Operating System

To equip a concurrent control system with a high-level interface to the outside world, an OS is desirable. As described previously, our system needs a basic I/O, serial communication, remote connectivity via an Ethernet controller and a TCP/IP stack. There are a number of operating systems available for the Xilinx MicroBlaze architecture on FPGAs [55]-[56], [49], [77]-[83]. With its out-of-box support for TCP/IP, free licensing, a large development community and existing FPGA ports supporting Xilinx peripherals, PetaLinux distribution (v0.40 RC4 by PetaLogix [49]) became the OS of choice. It is based on the 2.6.20 Linux kernel and includes cross-compilation tools that accelerate board support package (BSP) generation for Xilinx FPGAs. In addition, it provides tools for faster kernel module and application generation. MicroBlaze and PetaLinux events

related to this thesis are interposed on a timeline with DPR-FPGA technology development as shown in Figure 4.

3.2.6. Software Systems

This section describes a general software systems architecture. The benefits of CBD paradigm for a reuse-based approach to design scalable systems from smaller components have been already stated. It was applied to the creation of a DPR_SS architecture. A similar approach is used while defining a software subsystem architecture with loosely coupled interfaces. Communication between hardware and software components can be enabled with a packet-based approach. This is also encouraged by the scalable nature of the framework and implementation flexibility.

In order to support the use-cases introduced in Section 3.1, the following general components are required. A command line tool (CLT) is a user interface that is able to parse user commands and data and create packets. Furthermore, the DPR supporting service is desired to be running in the background while keeping DPR system status and serving as a gateway between the DPR subsystem and the high-level software. It must be able to be loaded and unloaded on demand. This service is depicted as a daemon process (DP) in Figure 14. Thus, the CLT can talk with the DPR subsystem via the DP.

As previously discussed, the DPR subsystem has a secondary asymmetric processor HA_CPU. A software agent running on the HA_CPU needs to have a routing functionality to pass the messages to PRMs via HA, thereby completing the implementation of HCC as shown in Figure 10. It must also assist the ICAP in order to realize the HRC functionality. Finally, this agent can contain the code for glueing the DPR subsystem's interface to the rest of the system, such as via the previously mentioned Mailbox hardware IP block. The combined functionality of this software agent is referred to as HA_CPU Application (HCApp).

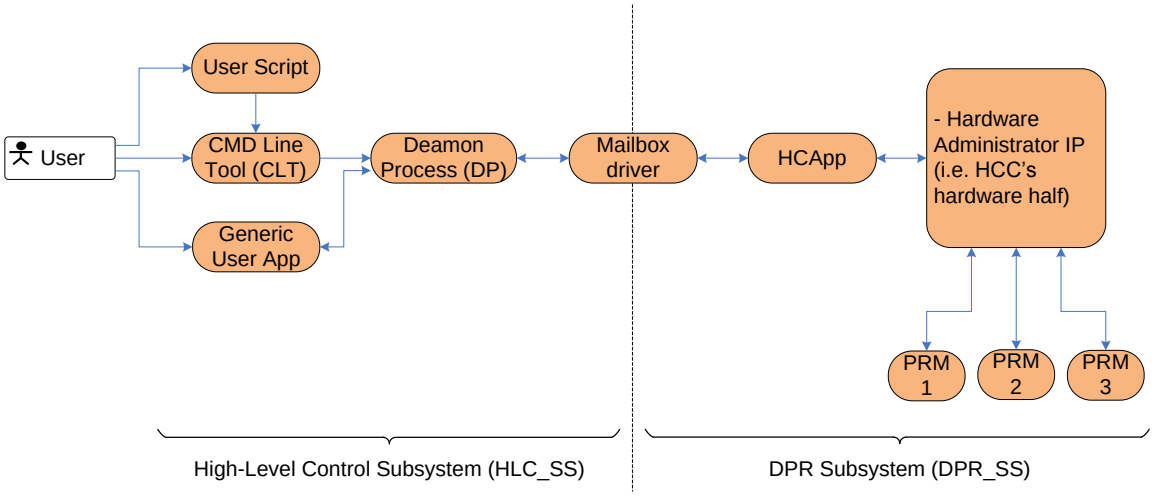


Figure 14: General architecture UML use-case diagram

3.3. Decided Architecture

Given the general hardware and software architecture described in the previous section, this section gives details of the final architectural. It further breaks down the system into smaller components in order to support the requirements. It starts by introducing a single bus concurrent control system developed in parallel research [13]-[14]. Then it expands into a dual bus computer architecture that integrates this work into a framework that enables its high-level control. The section concludes with a description of detailed software systems architecture and its components including operating system, library, drivers and applications.

3.3.1. Intraframework for DPR Access

In references [13]-[14], a scalable Intraframework was developed for management of DPR resources and abstraction of the implementation details of user logic cores within the DPRs. "Intra" refers to intra-hardware-module communication. This framework separates the FPGA fabric into the following two partitions: static system supporting resources for DPR and dynamic with PRRs. These two areas are interfaced via a Hardware Administrator (HA) Intellectual Property (IP) core as shown in Figure 15. Packet control and data communication and a modular framework facilitate the creation of scalable embedded systems with standardized interfaces at each level of hierarchy as

shown in Figure 12 and Figure 13. Such systems are easier to decompose, design, test and integrate, providing a component-based design (CBD) for hardware, as in software, where CBD and agent paradigms are used.

Figure 15 shows that Intraframework is a single bus system with a single processor connected to HA. It is implemented on the Xilinx FPGA and uses IPs that will be discussed in Section 4.2. The MicroBlaze processor in this example system serves two roles. First, it assists internal reconfiguration by reading bitstreams from a compact flash card and streaming them to the internal configuration port. Second, it provides a user command line interface to the system. The project described in this thesis builds a computer architecture and software system on top of the Intraframework to enable high-level control without affecting the concurrent properties of the Intraframework.

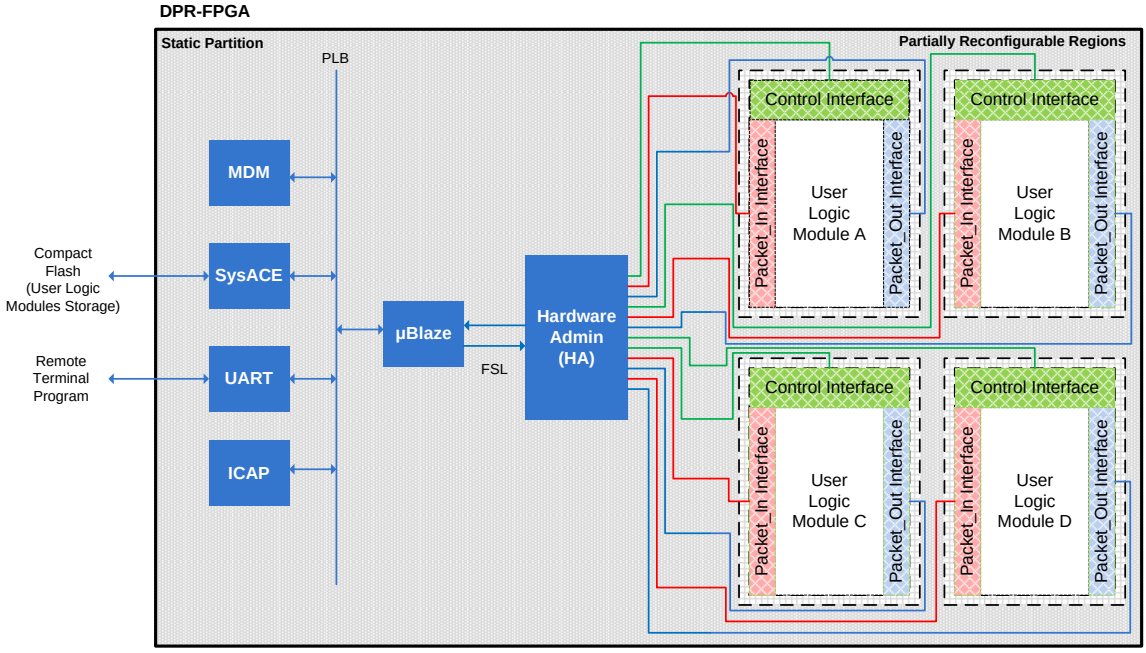


Figure 15: Intraframework block diagram

3.3.2. Computer Architecture of TERRAE

InTERfRAmEwork (TERRAE) [75] is a solution to the previously defined problem of connectivity between the DPR system described in Section 3.3.1 and the external world. It is another layer of hierarchy of the scalable adaptive system that abstracts the low-level complexities of the DPR and provides high-level software APIs for application

development. It consists of hardware subsystem and software components. The latter is further subdivided into high-level APIs, drivers and embedded system C code. Thus, the “inter” part of the framework’s name refers to DPR subsystem connectivity with its external environment and other similar DPR subsystems.

The architectural decisions provide generic solutions for a control system. The environments in which these systems operate often require network connectivity for remote control and system maintenance. For example, an administrator could request a change in the mode of its operation, which requires a live hardware upgrade. A new version of hardware in the form of a bitstream will be sent to the system followed by a switch over command. Meanwhile, the system must be able to respond to further requests both from the administrator and the controlled equipment.

The latter scenario requires the decoupling of high-level and low-level functionalities that ensure that time constraints of the system are satisfied. For this purpose, a dual architecture has been chosen to eliminate bus contention during reconfiguration that is currently driven by HA_CPU, and also communication bottlenecks between multiple HAs if they are connected to the same bus (see Figure 16). The first bus contains the LINUX_CPU running the Linux operating system, an external RAM controller with instruction and data caching, and UART for communication with a user and an interrupt controller. An Ethernet port will be included in a future version of the system for remote connectivity. The second bus has components from the Intraframework (HA_CPU attached to HA with PRMs, internal reconfiguration port, and external storage controller for access to external file system) as well as additional components added for the TERRAE such as an interrupt controller to avoid polling. All PRMs reside inside Partially Reconfigurable Regions (PRRs) and the rest of the system is static and configured at system bootup.

The two buses are connected via a bridge and a mailbox. The bridge is used by LINUX_CPU for accessing external storage on the second bus and by HA_CPU sending debugging messages to UART. The mailbox has two independent FIFOs providing duplex inter-processor communication. Its simple memory-mapped interface allows for generic character device driver design. Both processors can be interrupted when its mailbox is not empty. It is also the boundary between the control and data channels,

and a single packet stream. While the Intraframework has an independent control channel for prioritizing communication between PRMs and HAs, the TERRAE administration layer is not expected to send extra-long packets that would significantly stall command communication.

In a typical remote live hardware update scenario, an OS on LINUX_CPU receives a new bitstream over Ethernet, in other words, a binary representation of the PRM circuit instance. The bitstream is stored off-chip on a flash card. Subsequently, a user command is issued to update a PRR with a new PRM. At this time, the OS sends only a command packet to HA_CPU via the Mailbox that in turn reconfigures the PRR of interest when it is available. During reconfiguration the second bus takes the load of transferring the bitstream from the external storage to PRM via the internal reconfiguration port without burdening the first bus. In this manner, the OS and peripherals on its bus are free to perform other system wide tasks.

The combined TERRAE/Intraframework can be represented in a single block diagram shown in Figure 16. Referring to the diagram the boundaries of the above mentioned frameworks are defined as follows. TERRAE includes: the dual bus architecture; all IP components of Bus 0; a second interrupt controller IntC_1 connected to HA_CPU; Linux OS, drivers and applications on LINUX_CPU; and an embedded application on HA_CPU. The re-used components of the Intraframework are the HA, internal reconfiguration port, external storage controller, HA_CPU and PRMs. As will later be explained, both HA_CPU and PRMs were modified for compatibility and demonstration purposes of the combined frameworks.

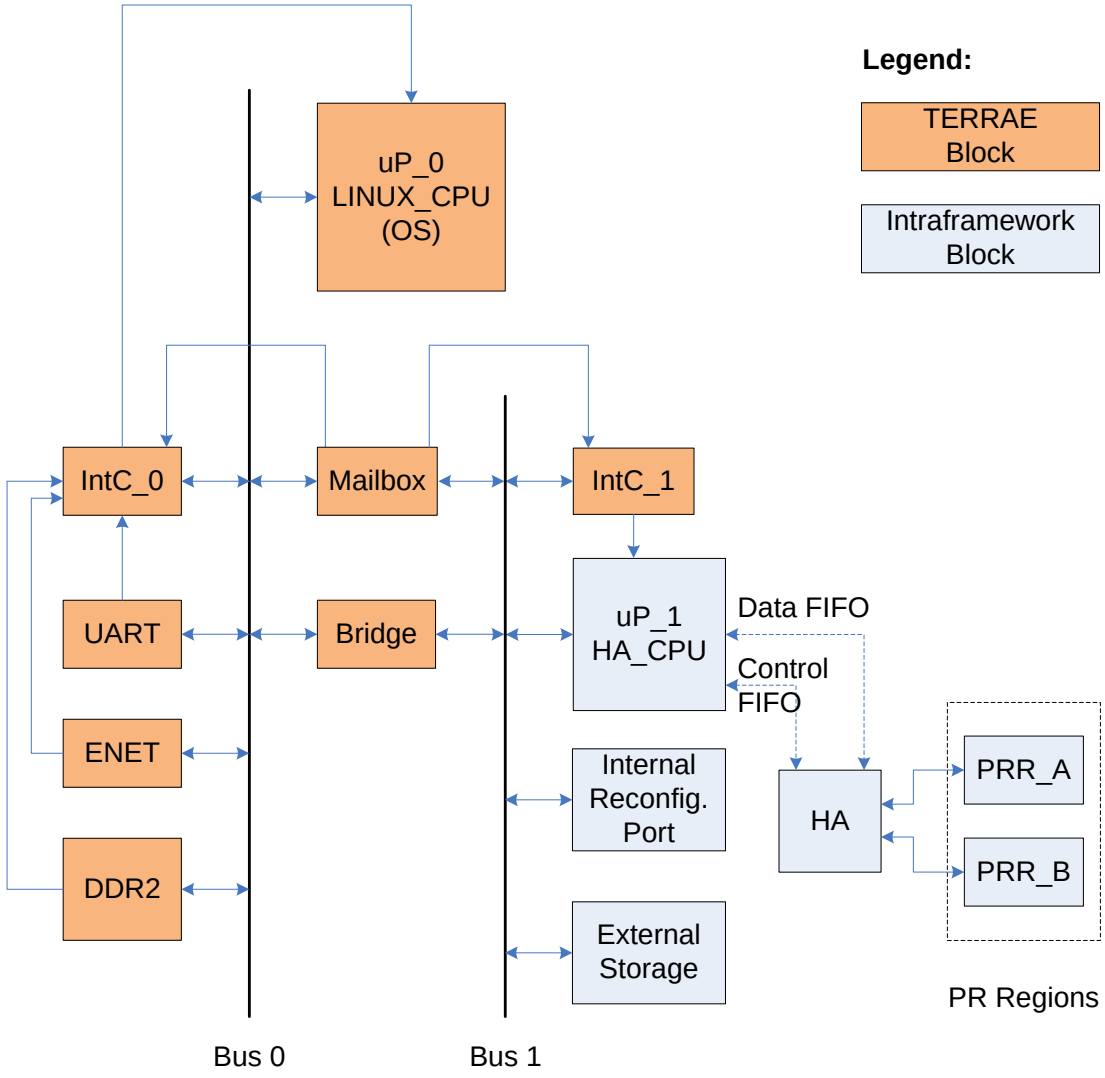


Figure 16: Specific computer architecture of TERRAE/Intraframework

3.3.3. Software Components of TERRAE

To drive the above hardware architecture, a number of soft system components were developed. Dedicated drivers for the Mailbox and packet communications, together with higher application support libraries were created to enable a user or application to interface with the DPR system.

Since there are two processors in the system, the software partitioning has been done as shown in Figure 14. Command line tool (CLT) and daemon process (DP) live in the user space of Linux that is running on LINUX_CPU. The advantages of running the

agent applications in the user space are as follows: ability to link against the full C library; easier debugging; and avoidance of kernel crashes. However, there are also disadvantages, such as slower response time, limited direct access to memory and the absence of interrupts [43]. Even though performance is an important attribute, for the purpose of this project the benefits provided by the user space outweighed the drawbacks.

A Linux version of the mailbox driver is handling the communication between user space, kernel space and hardware. The second processor, HA_CPU, runs the HA_CPU application (HCApp) and an embedded C version of the mailbox driver. The following subsections outline these tools and soft systems. As seen from Table 1, each of the applications serves the following purposes:

Table 1: Software components of TERRAE

Application	**Purpose**
Command Line Tool (CLT)	Command line processing; Creation of packets; Sending packets to daemon process.
Daemon Process (DP)	Maintains a PRR status table; Based on packet header information, it either forwards packets between CLT and HA/PRR or executes packet commands locally.
Mailbox drivers	LINUX_CPU: kernel loadable module that forwards user space's messages to mailbox IP (Netlink and soft Linux IRQ driven) and vice versa (mailbox's IRQ driven); HA_CPU: forwards messages generated by HA_CPU application to mailbox IP and vice versa (mailbox's IRQ driven); Both versions of the driver buffer one full message received from the mailbox IP.
HA_CPU interface application (HCApp)	Non-blocking PRR reconfiguration, meaning LINUX_CPU and its bus are not busy while a PRR is being reconfigured by the HA_CPU on the second bus; Conversion of control and data FSL channels interfacing the HA to/from a single packet bus for mailbox communication; Round robin processing of mailbox received and scheduled for processing messages and control/data FSL channels; Mailbox interrupt handling; Based on packet header information either forwards packets between the daemon process and HA/PRR or executes packet commands locally (i.e. reconfiguration).

3.3.3.1. PR API

PR API (*libpr*) is a collection of functions written in C that form the basis of the TERRAE's software component. As shown in Figure 17, it consists of command line APIs (pr_user_comm), status table management tools (pr_stat), functions for packet making, classification and routing (pr_packet), a communication interface with hardware drivers (pr_comm), printing and debugging utilities (pr_util). The source code is reused by Command Line Tool (CLT), Daemon process (DP) and HA_CPU application (HCApp). The library has a set of platform specific function and macros allowing the use of the same APIs when compiled for Linux or as a standalone application (pr_sys component).

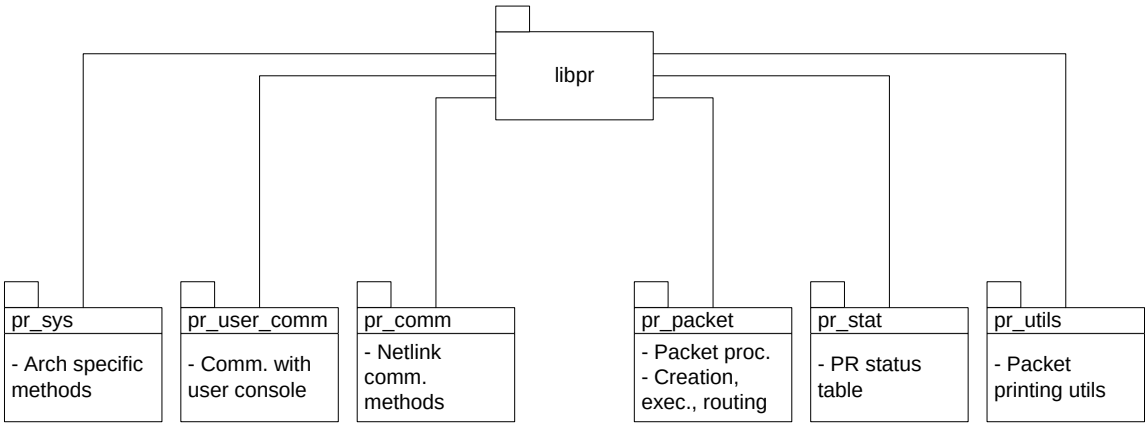


Figure 17: libpr library UML class diagram

3.3.3.2. TERRAE Connectivity

Referring to Figure 18, a user sends a command via CLT running on LINUX_CPU, which is forwarded to DP using *libpr* functions. Then, the DP passes the command message to the mailbox driver. Subsequently, the message travels through the mailbox to HA_CPU, goes through the driver and reaches the HCApp. At this point, the command can be either executed by the HCApp or forwarded to one of the PRMs. In the latter case, the command travels from the HCApp to PRM via the HA API of *libpr* and HA. The communication in the opposite direction follows the same path. Finally, a generic user application can be created. It can communicate with the DP via the same API of *libpr*.

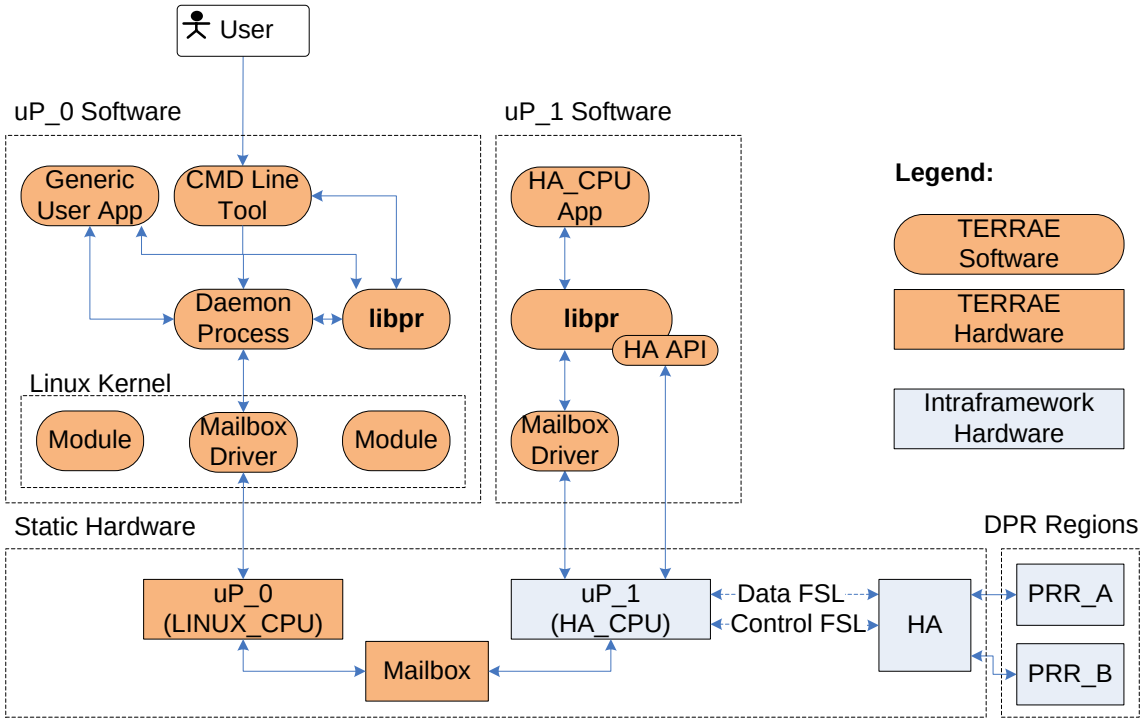


Figure 18: Software agents' connectivity in TERRAE

3.3.3.3. Packet Communication

For communication between the software and hardware agents of the TERRAE (i.e., CLT, DP, HCApp, HA, PRM), a packet-based communication functions has been chosen. It provides flexibility in implementation of connectivity between the agents, allowing for scalability, and enabling the packets to be tunnelled between TERRAE systems on different FPGAs to allow inter-system control. The benefits come at the cost of additional memory usage and the slower processing speeds resulting from additional message buffering.

The messages in the framework travel from one agent (i.e., either a software application, or a hardware module running in PRRs) to another. A typical path is from the CLT through DP, mailbox, and HCApp. Then, depending on the number of HAs and PRMs, the message is routed to the appropriate hardware module, addressed by HA-PRM ID pair. The messages can also travel back, from PRM to DP and possible user applications. The DP and HCApp have the ability to snoop on the packets and decide whether they have reached their destination. This decision is based on the command embedded in the packet and the responsibility of the agent. For example, if a

reconfiguration command arrives at HCApp, it will not be passed to HA/PRM, but instead will be executed locally, because this is one of the responsibilities of the HCApp. Figure 19 shows a UML action diagram with packet interfaces between the agents, and snooping capabilities of the DP and HCApp.

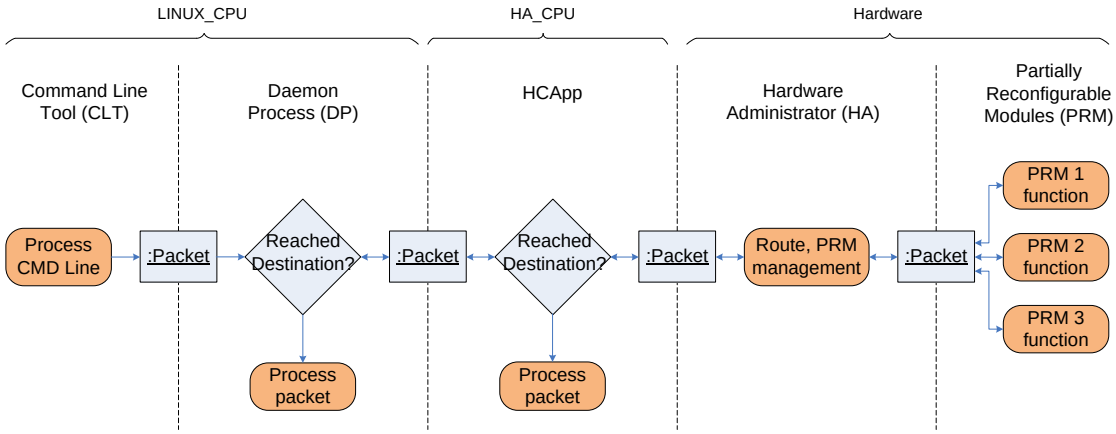


Figure 19: System UML action diagram (packet interfaces)

3.3.3.4. System Packet Data Flow

The result of using a message passing mechanism requires additional inter-agent buffers. As seen from the system data flow diagram in Figure 20, there is a pair of TX and RX buffers between each agent except for CLT and DP. Since the CLT is used for creation and pushing of user commands into the system, there is only a need for TX buffer. The diagram also shows a potential approach to extend the system for additional user applications and enable two-way communication with the DP. If such applications are desired, they would also require two-way buffering for sending and receiving the messages.

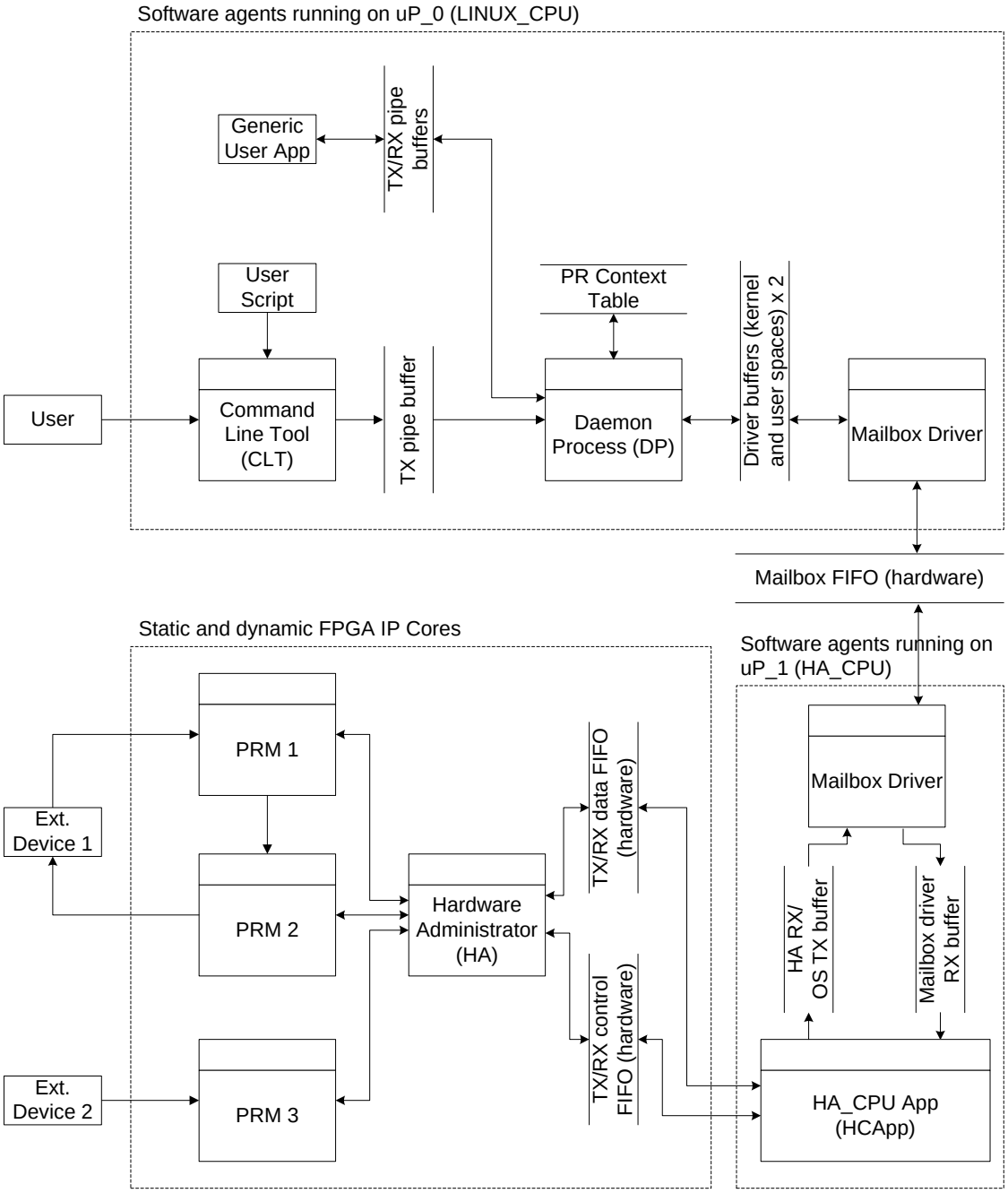


Figure 20: System data flow and store diagram

3.3.3.5. Synchronous Communication

With the packet based system introduced in Section 3.3.3.3, it is possible to implement asynchronous communication between the agents. Such communication would allow multiple command launching and faster system response time as compared with synchronous communication. However, since the first goal of this research is the

demonstration of an OS-based high-level control over dynamic reconfigurability of the FPGA fabric, the system can be simplified to initially support only synchronous communication. This means that several requirements can be relaxed. There is no need for additional buffering and related control logic to handle multiple packets and their flow control. Thus, the requirement and assumption for all agents of the system is that each buffer will contain one packet at a time until it is processed.

3.3.3.6. Command Line Tool (CLT)

CLT is a tool that was developed to provide a front end script based control of the system. It resides in the user space of Linux and supports generic commands for reconfiguration and status as well as custom commands for User Logic. The library can be extended to support more commands specific to user developed PRMs. Table 2a lists arguments recognized by the tool. Table 2b lists general control commands, and Table 2c control commands supported by a Counter PRM example.

Table 2: TERRAE commands

Command Switch	ID	Description
Table 2a. Command line switches of CLT		
-ctl <command> -dest <dest_ha> <dest_prr>	-	Send a command to PRR, where the command is as defined in Table 2b-c. dest_ha and dest_prr are destination HA and PRR respectively.
-ctlid <command_id> -dest <dest_ha> <dest_prr>	-	Send a command to PRR, where command_id is the ID of a command as defined in Table 2b-c. dest_ha and dest_prr are destination HA and PRR respectively.
-bitstream <bitfilename>	-	Send reconfiguration command to HA_CPU, where bitfilename is the name of a bitstream for configuration of PRR. The bitstream already incorporates the HA and PRR. Equivalent to using "-ctl CMD_RECONFIG_PRR –dest <dest_ha> <dest_prr> <bitfilename>".
-data <datafile> -dest <dest_ha> <dest_prr>	-	Send the contents of data file to HA/PRR defined by dest_ha/dest_prr.
Table 2b. General control commands of CLT		
CMD_PRR_SUMMARY	1	Request status summary for PRR
CMD_RECONFIG_PRR	65	Configure PRR with a specified bitstream found on the external storage. This command assumes the

		file is already stored in a file system

Table 2c. Example of custom control commands: Counter PRM

CMD_START_INCREMENT	10	Start incrementing the counter
CMD_STOP_INCREMENT	11	Stop incrementing the counter
CMD_IDLE	13	Set PRM's state machine to IDLE state
CMD_LOAD_RANDOM_SEED	14	Load random starting value
CMD_RESET	15	Reset PRM
CMD_LOAD_STARTING_VALUE	16	Load the starting value of the counter
CMD_LOAD_FINAL_VALUE	17	Load the final value of the counter

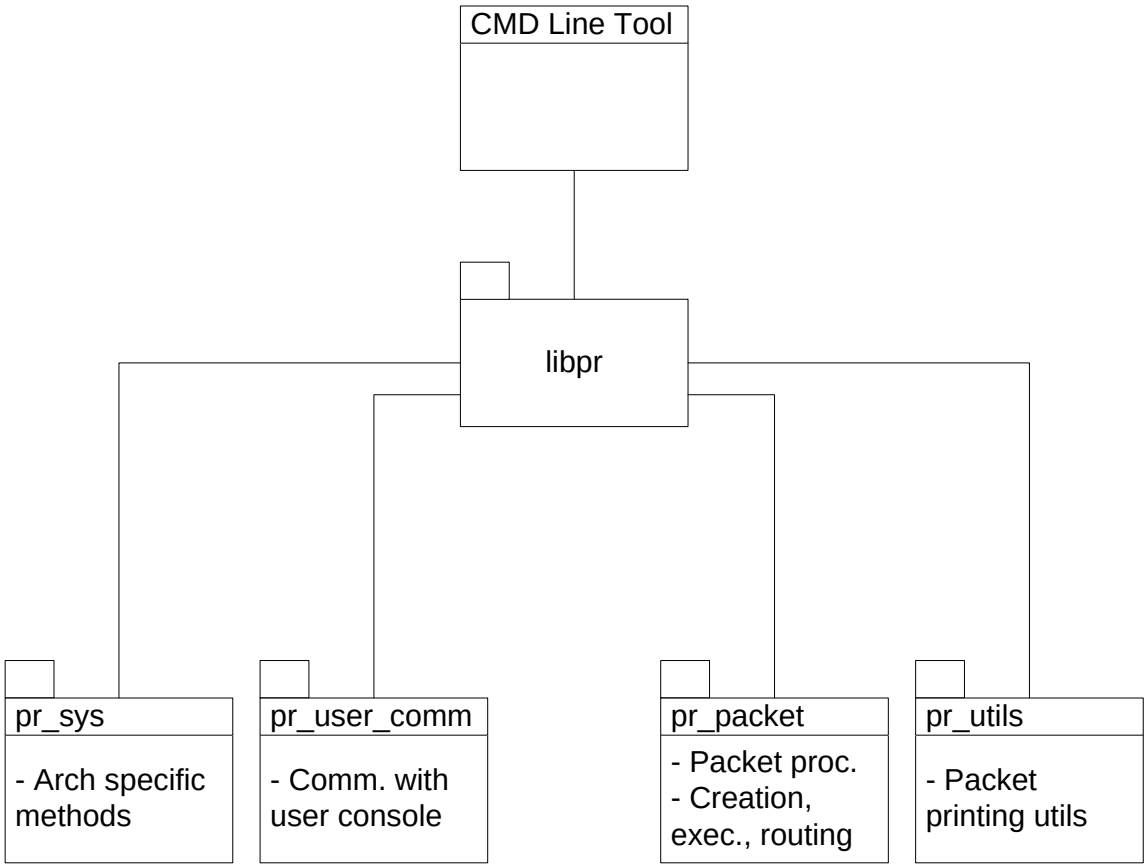


Figure 21: Command Line Tool (CLT) UML class diagram

The CLT converts user commands into control or data packets and sends them to the DP discussed in Section 3.3.3.7. The CLT consists of several packages that are part of the *libpr* as seen from Figure 21: pr_user_comm, pr_packet, pr_utils and pr_sys. The CLT utilizes functions from the pr_user_comm package for communication with the

DP. Pr_packet is used for generation of control and data packets. The pr_utils contains packet printing utilities. Finally, pr_sys has APIs that enable communication with other agent applications via interfaces that are specific to the target LINUX_CPU, i.e., named pipe communication with the DP agent.

The activity diagram in Figure 22 shows the flow through the CLT functions. First, user command line arguments are processed to derive an action: CTLID, CTL, DATA, or RECONFIG. Then, the created packets are sent to the daemon process by calling a pipe_send_packet_to_ha() function.

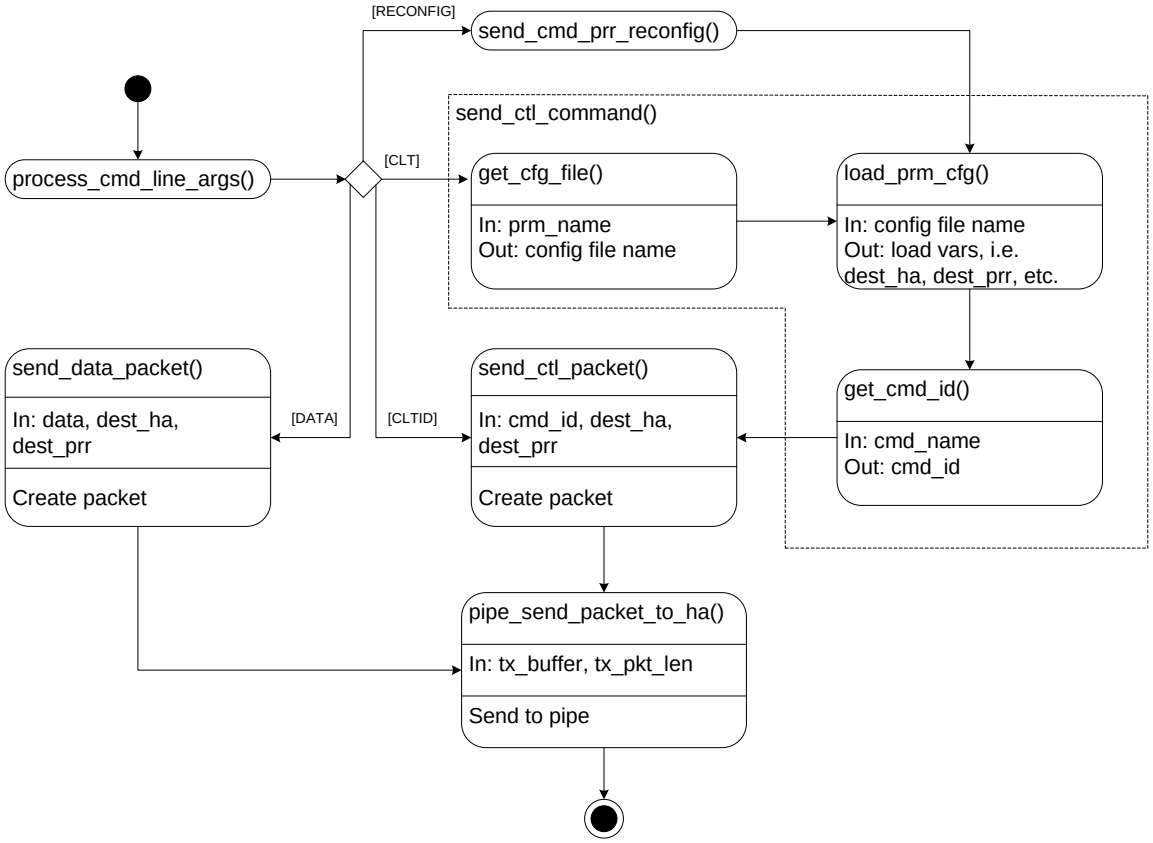


Figure 22: Command Line Tool UML activity diagram

The first supported action—CTLID—is responsible for creation of a command control packet based on a command ID. The command ID specifies a command to execute and is defined in Table 2a-c. Send_ctl_packet() function creates a control packet with command ID and destination HA and PRR.

The second supported action—CTL—takes responsibility of converting name identifiers for command and PRM, converting them into CTLID, HA and PRR, and creating a command control packet. Specifically, this path triggers execution of the send_ctl_command() function which in turn looks up the configuration file from the specified command name via get_cfg_file(), followed by deriving the values of destination HA, PRR and CTLID via load_prm_cfg() and get_cmd_id(), respectively.

The third supported action—DATA—is the creation of a data packet. Send_data_packet() function creates the packet with proper destination HA and PRR, data length and the data to be sent. The third supported action is a reconfiguration command itself. It is equivalent to calling a CTLID action with CMD_RECONFIG_PRR command to create a reconfiguration control command packet. It has been kept for historical reasons as it is the original command with which reconfiguration was tested.

3.3.3.7. Daemon Process (DP)

DP tracks the status of the DPR subsystem and resides in the user space of Linux. It forwards packets received from CLT to HA. Instead of allowing CLT to send packets directly to HA via Mailbox, DP serves the role of a router that can be extended to forward packets to other instances of HA_CPUs via Mailbox or other embedded systems via Ethernet. As shown in Table 3, the data is indexed by PRR, which at any given point in time can contain a single instance of PRM. Table 4 summarizes the fields of the table.

The UML class diagram in Figure 23 shows the *libpr* packages used by the DP: pr_user_comm for communication with the CLT application; pr_comm with mailbox driver described in Section 3.3.3.9; pr_packet for packet processing; pr_stat for PRM status table; pr_utils for packet printing utilities; and pr_sys for defining APIs, enabling communication with the CLT on LINUX_CPU and HCApp on HA_CPU.

Table 3: PR status table

Bitstream Management				Status	
HA	**PRR ID**	**Bitstream Name**	**Instance Name**	**Empty**	**Status**
0	0	Adr0.bit	Adder0	0	ACTION
0	1	Rnd1.bit	Rand0	0	READY

Table 4: PR Status table's fields

Field	Description
PRR ID	ID of PR region
PRM ID	ID of PR module
Bitstream Name	File name of bitstream configuration file
Instance Name	User selected name of HA's instance
Empty/Loaded	Empty/Loaded status of PRR
State	State of PRM customized by User Logic, i.e. RESET, IDLE, READY, ACTION, PAUSE, STOP, DONE.

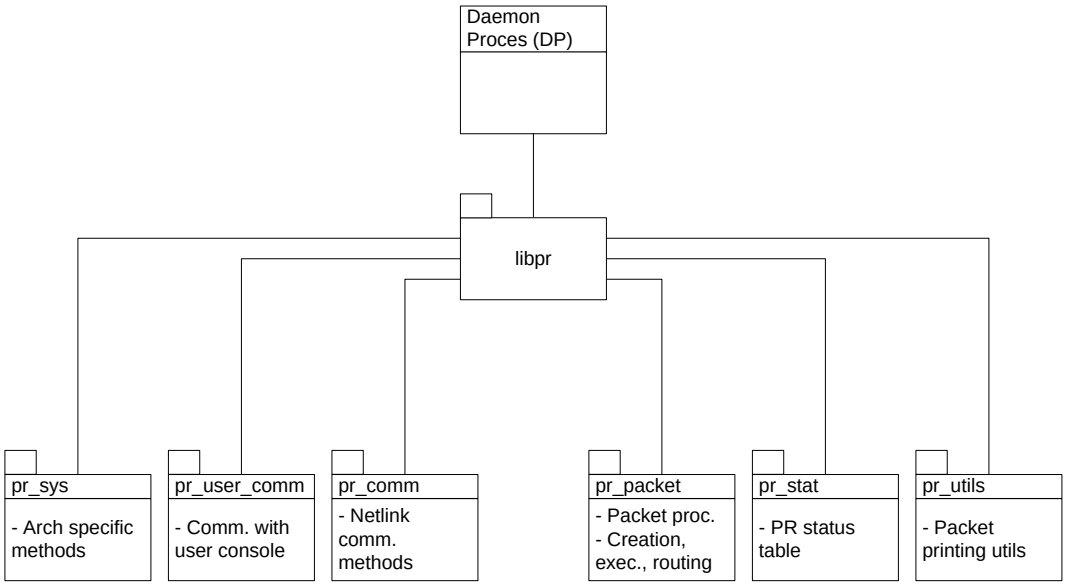


Figure 23: Daemon Process (DP) UML class diagram

Figure 24 below shows UML activity diagram of the DP. It is capable of receiving messages both from the user space and the kernel space alternating in a round robin. The internal packet router infers the packet destination from its class (control versus data), type (command versus status) and command/status ID according to Table 5. Packets that have reached their destination (i.e., destined for DP) are processed locally, while others are forwarded to either the user application or HA_CPU via the mailbox driver. As seen from the table, commands have one destination, yet can be created by different agents. This implies that for these commands, the framework allows for either OS or HA_CPU to initiate the communication, depending on the use-case of system application.

Currently, only one user application communicating via DP is supported, the CLT. To extend the support for multiple applications, the DP would need to either keep track of agent IDs or broadcast the messages to all applications.

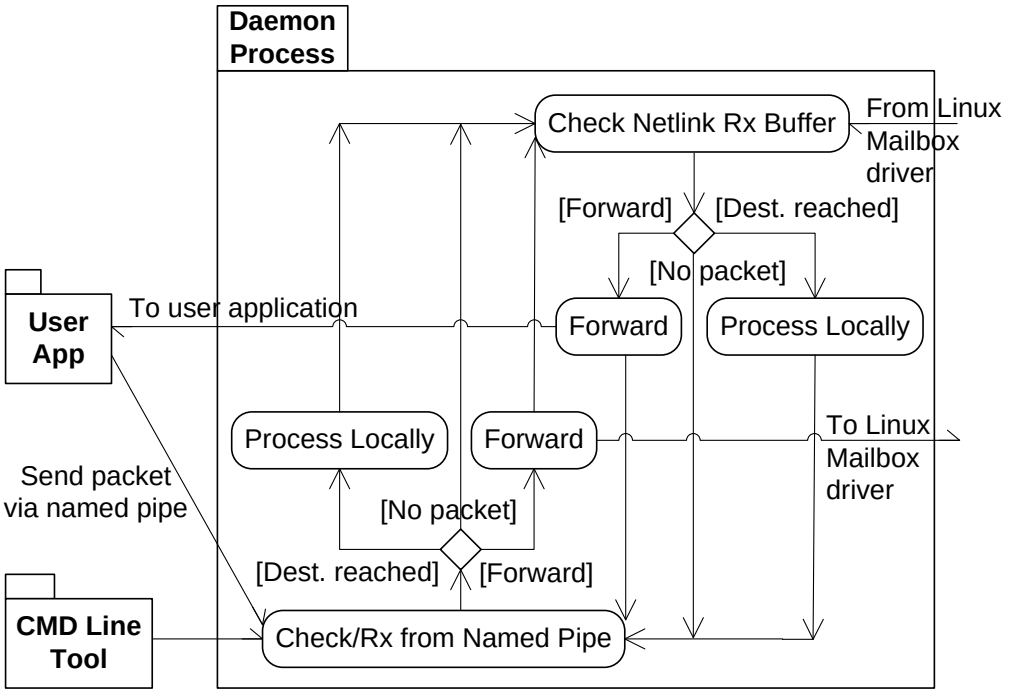


Figure 24: Daemon Process UML activity diagram

Table 5: Control packet routing based on ID of example counter PRM

Command/Status ID	Source	Destination
CMD_PRR_SUMMARY	OS, HA_CPU	PRR
CMD_RECONFIG_PRR	OS, HA_CPU	HA_CPU
CMD_START_INCREMENT	OS, HA_CPU	PRR
CMD_STOP_INCREMENT	OS, HA_CPU	PRR
CMD_IDLE	OS, HA_CPU	PRR
CMD_LOAD_RANDOM_SEED	OS, HA_CPU	PRR
CMD_RESET	OS, HA_CPU	PRR
STAT_PRR_SUMMARY	PRR	OS, HA_CPU
STAT_RECONFIG_PRR	PRR	OS, HA_CPU

3.3.3.8. HA_CPU Interface Application (HCApp)

The application running on HA_CPU (HCApp) serves a number of purposes, as shown in Figure 25. Since the HA_CPU has been chosen as a basic reconfiguration controller, the HCApp implements the routines necessary to support it. In particular, it includes the drivers for Compact Flash and HWICAP IPs to read the bitstream and reconfigure PRR, respectively.

The HCApp also acts as a gateway between the two FSL channels, control and data used by HA, and a single channel used for communication via the mailbox IP. A round robin scheduler performs arbitration between the control/data FSL links and data received from the mailbox that is scheduled for processing. Functions defined in the pr_packet package of the *libpr* are used for deciding whether the packets should be forwarded to HA/PRR or LINUX_CPU or executed locally. As an example, the HA_CPU is responsible for the reconfiguration of PRRs, so the CMD_RECONFIG_PRR command is launched on the HA_CPU itself, while CMD_PRR_SUMMARY is a status summary request command that is forwarded to the PRR.

Finally, HCApp utilizes the mailbox driver and implements an interrupt handler. As a result, the mailbox side of HA_CPU buffers data as soon as it is received via the interrupt, and processes it later during the allocated round robin slot in the scheduler.

Also, the control and data packets received from the FSL links are processed based on polling during their allocated scheduling slots.

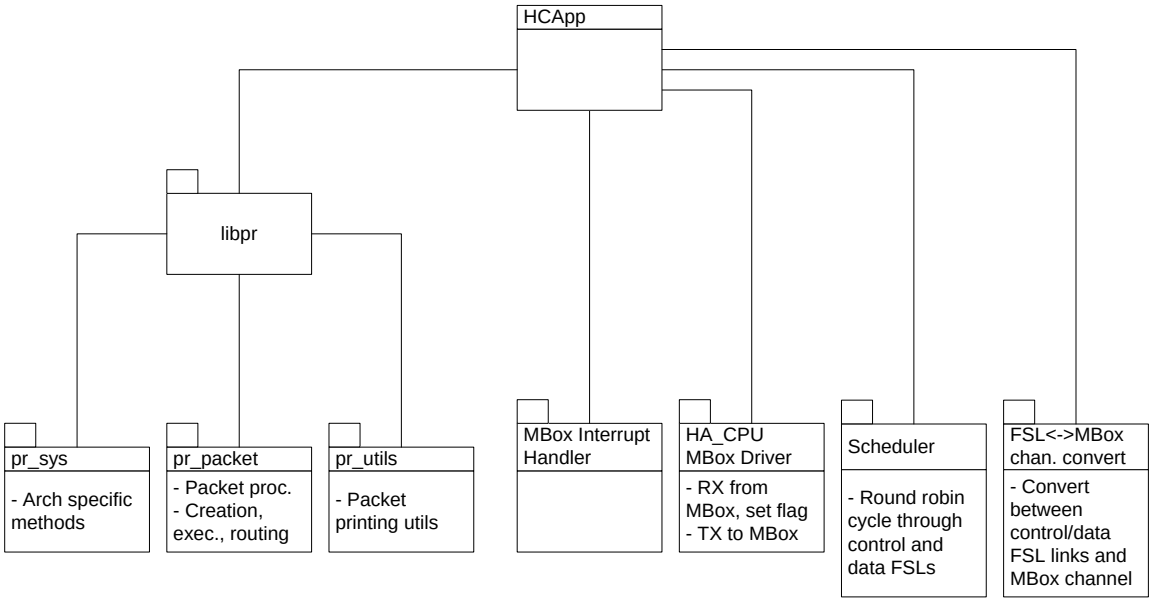


Figure 25: HA_CPU Application UML class diagram

As seen from the UML activity diagram in Figure 26 the HCApp operation is similar to that of the daemon process (see Figure 24). A scheduler loops through the available channels: control/data FSL and mailbox. When data is received by the mailbox, its driver buffers the message and processes on its next scheduling opportunity. If the destination has been reached, the packet is processed locally. Otherwise, it is forwarded to the appropriate interface based on the routing table and the command context.

The similar process is repeated for FSL channels, except without the interrupt handler. During their allocated scheduling slots, control and data FSL buffers are polled and read, if non-empty. The packets are again either processed locally or forwarded to their destination interface.

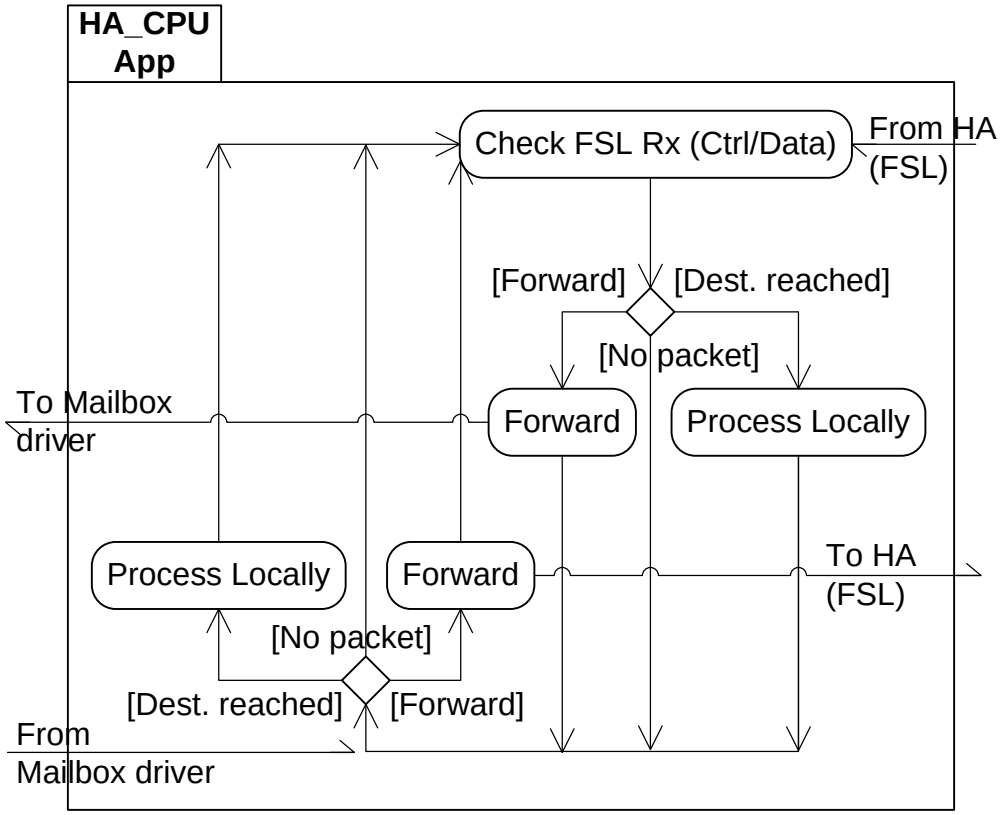


Figure 26: HA_CPU Application UML activity diagram

Table 6 shows a simple calendar used by the HCApp to process received packets from different interfaces. There are three slots with one slot per interface. If a status flag of the interface shows that there is nothing to process, the scheduler moves to the next slot.

Table 6: HCApp's calendar for receive packet scheduler (fair allocation)

Scheduled Interface
Slot 1: A: Check mailbox RX flag
Slot 2: B: Check control FSL flag
Slot 3: C: Check data FSL flag

The scheduling loop can be modified to increase the priority of a given interface by allocating it more slots in the calendar. For example, Table 6 above shows a fair allocation for all interfaces (i.e. ABC). In a scenario where the control FSL channel needs more attention, the calendar can be extended. For example, if the following priorities are established—50% for control, 30% for data and 20% for commands from

Linux—the calendar can increase to 10 slots with 5-3-2 occupation as shown in Table 7 (i.e., ABACABACAB).

Table 7: HCApp calendar for receive packet scheduler (50-30-20 allocation)

Scheduled Interface
Slot 1: A: Check mailbox RX flag
Slot 2: B: Check control FSL flag
Slot 3: A: Check mailbox RX flag
Slot 4: C: Check data FSL flag
Slot 5: A: Check mailbox RX flag
Slot 6: B: Check control FSL flag
Slot 7: A: Check mailbox RX flag
Slot 8: C: Check data FSL flag
Slot 9: A: Check mailbox RX flag
Slot 10: B: Check control FSL flag

3.3.3.9. Mailbox Driver (Linux)

There are two Xilinx mailbox drivers in the system. One resides in the Linux kernel space and addresses the mailbox from PLB0, where the LINUX_CPU is located. It will be described in this section. The other runs on HA_CPU and provides communication with the mailbox IP on the second PLB bus.

The Linux mailbox driver is a loadable kernel module launched on demand by a user when PRR access functionality is required. It allows a user application living in the user space of Linux, such as CLT, to communicate with the mailbox IP via the kernel. The driver has been developed in collaboration with C. Foucher from Université de Nice-Sophia Antipolis (UNS) and is maintained as a separate open source project [84].

A Netlink socket has been chosen as an inter-process communication (IPC) mechanism. It has a number of benefits over the other IPCs: system calls, ioctl or proc filesystems. One of the features important for this project is the support for dynamic loading. That is, there is no need to statically compile the driver with the kernel, which allows for embedding of the driver in a loadable kernel module. Another feature is the ability of the kernel to initiate communication with the user space without the need for the

user application to perform periodic polling. This feature is crucial to enable hardware-driven communication property of TERRAE without placing an additional burden on Linux. Other benefits of the Netlink are asynchronous communication, multicast support and a simple BSD socket-style API [85].

The resulting driver can be reused in a number of applications outside of the TERRAE. At the time of this research project, there was no equivalent driver on the Internet; however, some work on low-level mailbox access functionality via device nodes has been done by Foucher [84]. The decision has been made to merge both projects and create an open source version of the first Linux Xilinx mailbox driver that supports interrupts as well as both device node and Netlink communication with the kernel [84].

Figure 27 below shows an UML activity diagram displaying a two-directional communication between the user space and the mailbox via Netlink. A user space application such as DP encapsulates a TERRAE packet in a Netlink message and sends it via the kernel socket. In the background, the socket API queues the message in the driver's receive queue and invokes the reception handler. The handler in turn extracts the packet and sends it to the mailbox via a low-level driver call.

In the other direction, an interrupt is raised when a mailbox is non-empty. The driver's receive handler is split into two parts, which in Linux are historically called top and bottom halves. In TERRAE, the top half is meant to quickly copy the message from the mailbox into an internal buffer while Linux is in interrupt context. It then schedules a tasklet for buffer processing to be performed in the bottom half. When a tasklet is scheduled by Linux at a later point in time, it places the TERRAE packet inside a Netlink message and sends it to a user space application via the kernel socket (unicast send).

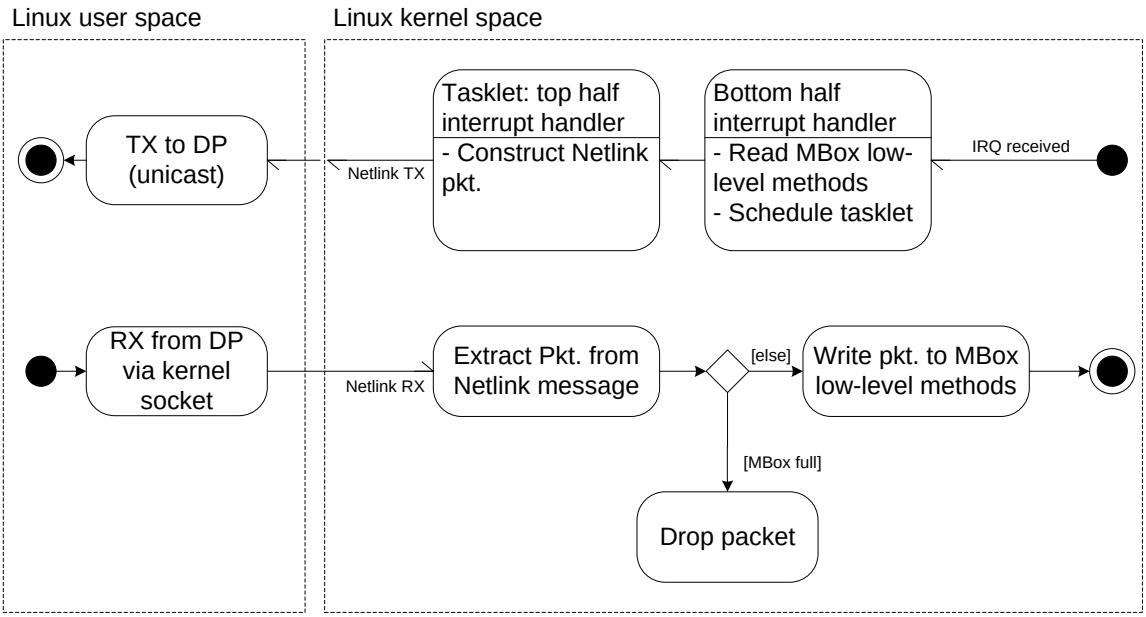


Figure 27: Mailbox driver (Linux) UML activity diagram

3.3.3.10. Mailbox Driver (Standalone on HA_CPU)

HA_CPU has its own version of the mailbox driver. The low-level driver routines were adopted from Xilinx mbox_v1_00_a driver and an interrupt handler has been created for this project. Its UML activity diagram is shown in Figure 28 below. When IRQ is triggered, initially all interrupts are disabled and the status bit of the mailbox is polled to ensure it is not empty. There is a dedicated receive buffer sized to the maximum capacity of the mailbox. It is allocated once at a startup and kept throughout the lifetime of the program in order to minimize allocating and clearing the buffer every time a new message is received. After the status confirms the mailbox has a message, the buffer is cleared and the mailbox is read up to the maximum size of the allowed message or until the mailbox runs empty. Then, a global pending_packet_from_os flag is set, indicating that a packet has been received and is ready to be processed. Finally, interrupts are enabled again.

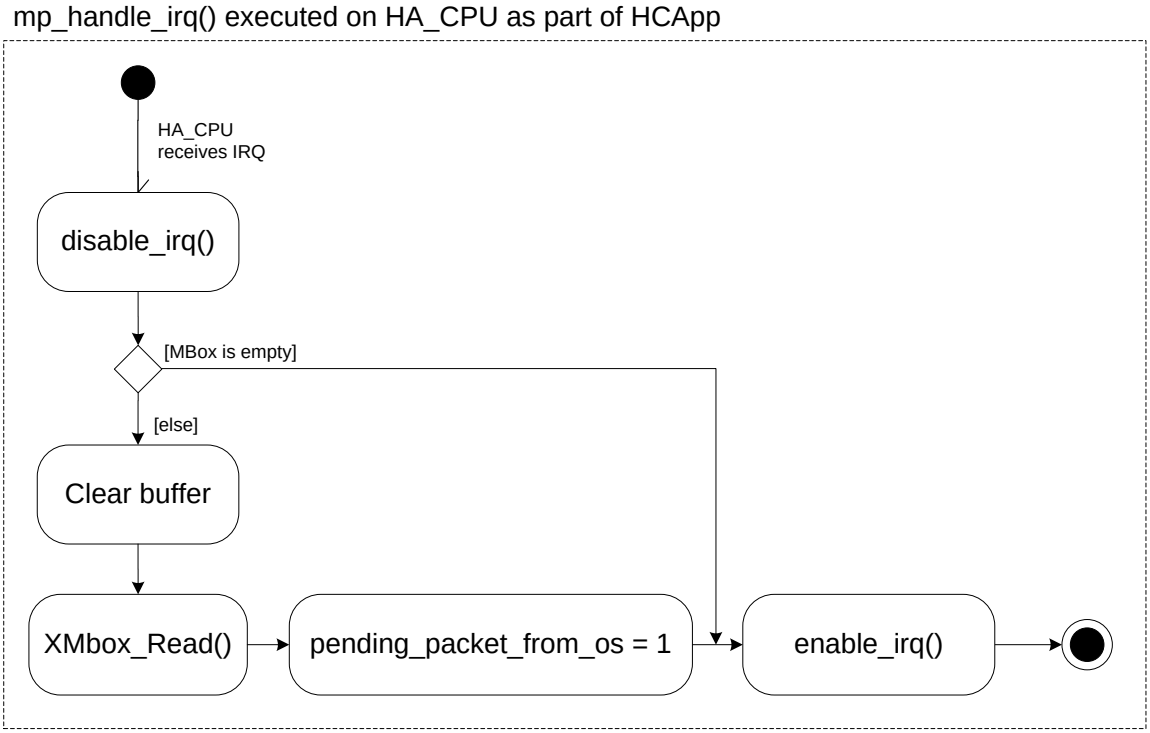


Figure 28: HA_CPU mailbox driver interrupt handler UML activity diagram

Writing to the mailbox is accomplished via a low-level Xilinx mailbox driver call XMbox_Write(). This call will be wrapped with an API and placed inside the platform specific part of the *libpr*, as will be explained in the next chapter.

3.3.3.11. System View: All Applications and Drivers

Figure 29 below shows a high-level UML activity diagram incorporating the CLT, DP, mailbox drivers and HCApp, and their partitioning across both processors.

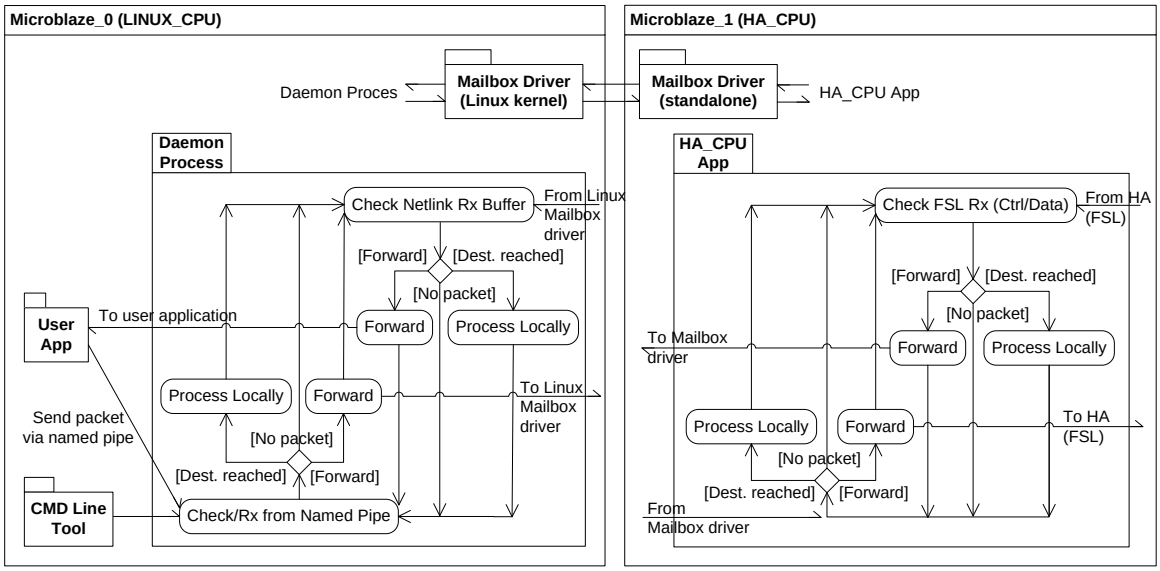


Figure 29: UML system activity diagram

Figure 30 represents a UML sequence diagram summarizing the messaging between applications with an example. First, a user requests a status (1). This is followed by sending a reconfiguration request via the CLT (2).

(1) The status request propagates to the appropriate PRR, which in turn replies with its status. The DP receives the status packet, updates its status table and prints the results.

(2) The second user request is reconfiguration. The packet reaches HCApp, which in turn reconfigures PRR1 and then probes its status by generating the same status requesting packet. As a result, PRR1 replies with a status packet containing the asserted “PRM is loaded” bit. The DP updates the table and prints it to the screen.

3.4. TERRAE Scripts

The framework has a set of scripts that aid in streamlining the development flow. These scripts include individual application makefiles, *libpr* customization utilities, scripts for appending romfs to the Linux kernel image, target board debug download tools, and scripts assisting the source code release process. These are described in more detail in Section 4.6.

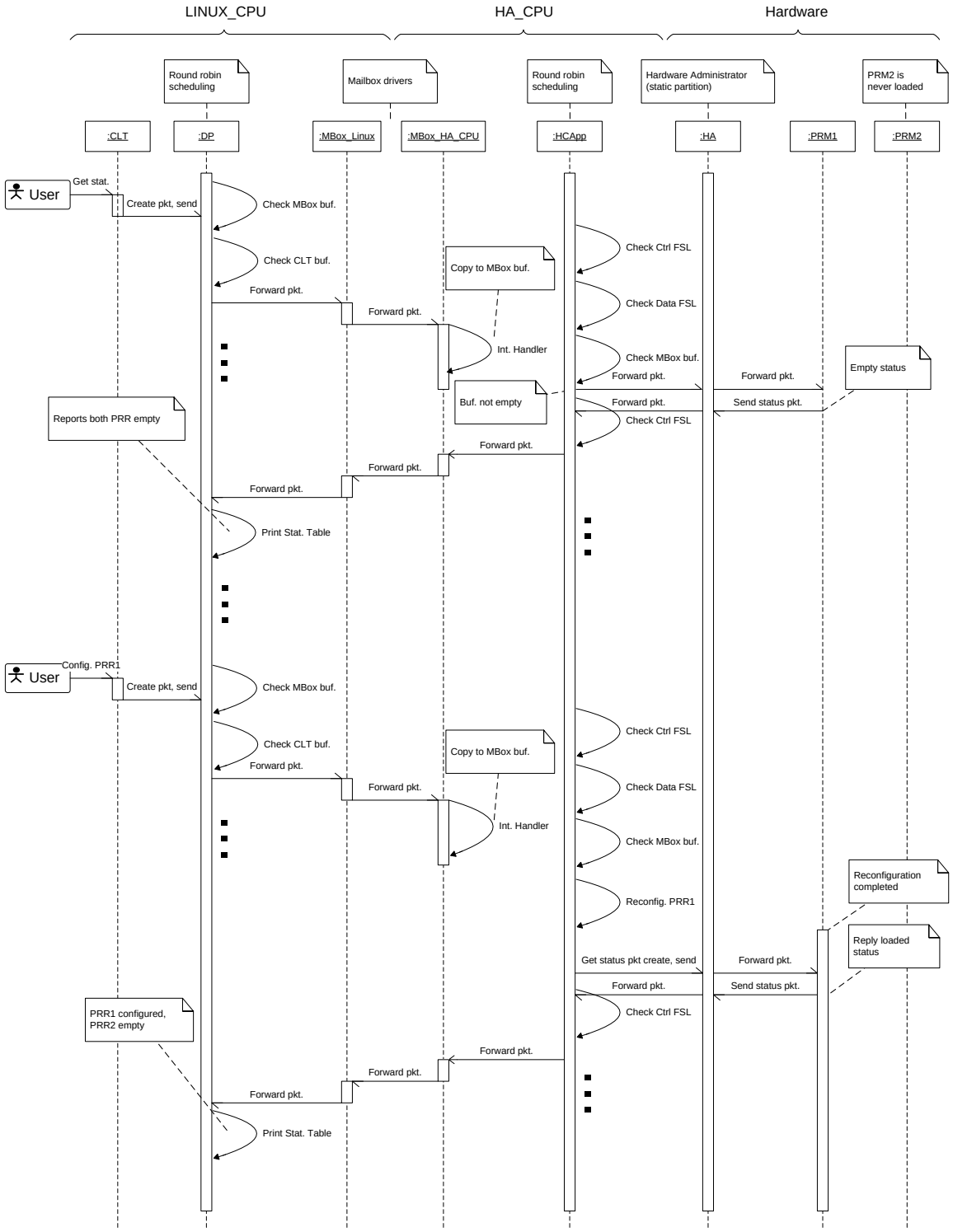


Figure 30: UML system sequence diagram

4. TERRAE Implementation

This chapter presents implementation details of TERRAE. It begins by reviewing the hardware and software architecture selected in the previous chapter. Then, it examines implementation details of the TERRAE hardware and modifications performed to the previous Intraframework implementation in order to support the combined adaptive hardware concurrent system. It continues with the PR library, choice of OS, packet structure, and individually developed applications and functions. Then, it describes the development flow and supporting scripts. Finally, it concludes with tool and other technical issues encountered during implementation, and summarizes workarounds.

4.1. TERRAE Architecture Review and Chapter Index

This previous chapter discussed the choice of hardware and software architecture, as summarized in Figure 16 and Figure 18, respectively. For convenience of the reader, they are listed again here in Figure 31 and Figure 32, with the relevant implementation chapter numbers added. In this chapter, the reader will find it useful to refer to the previous chapter describing high-level architecture of each component of the system.

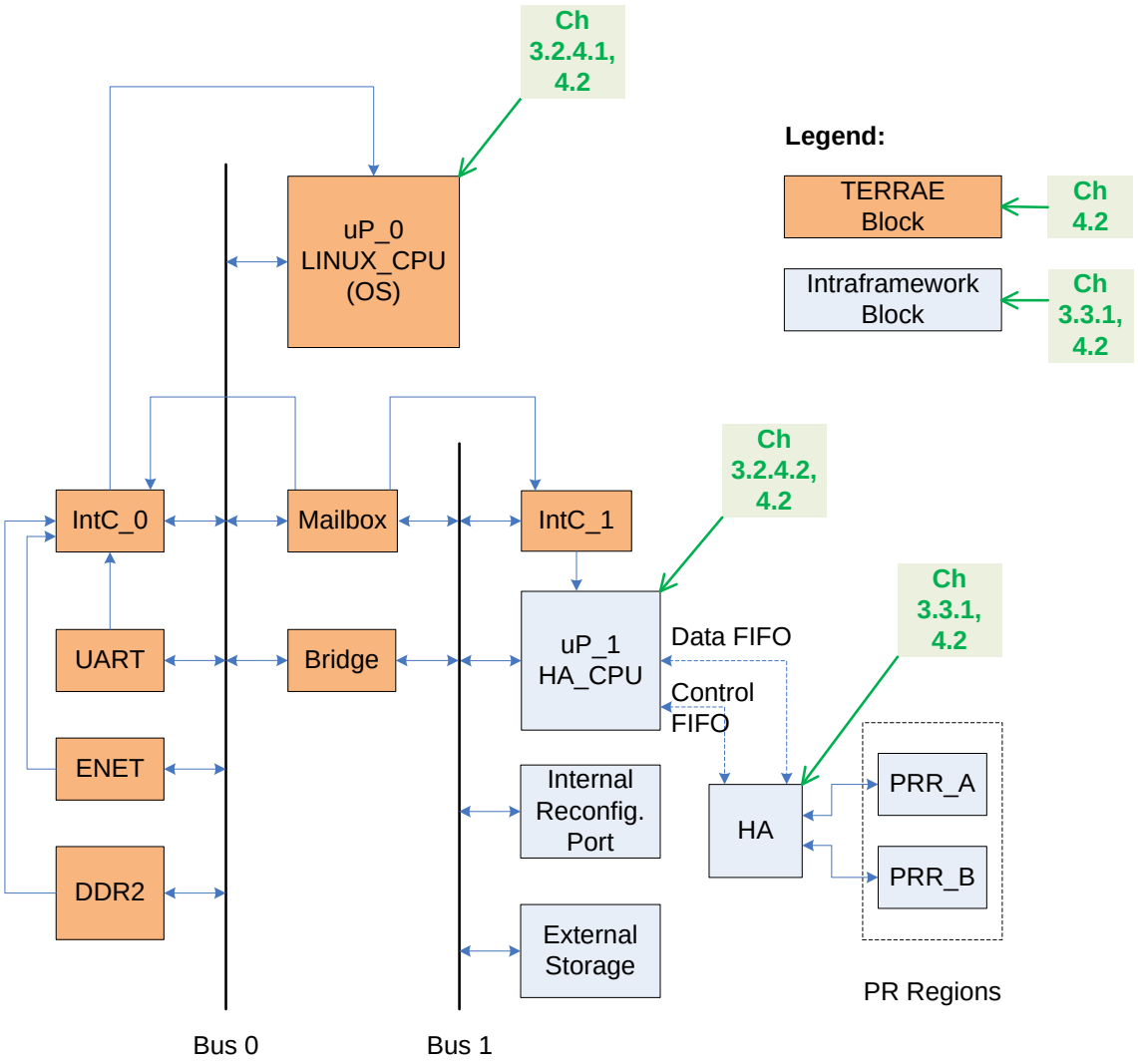


Figure 31: TERRAE hardware architecture and chapter index

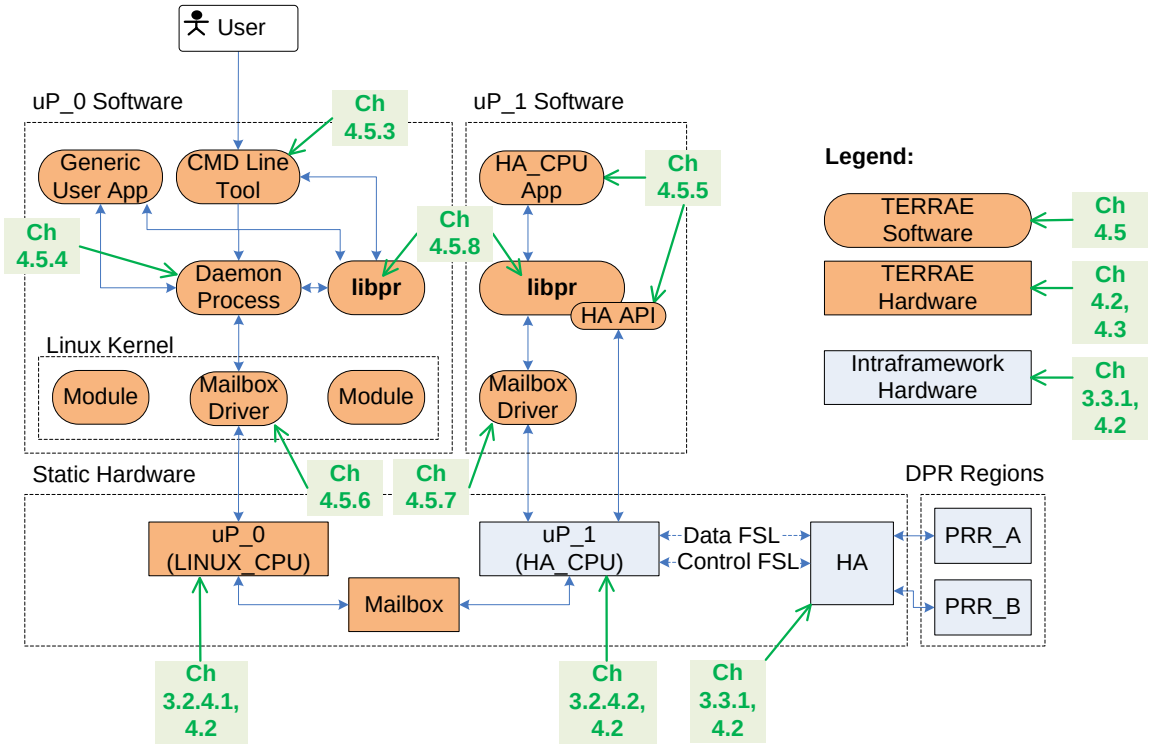


Figure 32: TERRAE software architecture and chapter index

4.2. TERRAE Hardware

The system was prototyped on the Xilinx ML505 development board, which has the Virtex-5 LX50T FPGA, as shown in Figure 33. This platform was chosen because the Intraframework project was built at the same time with TERRAE. Intraframework is a part of a Ph.D. dissertation [14] in the iDEA Laboratory of Simon Fraser University, which developed the DPR technology for agentizing hardware IP cores, and was used for interfacing with TERRAE. This decision enabled keeping both projects working largely with the same tools and minimized versioning difficulties, which typically become problematic during system integration. Even though there was an attempt to initially create a trial single bus Linux system with the Xilinx XUP V2Pro board that was also available, it later became evident that porting the project to ML505 had a number of benefits, including a larger real estate for debugging and demonstration of TERRAE.

The system was developed on a Ubuntu 10.04 [86] machine with Xilinx ISE 9.2.04, EDK 9.2.02 and PlanAhead 10.1 [87]. Even though Xilinx tools of version 10.x

were already available at the start of the project, only version 9.2.x supported DPR as part of the Xilinx Early Access [88]. Thus, the choice of IP versions was affected by the version supported by this tool.

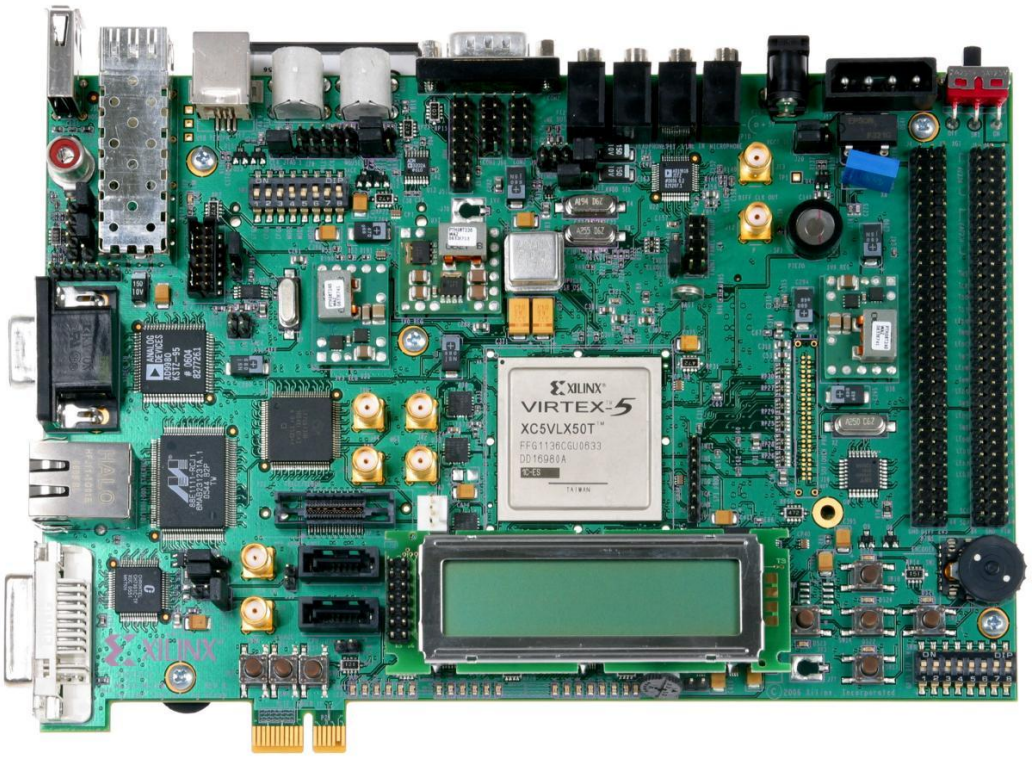


Figure 33: Xilinx ML505 board with Virtex-5 LX50T FPGA

Furthermore, the choice of Linux distribution also added additional constraints on the selection of IP versions. For example, ML505 was not available as a reference system for the EDK 9.2 toolset. Therefore, a reference Spartan-3 based design was ported to the ML505 board. However, the MicroBlaze version previously used for Intraframework was not compatible with the available Linux kernel port. The decision was made to downgrade the version of processor and other affected IPs that is selected by default in EDK 9.2.04. Both processors used in the system are soft MicroBlaze 7.00.a 32-bit RISC cores. Both run at 125 MHz to allow the fastest clock on this platform that is compatible with the DDR2 memory controller (MPMC 3.00.a). Figure 34 is a more detailed version of the system block diagram previously introduced in Figure 16 of Section 3.3.2. It shows Xilinx specific IP blocks. Versions and descriptions of the main IP blocks used in the system are summarised in Table 8 and Table 9 below.

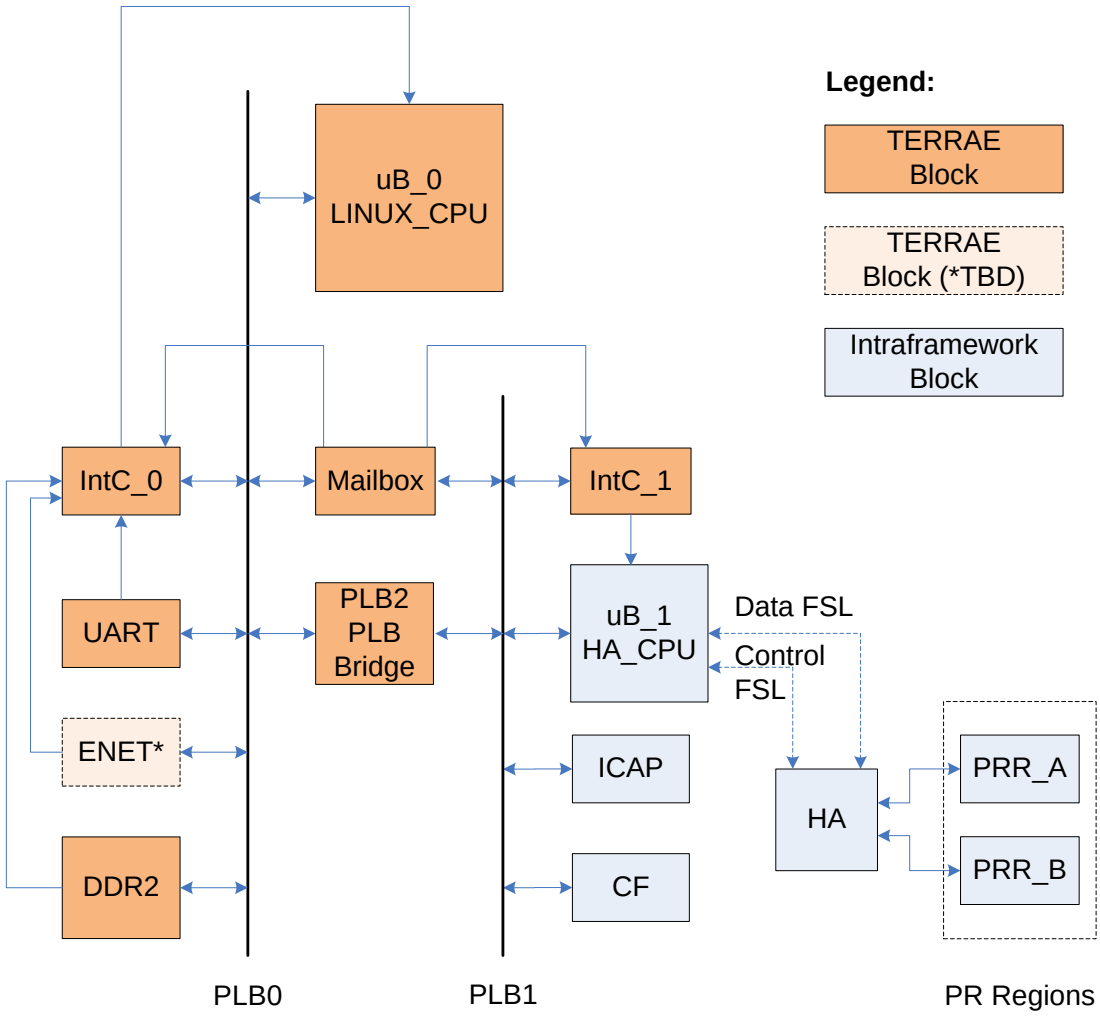


Figure 34: System block diagram (implementation with Xilinx FPGA)

Table 8: TERRAE IP descriptions

Block Name	IP Name	IP Version	Description
uB_0	Microblaze	7.00.a	MicroBlaze processor@125 MHz with instruction and data caching of DDR2.
Mailbox	Xps_mailbox	1.00.a	XPS Mailbox. Enables duplex FIFO based communication between two PLB busses with interrupt support.
PLB0	Plb_v46	1.00.a	Processor Local Bus for LINUX_CPU and its peripherals.

UART	Xps_uartlite	1.00.a	RS-232 serial communication port controller.
DDR2	Mpmc	3.00.a	Multi-port memory controller. Provides interface to external DDR2 memory and Xilinx CacheLink (XCL) to the processor (i.e. LINUX_CPU has two XCL lines for instruction and data directly connected to MPMC).
ENET	N/A	N/A	Ethernet controller enabling networking. Not implemented in the initial version of the framework.
IntC_0	xps_intc	1.00.a	Interrupt controller for Microblaze_0. Mailbox, UART and DDR2 have dedicated IRQ lines. In TERRAE it is used to notify LINUX_CPU of non-empty Mailbox's receive FIFO.
IntC_1	xps_intc	1.00.a	Interrupt controller for MicroBlaze_1. In TERRAE it is used to notify HA_CPU of non-empty Mailbox's receive FIFO.
PLB2PLB bridge	Plbv46_plbv46_bridge	1.00.a	Bridge between 2 PLB busses mainly for mapping UART to PLB1 so that both processors can use it as a stdout.

Table 9: Modified Intraframework IP

Block Name	**IP Name**	**IP Version**	**Description**
uB_1	Microblaze	7.00.a	MicroBlaze processor@125 MHz with instruction and data caching of DDR2.
PLB1	Plb_v46	1.00.a	Processor Local Bus for HA_CPU and reconfiguration subsystem.
HWICAP	Xps_hwicap	1.00.a	Hardware internal configuration access port. Enables partial dynamic reconfiguration of Xilinx FPGAs.
CF	Xps_sysace	1.00.a	XPS System ACE Interface Controller. Enables access to external System ACE CF storage.
HA	Fsl_control_hardware_manager	1.00.a	Hardware Administrator's control instance. Custom developed IP for Intraframework. Provides a duplex communication channel for control packets using two Fast Simplex Links (FSLs) as interface.
	Fsl_data_hardware_manager	1.00.a	Hardware Administrator's data instance. Custom developed IP for Intraframework. Provides a duplex communication channel for data packets using two Fast Simplex

			Links (FSLs) as interface.
PRM 1	PR_Region_A	1.00.a	Custom partial reconfiguration module 1 placed in region A
	PR_Region_B	1.00.a	Custom partial reconfiguration module 1 placed in region B
PRM 2	PR_Region_A2	1.00.a	Custom partial reconfiguration module 2 placed in region A
	PR_Region_B2	1.00.a	Custom partial reconfiguration module 2 placed in region B

Figure 35 is a screenshot of the Addresses tab of the EDK tool showing physical addresses of all peripherals used in the system. Figure 36 is a screenshot of the Bus Interfaces tab showing IP versions and their bus connections. Some IP blocks are not part of the framework. They either exist for historical reasons or were used during debugging. These blocks are FLASH, gpio_to_hm, gpio_from_hm, xps_timer_0, xps_timer_1, and math_0. The source code can be found on Source Forge [89].

Instance	Name	Base Address	High Address	Size	Bus Interface(s)	Bus Connection	IP Type	IP Version
dlmb_cntlr	C_BASEADDR	0x00000000	0x00001FFF	8K	SLMB	dlmb	lmb_bram_if_cntlr	2.10.a
ilmb_cntlr	C_BASEADDR	0x00000000	0x00001FFF	8K	SLMB	ilmb	lmb_bram_if_cntlr	2.10.a
dlmb_cntlr_1	C_BASEADDR	0x00000000	0x0000FFFF	64K	SLMB	dlmb_1	lmb_bram_if_cntlr	2.10.a
ilmb_cntlr_1	C_BASEADDR	0x00000000	0x0000FFFF	64K	SLMB	ilmb_1	lmb_bram_if_cntlr	2.10.a
dlmb_cntlr_2	C_BASEADDR	0x00010000	0x00017FFF	32K	SLMB	dlmb_1	lmb_bram_if_cntlr	2.10.a
ilmb_cntlr_2	C_BASEADDR	0x00010000	0x00017FFF	32K	SLMB	ilmb_1	lmb_bram_if_cntlr	2.10.a
DDR2_SDRAM	C_MPMC_BASEADDR	0x20000000	0x2FFFFFFF	256M	XCL0:XCL1:SPLB2		mpmc	3.00.a
math_0	C_BASEADDR	0x30000000	0x300000FF	256	SPLB	mb_plb	math	1.00.a
debug_module	C_BASEADDR	0x41400000	0x4140FFFF	64K	SPLB	mb_plb_1	mdm	1.00.a
xps_sysace_0	C_BASEADDR	0x50000000	0x5000FFFF	64K	SPLB	mb_plb_1	xps_sysace	1.00.a
xps_hwicap_0	C_BASEADDR	0x60000000	0x6000FFFF	64K	SPLB	mb_plb_1	xps_hwicap	1.00.a
RS232_DTE	C_BASEADDR	0x71000000	0x7100FFFF	64K	SPLB	mb_plb	xps_uartlite	1.00.a
mb_plb_plb_bridge_1	C_RNG0_BASEADDR	0x71000000	0x7100FFFF	64K	SPLB	mb_plb_1	plbv46_plbv46_bridge	1.00.a
xps_mailbox_0	C_SPLB0_BASEADDR	0x71010000	0x710100FF	256	SPLB0	mb_plb	xps_mailbox	1.00.a
xps_intc_0	C_BASEADDR	0x71100000	0x7110FFFF	64K	SPLB	mb_plb	xps_intc	1.00.a
xps_timer_0	C_BASEADDR	0x71C00000	0x71C000FF	256	SPLB	mb_plb	xps_timer	1.00.a
xps_mailbox_0	C_SPLB1_BASEADDR	0x81010000	0x810100FF	256	SPLB1	mb_plb_1	xps_mailbox	1.00.a
gpio_to_hm	C_BASEADDR	0x81020000	0x8102FFFF	64K	SPLB	mb_plb_1	xps_gpio	1.00.a
gpio_from_hm	C_BASEADDR	0x81030000	0x8103FFFF	64K	SPLB	mb_plb_1	xps_gpio	1.00.a
xps_intc_1	C_BASEADDR	0x81200000	0x8120FFFF	64K	SPLB	mb_plb_1	xps_intc	1.00.a
xps_timer_1	C_BASEADDR	0x81C00000	0x81C000FF	256	SPLB	mb_plb_1	xps_timer	1.00.a
DDR2_SDRAM	C_SDMA_CTRL_BASEADDR	0x84600000	0x8460FFFF	64K	SDMA_CTRL3	mb_plb	mpmc	3.00.a
FLASH	C_MEM0_BASEADDR	0x89000000	0x89ffffff	16M	SPLB	mb_plb	xps_mch_emc	1.00.a
mb_plb_plb_bridge_1	C_BRIDGE_BASEADDR	0x90000100	0x900001FF	256	SPLB	mb_plb_1	plbv46_plbv46_bridge	1.00.a

Block Diagram1 | System Assembly View

Figure 35: TERRAE physical addresses

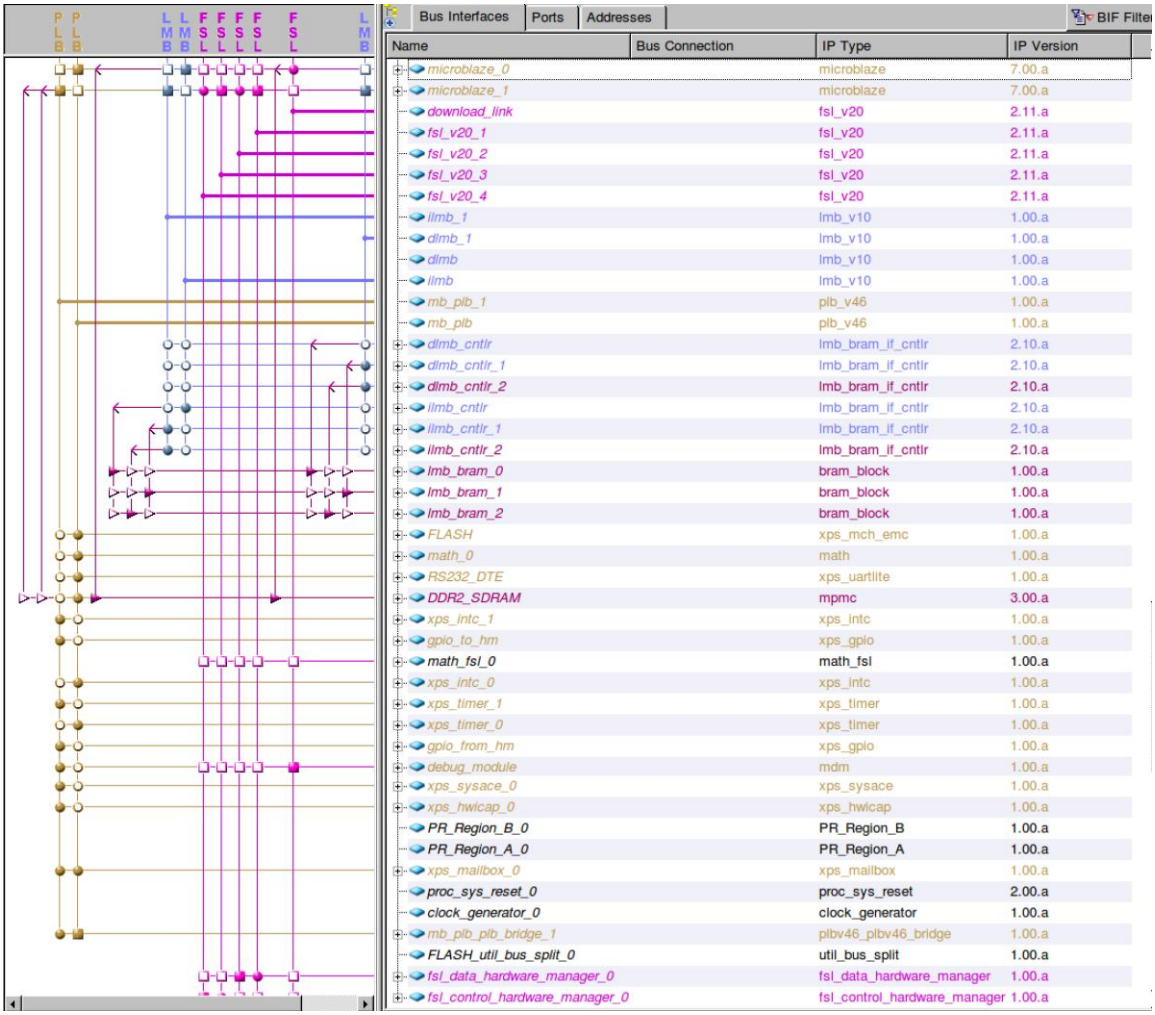


Figure 36: TERRAE IP versions

4.3. DPR-enabled TERRAE

In order to add DPR functionality to the EDK project described in Section 4.2, the Xilinx Early Access PR flow [88] is followed. A more detailed process is described in Section 4.6.1. In summary, the EDK project is wrapped with a top level shell RTL module. The digital clock manager (DCM) is moved to the top level. The clocks used by the system had to be generated with the help of two DCM_ADV blocks [90] chained together to produce the required frequencies and phases, as seen in Figure 37, Table 10 and also described in Section 4.7.4. PRMs were designed and synthesized in the ISE tool, and their netlists associated with PRRs in PlanAhead. The final synthesis, mapping, place, route and bitstream generation were performed in PlanAhead.

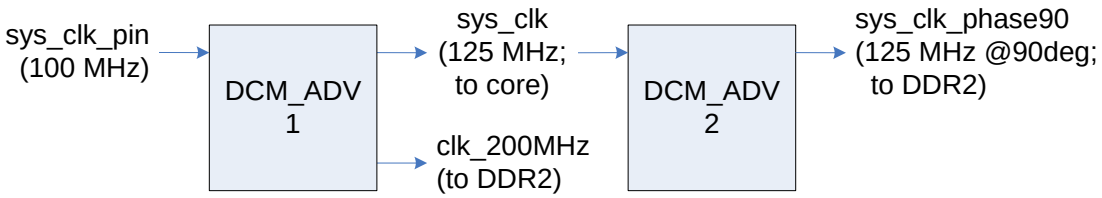


Figure 37: System clocks

Table 10: System clocks

File	Description
sys_clk_pin	100 MHz ML505 on-board oscillator clock connected to FPGA's input pin
sys_clk	125 MHz generated clock for the core FPGA logic
clk_200MHz	200 MHz generated clock for DDR2 RAM
sys_clk_phase90	125 MHz generated clock with 90 degrees offset for DDR2 RAM

4.4. Partial Reconfiguration Modules

For demonstration and validation purposes, a pair of modules were created, which allow for distinguished functionality, specifically an incrementor (I-PRM) and decrementor (D-PRM). As their names suggest, the only difference between the modules is their arithmetic operation. The modules share the following functionality. During a setup phase, the initial and the final values for the counters are loaded into internal registers, Parameter_A and Parameter_B, respectively. Then, after receiving the CMD_START_INCREMENT or CMD_START_DECREMENT command, the corresponding module begins its operation. The value of the free running counter can be read via the data interface. The counter can be stopped with the TERRAE's CMD_STOP command. I-PRM's state machine is captured in Figure 38. This figure uses Intraframework's command naming terminology [13]-[14], for example, STOP_COMMAND instead of the TERRAE's CMD_STOP, and also has some redundant states.

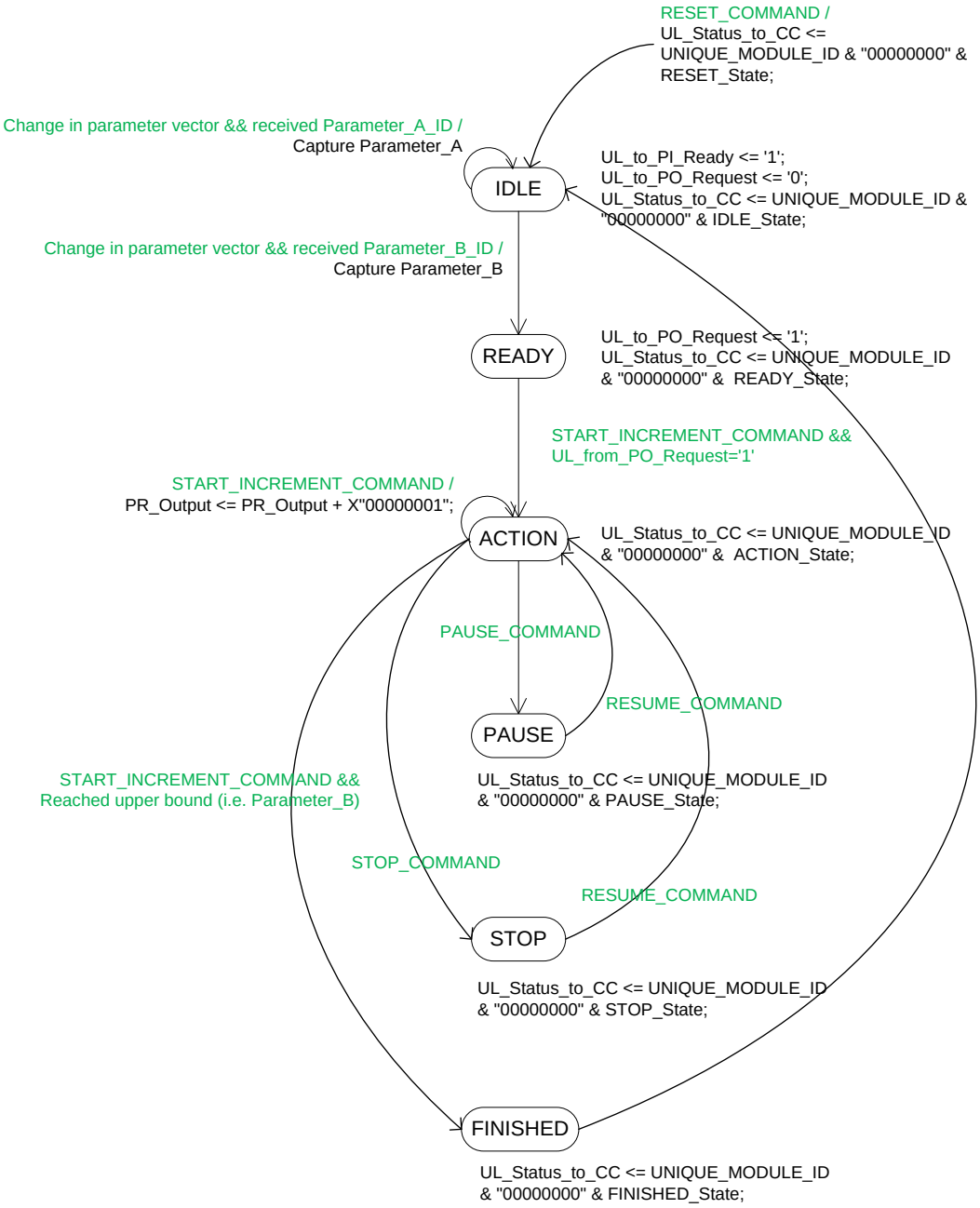


Figure 38: Incrementor PRM (I-PRM) state machine

Both PRMs can be configured into PR region A (PRR_A) and/or PR region B (PRR_B). Due to current limitations of the tools, separate bitstreams need to be synthesized for each PRR. For example, in the case of m PRMs and n possible PRR locations, a total of ℓ bitstreams is required, where $\ell = m \times n$. Following the example, if I-PRM is currently configured in PRR_A and desired to be relocated to PRR_B, the I-

PRM_B version of the bitstream needs to be loaded and written to PRM_B, while the I-PRM_A version of the bitstream can be replaced with a blank bitstream or simply disabled to save time. Depending on the frequency of reconfiguration, both dynamic and static power can be saved by overwriting with an empty bitstream [91].

4.5. TERRAE Software PR Library, Applications and Drivers

TERRAE includes two versions of the mailbox driver, for Linux and standalone targets, and three main agent applications, CLT, DP, and HCApp. The code shared between the applications is packaged in PR Library (*libpr*) introduced in Section 3.3.3.1. Since the applications running on LINUX_CPU and HA_CPU communicate with specific hardware interfaces, #ifdef directive is used with LINUX_C and STANDALONE_C constants in order to reuse the same API while selecting the appropriate functionality at compile time. These target processor-dependent functions reside in pr_sys.c files (i.e., part of *libpr*). A set of build tools is used for sourcing the correct pr_sys.c before compilation. These are described in Section 4.6.

Figure 39 is a more detailed version of Figure 18 of Section 3.3.3, showing implementation aspects of hardware and software including Xilinx specific terminology for the processor, bus and other IP.

Figure 39 also shows a small deviation from the previously defined architecture in the way HCApp communicates with the mailbox driver. For implementation simplicity, a shortcut was taken allowing HCApp to contain an interrupt handler rather than abstracting it away via the *libpr* API. Thus, the diagram shows that HCApp is able to communicate with the driver both directly in the receive direction and also via the *libpr* during transmission.

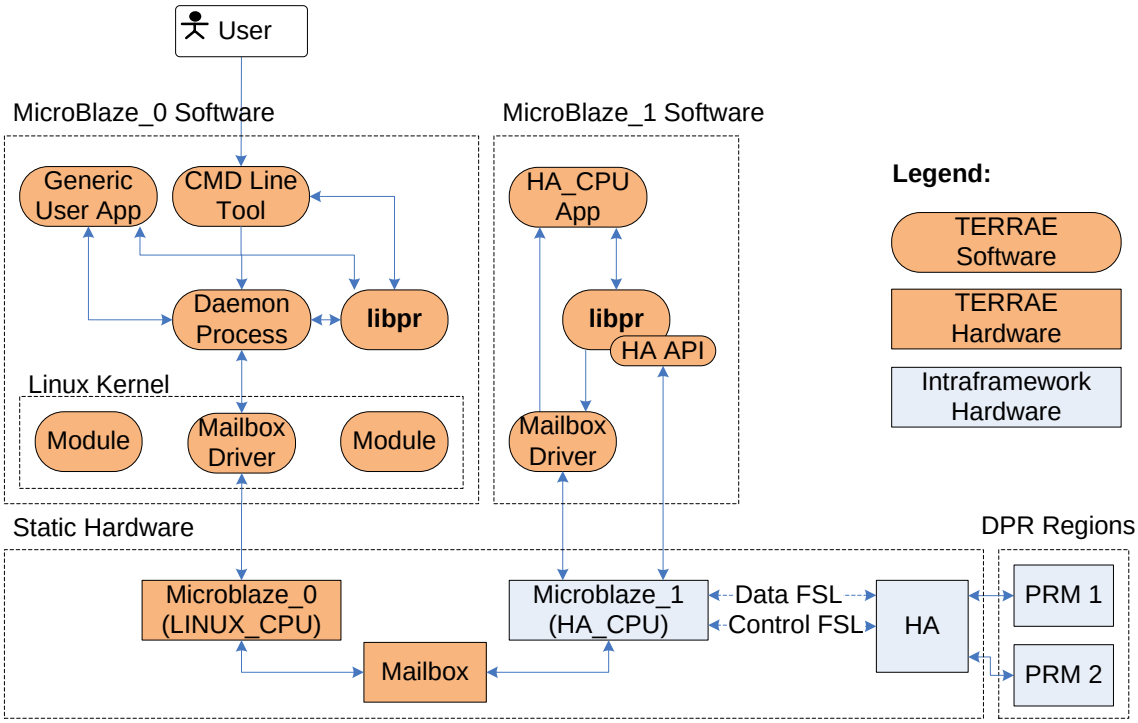


Figure 39: Implementation of software connectivity in TERRAE

4.5.1. Linux Kernel Configuration

After the hardware system has been implemented, it was necessary to customize the kernel. For assistance in this process the PetaLinux package is used, as illustrated in Figure 40. It shows step-by-step instructions along with screenshots from user guide [92], which addresses more salient points. The major benefit of using the PetaLinux toolchain for this research is automatic BSP generation based on hardware description files extracted from Xilinx EDK project directories, as well as tool assistance during kernel configuration.

After defining a new hardware platform and selecting it in *menuconfig*, hardware description files from the EDK project directory are populated in the PetaLinux environment with the aid of the *petalinux-copy-autoconfig* utility. Next, the *make menuconfig* is run again and kernel settings are customized:

- Deselect “Preemptible Kernel”;
- Deselect EMAC support. Ethernet is not supported in the first version of this project;

- Select Xilinx uartlite serial port support;
- Select support for console on Xilinx uartlite port;
- Deselect u-boot. Instead the u-boot, the XMD debugger is used for downloading executables.

Finally, the kernel is built with the help of *yes* utility [92].

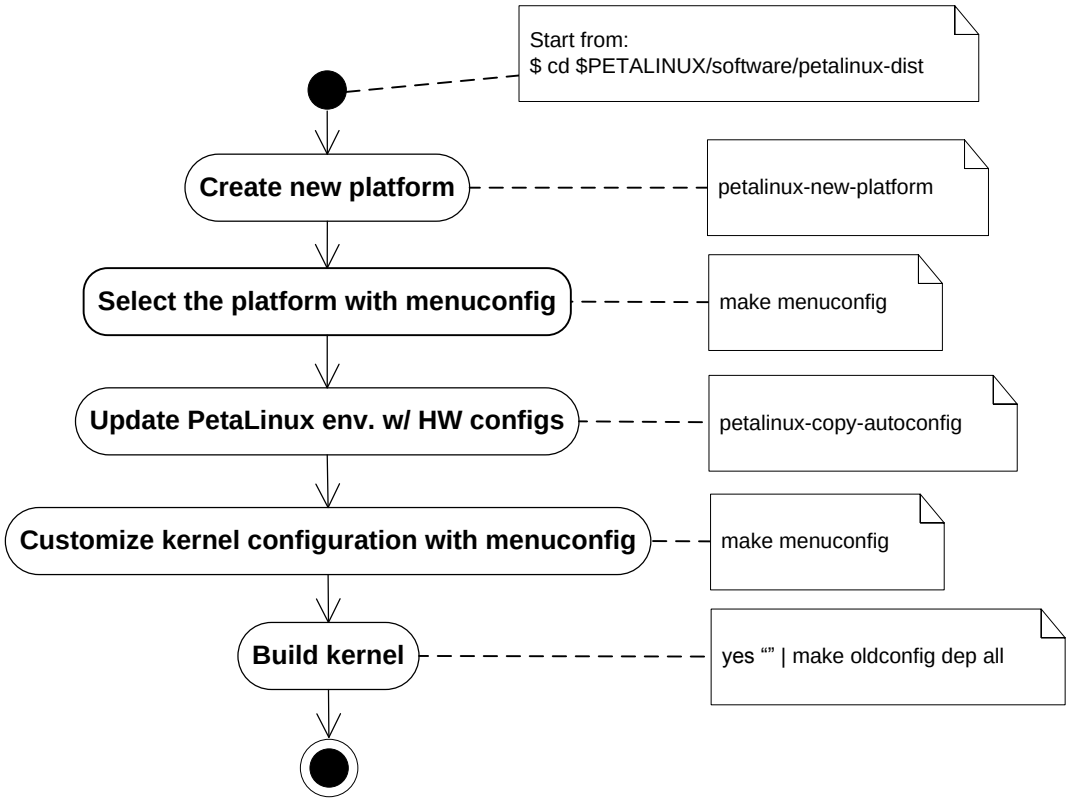


Figure 40: Building PetaLinux kernel UML activity diagram

4.5.2. Packet Structure

A packet-oriented networking communication was developed for hardware and software agents residing in Linux, HA_CPU and PRMs. This packeting system implements the TERRAE structure for exchange of messages between OS, HA_CPU, HA and PRMs. Each packet contains at least one 32-bit word. The first word of any packet has a header. Depending on packet class and type, the bit fields of the header are uniquely defined.

Packet class bit distinguishes between control and data packets (Figure 41 and Figure 42). This bit was specifically added on top of the Intraframework packet structure because the OS receives a single stream of packets that need to be further classified. Based on this bit, HA_CPU sends packets to either control or data FSL.

The packet type bit field specifies whether a packet is a single 32-bit word with the upper 16 bits used as a parameter, status or data, or whether it is a burst packet where the upper bits represent the number of subsequent packets to follow.

Control packets are further decomposed into command and status. Currently, the applications originate commands from Linux and receive status back. However, the packet forwarding subsystem supports command packets originated from PRMs and other HA instances as long as commands are added into the internal forwarding table. In the example of Figure 41, the status packet for PRM carries the following fields in its upper 16 bits: a unique 4-bit PRM ID; 4-bit state information; and the PRR empty/loaded status bit. Other bits are reserved for future expansion. Figure 30 shows an example UML sequence diagram with the exchange of command and status packets between agents.

The command ID is stored in the lower 7 bits allowing for 2^7 unique commands and the same number of status messages. Destination HA is used by HA_CPU to route the packet to the appropriate HA and destination PRR is used by HA to forward to the appropriate PRR.

Data packets are slightly different from control. Instead of the command ID, they have source HA and PRR. This provides the OS with information about the origin of data.

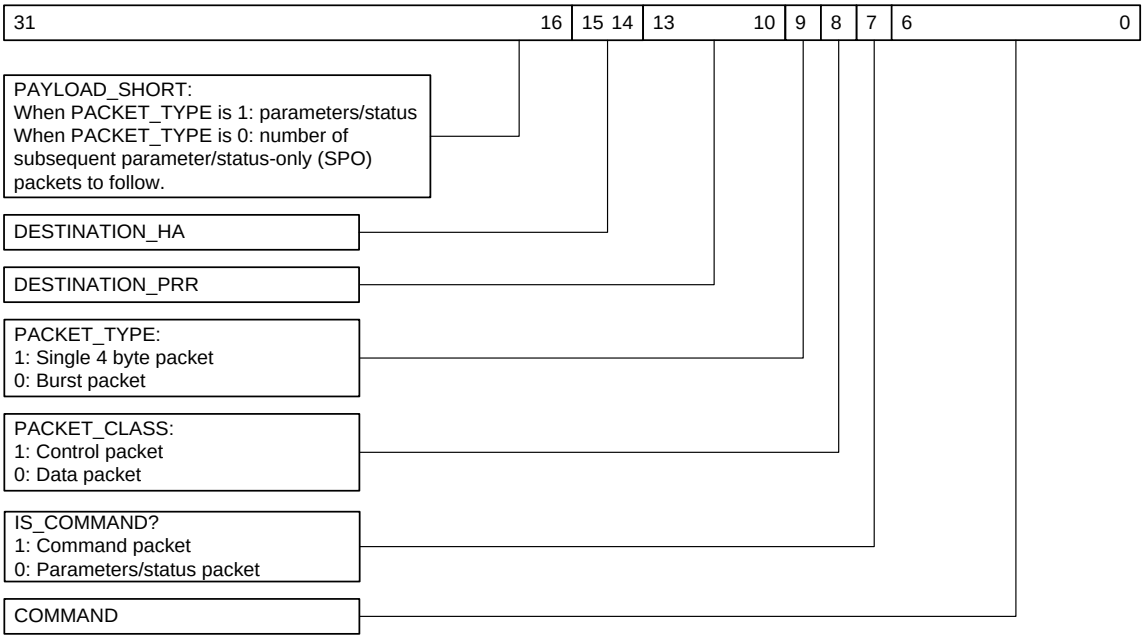


Figure 41: Control packet structure

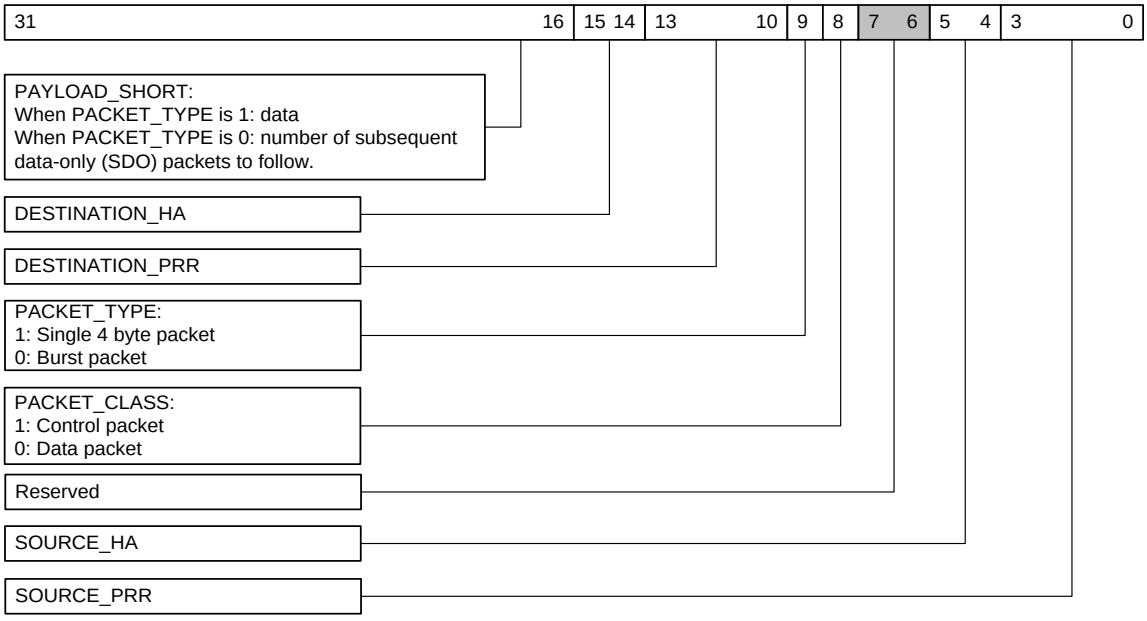


Figure 42: Data packet structure

4.5.3. Command Line Tool (CLT) Implementation

As was introduced in the UML activity diagram of Figure 22, the CLT has two main steps. First, it reads the command line arguments by executing the

process_cmd_line_args() function. Second, it executes a corresponding action command. Even though Figure 22 introduces four separate actions, only two were implemented (ARGV_CTL_RECONFIG and ARGV_CTL_ID), as seen from the code snippet below.

```
// Execute command
switch (action) {
  case ARGV_CTL:
    if (VERBOSE) {printf("case ARGV_CTL\n");}
    break;
  case ARGV_CTL_ID:
    // Note: -ctlid id does not check whether -dest has been entered
        printf("Sending command %d to (HA, PRR) (%d, %d)\n",
               command, dest_ha, dest_prr);
        send_ctl_packet(command, dest_ha, dest_prr, AGENT_ID);
    break;
  case ARGV_CTL_RECONFIG:
        printf("Reconfigure (HA, PRR) (%d, %d) with %s\n",
               dest_ha, dest_prr, bitfile);
        send_cmd_prr_reconfig(AGENT_ID, bitfile, dest_ha, dest_prr);
        break;
  default:
        printf("%s: No recognized action\n", __FUNCTION__);
    break;
}
```

The reconfiguration command line argument ARGV_CTL_RECONFIG creates a control packet with fields set to the values summarized in Table 11. An example reconfiguration control command packet to HA#0, PRR#1 can be sent by executing the following command:

```
# /bin/prctl -bit inca.bit -dest 0 1
```

Table 11: CMD_RECONFIG_PRR control packet fields

Header Field	Value
COMMAND	CMD_RECONFIG_PRR
IS_COMMAND	1: PKT_CTL_CMD (command packet)
PACKET_CLASS	1: PKT_CLASS_CTL (control packet)
PACKET_TYPE	0: PKT_TYPE_BURST
DESTINATION_PRR	dest_prr
DESTINATION_HA	dest_ha
PAYLOAD_SHORT	strlen(bitfilename)

The second recognized command line argument ARGV_CTL_ID allows for creation of a general single word control packet with COMMAND field set to the specified ID value. Its header word has field values summarized in Table 12. The following command can be executed to send CMD_PRR_SUMMARY status summary request packet (i.e. CTLID is 1 according to Table 2) to HA#0, PRR#0:

```
# /bin/prctl -ctlid 1 -dest 0 0
```

Table 12: General control packet fields

Header Field	Value
COMMAND	One of CTLIDs from Table 2 including CMD_RECONFIG_PRR
IS_COMMAND	1: PKT_CTL_CMD (command packet)
PACKET_CLASS	1: PKT_CLASS_CTL (control packet)
PACKET_TYPE	1: single 4 byte packet
DESTINATION_PRR	dest_prr
DESTINATION_HA	dest_ha
PAYLOAD_SHORT	16-bit short command parameter

4.5.4. Daemon Process (DP) Implementation

The DP implementation is visualized with a detailed UML activity diagram in Figure 43. The DP can be loaded via the following command line from the operating system terminal:

```
# /bin/pr_config &
```

As a first step, the DP initializes the PRR status table with N records corresponding to the number of PRRs in the system. Currently, the values are predefined in the pr_status.c file of *libpr*. The DP also initializes a named pipe for communication with the CLT application. Then, an infinite round robin loop begins, that checks for packets received from the CLT and mailbox. The current implementation equivalently distributes its attention between the two interfaces, such that it checks for CLT, then mailbox, and repeats.

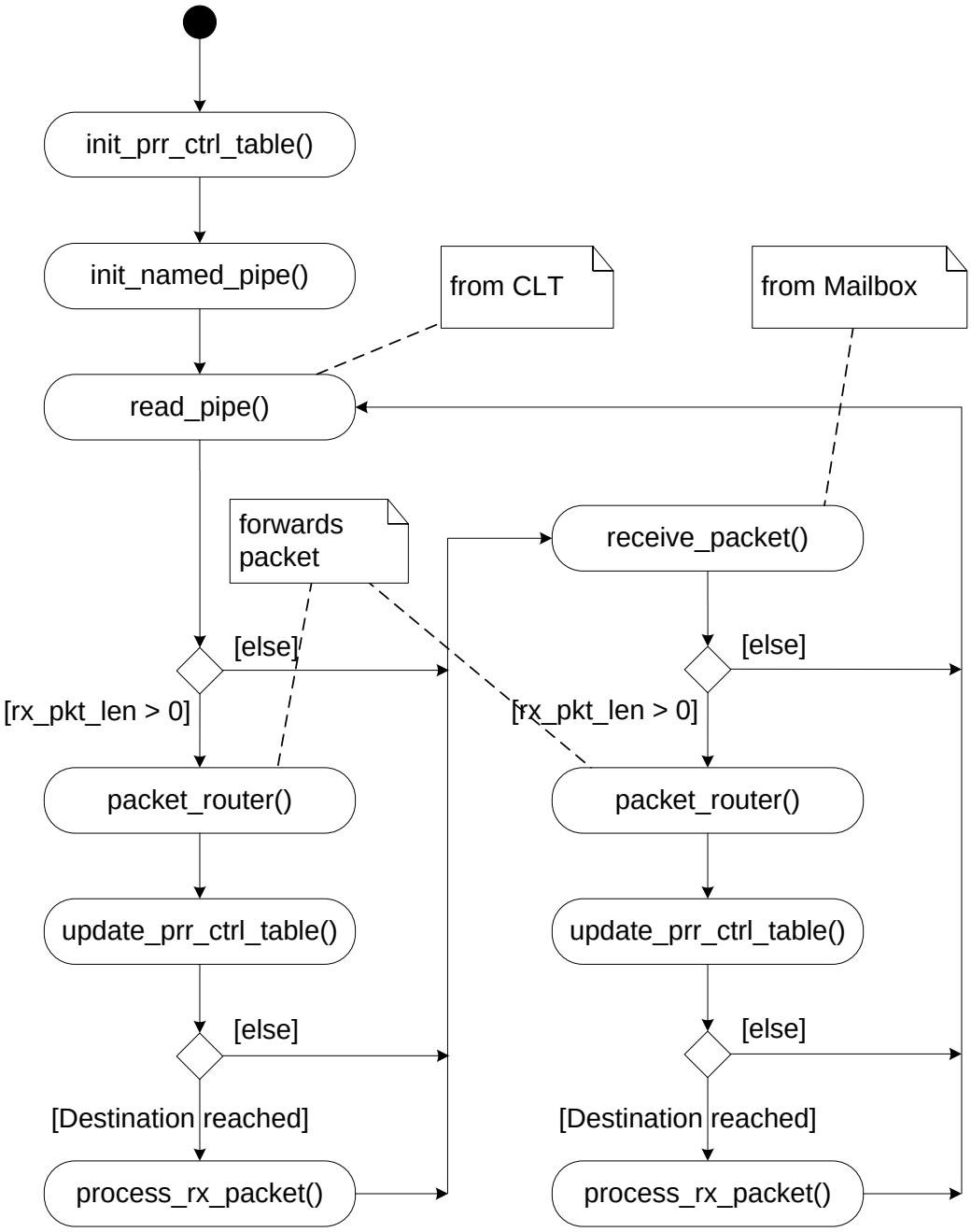


Figure 43: Daemon Process detailed UML activity diagram

While in the loop, the DP checks for a packet received from the CLT by reading the named pipe. If a non-zero packet is received, it is processed by a packet_router() function, and, if the packet is destined to another agent, it forwards the packet in the appropriate direction. The forwarding direction is derived from the information extracted from packet header, such as packet class and command type. The current

implementation of the DP supports forwarding towards the mailbox only; however, it can be expanded to enable forwarding to user applications or agents residing in external systems via the mailbox or other interfaces. The packet_router() function is described in more detail in Section 4.5.8.1.

As a next step, the PRR status table is optionally updated. Then, depending on whether the status of is_dest_reached flag set by the packet_router() indicates that the packet has reached its destination, it is processed locally by executing the process_rx_packet() (see Section 4.5.8.1.3).

4.5.5. HA_CPU Application (HCApp) Implementation

The HCApp runs with the HA_CPU startup. Figure 44 shows its implementation in a UML activity diagram. This embedded application represents a round robin scheduler that sequentially checks pending packets from the OS and both HA's FSL interfaces, control and data. Before getting into the scheduler loop, it initializes the MicroBlaze instruction and data cache with init_hw(), configures mailbox IP with XMbox_LookupConfig(XPAR_XPS_MAILBOX_0_IF_1_DEVICE_ID), registers the IRQ handler with irq_register_handler(mp_handle_irq) and enables bus macros with a simple 32-bit word write to the corresponding GPIO. Bus macros are disabled before reconfiguring a PRR and enabled before accessing it via FSL.

```
XIo_Out32(get_prr_bm_gpio_addr(1), 0x00000001);
XIo_Out32(get_prr_bm_gpio_addr(2), 0x00000001);
```

The scheduler loop can be coded with a user defined calendar. In this example, it is a fair scheduler that is allocating one slot for each interface. First, it disables interrupts and starts by checking whether there is a pending packet from OS by referring to the pending_packet_from_os flag set by mp_handle_irq() as seen from the code snippet below. If there is an unprocessed non-zero packet in the mailbox receive buffer, the HCApp calls packet_router() function of the *libpr* that based on the packet's header determines whether it has reached destination or should be forwarded. If the HCApp is the destination, the packet is passed to process_rx_packet() function of the *libpr* for processing. Then, the pending_packet_from_os flag is reset and interrupts are enabled.

At this time, any new packets received by the mailbox will be copied to the receive buffer by the interrupt handler.

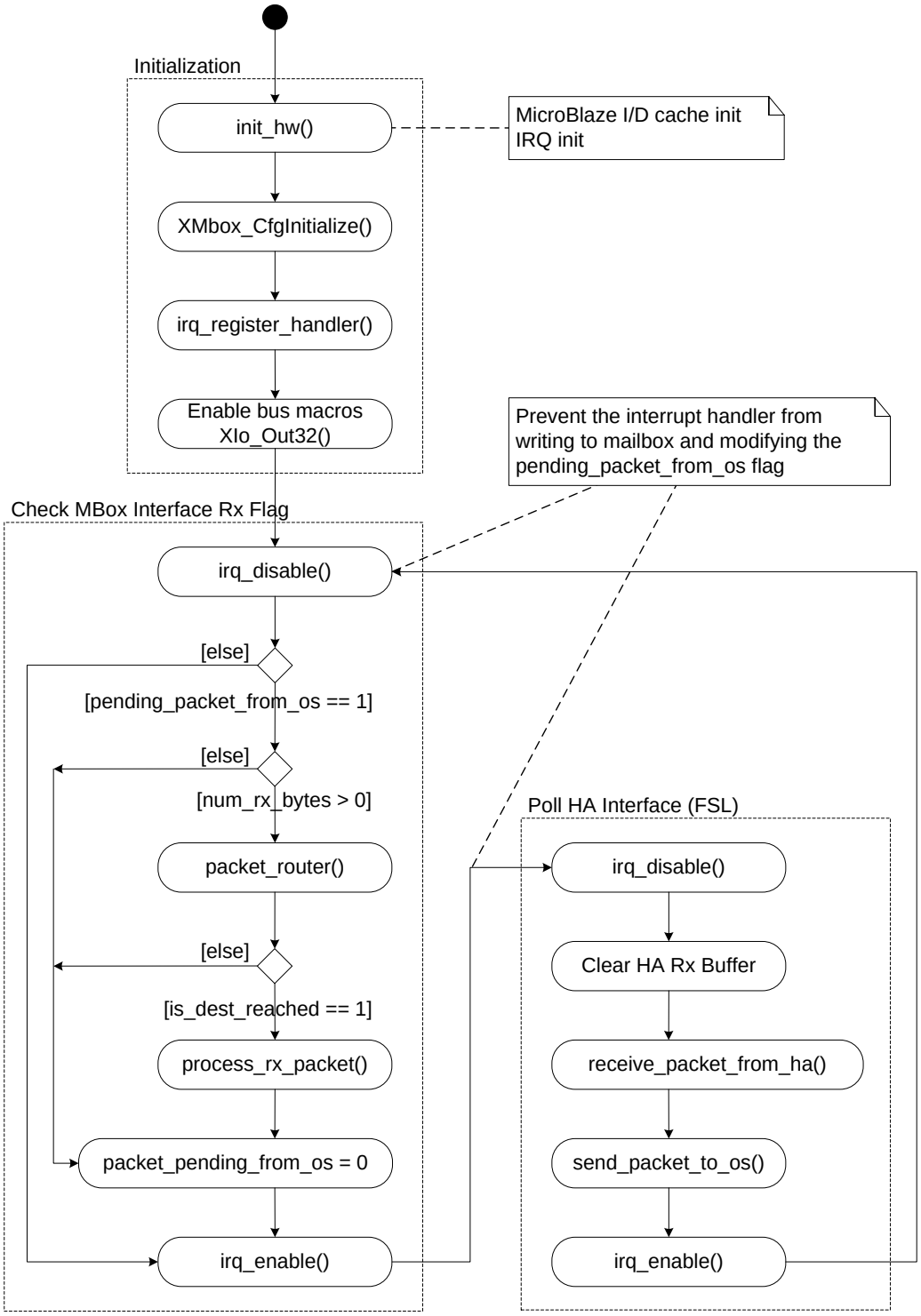


Figure 44: HA_CPU Application detailed UML activity diagram

```
if (pending_packet_from_os) {
      if (num_rx_bytes > 0) {
            // Forward packet if no local action is required
            packet_router((char *)rx_buffer,  // treat u32* as char*
                                    num_rx_bytes,
                                    AGENT_ID,
                                   &is_dest_reached,
                                    get_mailbox_handle);

            // Process packet if the current node is its destination
            if (is_dest_reached) {
                  if (VERBOSE) {printf("Process packet locally\r\n");}
                  process_rx_packet((char *)rx_buffer,
get_mailbox_handle);
            }
      } else { // !if (num_rx_bytes > 0)
          printf("%s: HA_CPU: refusing to process zero length
packet\r\n", __FUNCTION__);
      }
      pending_packet_from_os = 0;
} else {
      if (VERBOSE) {printf("%s: HA_CPU: no packet from OS\r\n",
__FUNCTION__);}
}
```

Next, the scheduler checks the control FSL interface. It disables the interrupts, clears the HA receive buffer and calls receive_packet_from_ha() with a control FSL flag PKT_CLASS_CTL. The latter function checks the control FSL interface for a packet and copies it to the buffer as seen in the code snippet below.

```
int receive_packet_from_ha(char *ha_rx_buffer,
                           int *num_ha_rx_bytes,
                           int class) {
      int type;
      int fsl_error = 0;
      int fsl_invalid = 0;
      int fsl_rx_value = 0;

      // Assume it's a control packet. OK for the common class/type
fields.
      CTL_STAT_PKT_PTR pkt_hdr = (CTL_STAT_PKT_PTR) ha_rx_buffer;

      if (VERBOSE) {printf("%s: Before reading FSL, fsl_error=%d,
fsl_invalid=%d\r\n", __FUNCTION__, fsl_error, fsl_invalid);}
      fsl_rx_value = 0;
      if (VERBOSE) {print_packet_bin((char *)&fsl_rx_value, 4);}

      // First read the header
      if (VERBOSE) {printf("%s: Read packet header from FSL ",
__FUNCTION__);}
      if (class == PKT_CLASS_CTL) {              // Control
```

```
            if (VERBOSE) {printf("CONTROL\r\n");}
            getfslx(fsl_rx_value, FSL_CONTROL_HW_ADMINISTRATOR,
FSL_NONBLOCKING);
            fsl_isinvalid(fsl_invalid);
      } else {                                    // Data
            if (VERBOSE) {printf("DATA\r\n");}
            microblaze_nbread_datafsl(fsl_rx_value,
                                      FSL_DATA_HW_ADMINISTRATOR);
            fsl_isinvalid(fsl_invalid);
      }

      if (VERBOSE) {printf("%s: After reading FSL, fsl_error=%d,
fsl_invalid=%d\r\n", __FUNCTION__, fsl_error, fsl_invalid);}
      if (VERBOSE) {print_packet_bin((char *)&fsl_rx_value, 4);}

      if (fsl_invalid) {
            *num_ha_rx_bytes = 0;
      } else { // Valid read, received at least FSL_WIDTH bytes
            type = get_packet_field( pkt_hdr, type );
            *num_ha_rx_bytes = PKT_HEADER_LEN;     // Has to match FSL
width

            if (type == PKT_TYPE_SINGLE) {             // Short packet
                  printf("HA_CPU%s: Received a short packet from
HA\r\n", __FUNCTION__);
                  memcpy(ha_rx_buffer, &fsl_rx_value, *num_ha_rx_bytes);
                  print_packet_bin(ha_rx_buffer, *num_ha_rx_bytes);
            } else {                                   // Burst packet
                  // …
            }
      }
      return LIBPR_SUCCESS;
}
```

The get_packet_field() macro is defined in pr_packet.h:

```
#define get_packet_field( packet_pointer, field_name ) \
        packet_pointer->field_##field_name
```

Finally, send_packet_to_os() function forwards the packet to OS via mailbox and interrupts are enabled. This flow assumes that packets from HA/PRR are always forwarded to OS because at the time of creation of this framework, there were no commands sent by HA/PRR that were processed by the HCApp. Such simplification allows for skipping the call to the packet_router() and process_rx_packet() functions.

```
irq_disable();
memset(ha_rx_buffer, 0, MAX_PAYLOAD/4);
receive_packet_from_ha((char *)ha_rx_buffer,
                              &num_ha_rx_bytes,
                               PKT_CLASS_CTL);
```

```
send_packet_to_os((char *)ha_rx_buffer,
                          num_ha_rx_bytes,
                          get_mailbox_handle);
irq_enable();
```

The similar procedure is repeated for data FSL interface with PKT_CLASS_DATA directive passed to receive_packet_from_ha() function. Then the loop is repeated again by checking for pending packets from the mailbox.

4.5.6. Mailbox Driver Implementation (Linux)

This project modifies and enhances a version of the mailbox driver created by C. Foucher at Université de Nice-Sophia Antipolis. Both revisions are now available on Source Forge as an open source project [84]. The original version contained low-level mailbox access functions, kernel-user-space communication via device node, and loadable Linux kernel module initialization and cleanup functions. It was tested with a PowerPC architecture and contained several specialized assembly instructions.

This project improves Foucher's version by adding support for interrupt handling with top and bottom halves (i.e., two phases used in Linux for interrupt handling that minimize the time in interrupt context); Netlink communication between kernel and user space; as well as a configuration file with additional directives that allow enabling Netlink, disabling PowerPC specific instructions and allowing configuring bus endian type.

The driver release 2.0.0 consists of 9 files archived in mailbox_linux_drivers.7z.

```
$ ls -al
total 68
drwxr-xr-x 4 victor victor  4096 2012-02-07 01:30 ./
drwxr-xr-x 4 victor victor  4096 2012-02-07 01:33 ../
-rwxr--r-- 1 victor victor    79 2012-02-07 01:30 CLEAN*
drwxr-xr-x 2 victor victor  4096 2012-02-07 01:30 config-microblaze/
drwxr-xr-x 2 victor victor  4096 2012-02-07 01:32 config-powerpc/
-rw-r--r-- 1 victor victor  1450 2012-02-07 01:30 driver_config.h
-rw-r--r-- 1 victor victor 24689 2012-02-07 01:30 mailbox_driver.c
-rw-r--r-- 1 victor victor  2289 2012-02-07 01:30 mailbox_driver.h
-rw-r--r-- 1 victor victor   231 2012-02-07 01:30 Makefile
-rw-r--r-- 1 victor victor  1506 2012-02-07 01:30 Makefile_petalinux
-rw-r--r-- 1 victor victor  3525 2012-02-07 01:30 README
```

Table 13: Mailbox driver files (Linux)

File	Description
*CLEAN	Cleans object and intermediate files.
config-microblaze/driver_config.h	Configuration file example for MicroBlaze processor with interrupt support and Netlink communication. This file is used for TERRAE.
config-microblaze/driver_config.h	Configuration file example for PowerPC processor (not used for TERRAE).
driver_config.h	Configuration file used by the driver. For TERRAE it is a copy of config-microblaze/driver_config.h.
mailbox_driver.c	Driver C code
mailbox_driver.h	Driver header file
Makefile	Generic Makefile (not used by TERRAE).
Makefile_Petalinux	Makefile used with PetaLinux (used by TERRAE).
README	Documentation

The driver is configured by defining the appropriate directives in ./driver_config.h header file. The directives important for TERRAE are:

- #define NETLINK. When uncommented this directive enables Netlink communication between kernel and user spaces;
- #define ARCH ARCH_DEFAULT. This directive ignores PowerPC specific instructions that the driver also supports;
- #define ENDIANNESS ENDIANNESS_BIG. This directive tells the driver to use big endianness when accessing mailbox's registers via PLB bus.

Further directives are configured in mailbox_driver.h header file. These are:

- #define IRQ_NUM 0. This directive matches the IRQ selected during hardware system implementation in EDK;
- #define NETLINK_USER 31. This directive defines a new socket protocol type used for TERRAE. This number is selected because it is not used in linux/include/linux/netlink.h;
- #define MAX_PAYLOAD 1024. This directive specifies the maximum size in bytes of the supported Netlink message. It also aligns with the physical size of the mailbox;
- #define NUMBER_OF_MAILBOXES 1. This directive specifies the number of supported mailboxes. Keep it at one for the TERRAE.

To load the driver in Linux, first use *mknod* utility to create a device node and associate it with major and minor numbers, and then execute *insmod* as below:

```
# mknod /tmp/mailbox_driver c 32 0
# insmod /lib/modules/kernel/drivers/
misc/mailbox_driver.ko
```

The driver prints initialization messages:

```
Initializing interrupts..... Done
Allocating MBox tx/rx buffers..... Done
Registering IOs..... Done
Requesting IO ports.............
- Real address 0   : 0x71010000.... Done
Registering driver to kernel..... Done
Creating netlink socket..... Done
```

Once the driver is loaded, a tasklet is declared and the following module initialization function is called. The latter allocates memory dynamically for a receive buffer and initializes top and bottom halves of the interrupt handler.

```
static DECLARE_TASKLET(read_mailbox_tasklet, read_mailbox_handler, 0);
static int __init mbox_driver_init(void) {
…
mbox_rx_buffer = kmalloc(MAX_PAYLOAD, GFP_KERNEL)
…
request_irq(IRQ_NUM, IntrHandler, NULL, names[0], NULL)
…
nl_sk = netlink_kernel_create(NETLINK_USER, 0, mbox_nl_recv_msg,
THIS_MODULE)
…
}
```

In the user-space-to-mailbox receive path (static void mbox_nl_recv_msg (struct sock *sk, int len)) dequeues a Netlink socket message previously placed in the queue by a user space application (i.e. daemon process). It then extracts a packet and writes it to mailbox one 32-bit word at a time.

```
// Netlink message length minus header
// Note: ignores padding if any
msg_size   = nlh->nlmsg_len - NLMSG_HDRLEN;
remaining  = msg_size;
word_index = 0;
while ((skb = skb_dequeue(&sk->sk_receive_queue)) != NULL) {
      // Process netlink message pointed by skb->data
      nlh = (struct nlmsghdr *)skb->data;
```

```
}
msg = (char*)nlmsg_data(nlh);
while (remaining != 0) {
      if (XMbox_mIsFull(MAILBOX_ID)) {           // Assume using one
mailbox
            printk(KERN_INFO "%s: Mailbox is full\n", __FUNCTION__);
            break;
      }
      i = 0;
      temp = 0;
      while ((remaining > 0) && i < 4)
      {
            temp |= ((unsigned int)(*(msg + word_index * 4 + i)) << 24)
>> i * 8;
            remaining--;
            i++;
      }
      XMbox_mWriteMBox(MAILBOX_ID, temp);
      word_index++;
}
```

In the mailbox-to-user-space receive path, IRQ invokes top half of Linux interrupt handler, which has the following function prototype:

```
static irqreturn_t IntrHandler(int irq, void *dev_id, struct pt_regs
*regs)
```

This command reads the entire mailbox a word at a time (4 bytes), and then saves contents in an internal buffer. Then the bottom half of the interrupt driver is called with the following command:

```
tasklet_schedule(&read_mailbox_tasklet);
```

When a tasklet (static void read_mailbox_handler(unsigned long unused)) is executed by Linux at a later point in time, it creates a unicast Netlink message and sends it to user space, where the appropriate application will detect the message and process it. During this time, interrupts are disabled to prevent receive buffer override.

```
disable_irq(IRQ_NUM);
skb      = nlmsg_new(msg_size, 0);
if(!skb)
{
      printk(KERN_ERR "Failed to allocate new skb\n");
      return;
}
nlh = nlmsg_put(skb, 0, 0, NLMSG_DONE, msg_size, 0);
NETLINK_CB(skb).dst_group = 0;     // Unicast
```

```
memcpy(nlmsg_data(nlh), mbox_rx_buffer, msg_size);
res = nlmsg_unicast(nl_sk, skb, app_pid);
enable_irq(IRQ_NUM);
```

As it stands, the driver has the following limitations. These limitations and how to overcome them will be discussed in Section 5.4.1.

- IRQ is statically assigned;
- When using interrupts with Netlink, only one mailbox can be used at a time;
- Packet cannot exceed the size of the mailbox;
- When using interrupts with Netlink, only one Netlink packet should reside in the mailbox at a time (i.e. single or multiword). This limitation aligns with Intraframework, which also does not support burst of packets (but supports burst of words). Thus, it is critical to pay attention to flow control of the packets. Specifically, the current implementation of the bottom half interrupt handler (i.e., mailbox-to-user-space direction) reads everything from the mailbox, but the top half tasklet disregards anything that is beyond the first packet. This shortcoming can be dealt with in future versions. Another reason is that when a mailbox's FIFO overflows in the user-space-to-mailbox direction, the driver will drop the extra packets.

4.5.7. Mailbox Driver Implementation (Standalone on HA_CPU)

The mailbox driver on HA_CPU has a simple implementation. To retrieve data from mailbox, an interrupt routine is used as shown in Figure 45, with code also provided below. Currently it is built into the HCApp rather than having its own set of files. This diagram is a copy of Figure 28 that is reused here for convenience of the reader.

```
int mp_handle_irq ()
{
    int status;

    irq_disable();
    if (status = XMbox_IsEmpty(&mbox)) {
        if (VERBOSE) {printf("HA_CPU: Mailbox is empty, status =
%d\r\n", status);}
    }
    else {
        memset(rx_buffer, 0, MAX_PAYLOAD);
        XMbox_Read (&mbox, rx_buffer, MAX_PAYLOAD, &num_rx_bytes);
        pending_packet_from_os = 1;

        if (VERBOSE) {printf("HA_CPU: Read %d bytes. Got an intr: %d
(0x%x). \r\n", num_rx_bytes, *rx_buffer, *rx_buffer);}
        if (VERBOSE) {print_packet_bin((char *)rx_buffer,
num_rx_bytes);}
```

```
        if (VERBOSE)
{print_ctl_stat_packet_header((CTL_STAT_PKT_PTR)rx_buffer);}

        if (VERBOSE) {printf("HA_CPU: delay packet processing until
scheduled\r\n");}
    }

    irq_enable();
}
```

mp_handle_irq() executed on HA_CPU as part of HCApp

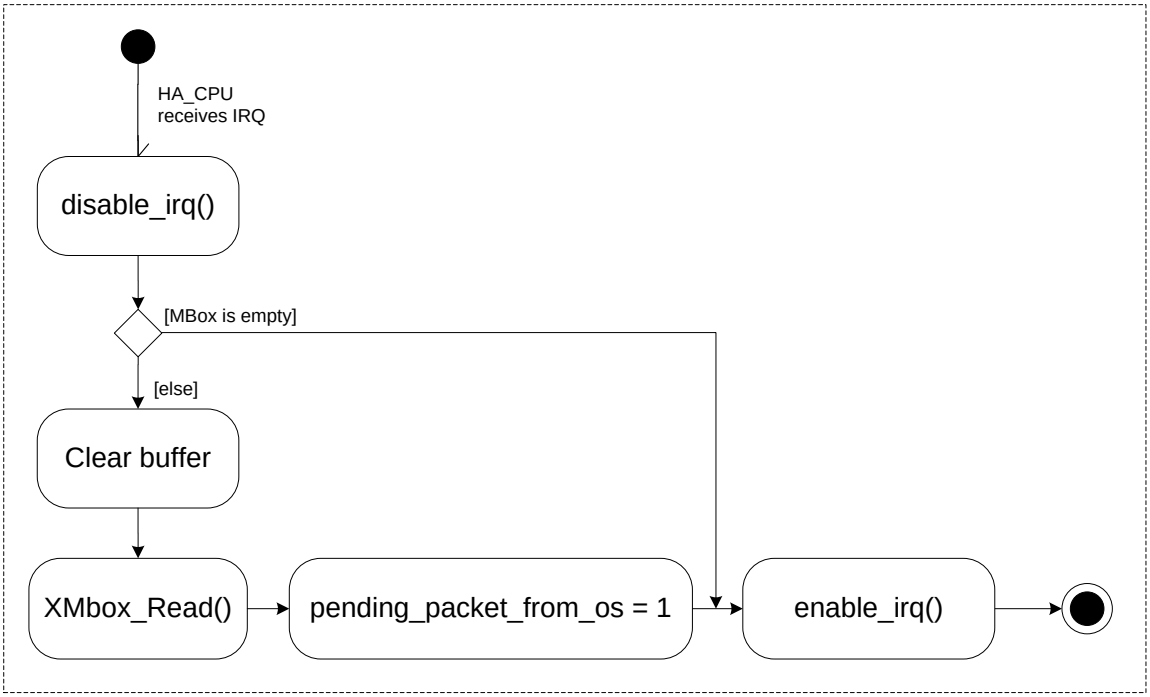


Figure 45: HA_CPU mailbox driver interrupt handler UML activity diagram

A function to send data to mailbox from HA_CPU is placed into the platform-specific part of the *libpr*, libpr/os-standalone/pr_sys.c, and abstracted by send_packet_to_os() API call, as shown in the following code snippet:

```
int send_packet_to_os(char *tx_buffer,
                      int tx_pkt_len,
                      void * (*get_mailbox_handle)(void)) {
      int num_tx_bytes;
      XMbox *mbox;

      if (tx_pkt_len > 0) {
            if (VERBOSE) {printf("%s: Ready to send packet\r\n",
__FUNCTION__);}
            if (VERBOSE) {print_packet_bin(tx_buffer, tx_pkt_len);}
```

```
            // Get address of mailbox
            mbox = (XMbox *) get_mailbox_handle();

            // TBD: Consider casting to u32* 2nd and 4th arguments
            XMbox_Write(mbox, tx_buffer, tx_pkt_len, &num_tx_bytes);
            if (num_tx_bytes != tx_pkt_len) {
                  printf("Warning: Wanted to send %d bytes, but sent %d
instead\r\n",
                              tx_pkt_len, num_tx_bytes);
                  return LIBPR_FAIL;
            }
      } else {
            if (VERBOSE) {printf("%s: Zero length packet. There is
nothing to send\r\n", __FUNCTION__);}
      }

      return LIBPR_SUCCESS;
}
```

4.5.8. PR Library (LIBPR) APIs

Libpr consists of several packages, each implemented in its own C file. Their short summaries are provided in Table 14. More detailed descriptions can be found in the following subsections and source code online [89].

Table 14: libpr file descriptions

Path under libpr/	uP Target	File Description
os-linux/pr_sys.c os-linux/pr_sys.h	LINUX_CPU	Target specific functions for handling named pipe and mailbox communication.
os-standalone/pr_sys.c os-standalone/pr_sys.h	HA_CPU	Target specific functions for handling mailbox and FSL interfaces.
os-standalone/SysACE_Header.c os-standalone/SysACE_Header.h os-standalone/xhwicap_parse.h	HA_CPU	Functions for ICAP and CF IP management during reconfiguration.
pr_comm.c pr_comm.h	LINUX_CPU	Functions for communication between agents.
pr_packet.c pr_packet.h	Both	Packet processing, creation, routing, command execution.
pr_status.c pr_status.h	LINUX_CPU	Functions for PR status table management.
pr_user_comm.c pr_user_comm.h	LINUX_CPU	Functions for communication between the DP and CLT or user applications in Linux's user space.

pr_utils.c pr_utils.h	Both	Functions for formatted packet printing to stdio.

4.5.8.1. pr_packet.c Package

The pr_packet package defines functions for packet processing: creation, deassembly, processing and routing. Table 15 summarises their description as well as the processor target selected with #ifdef C directives. The following subsections describe several key functions.

Table 15: pr_packet.c package functions

Function	uP Target	Description
get_pkt_len()	Both	Knowing packet pointer, returns its length in bytes.
is_burst_pkt()	Both	Returns 1 if the input packet's TYPE field indicates a burst packet.
create_ctl_stat_packet ()	Both	Create control command or status packet.
create_burst_ctl_packet()	Both	Create control command or status burst packet.
create_burst_data_packet()	Both	Create burst data packet.
send_ctl_packet()	LINUX_CPU	Creates control packet, send via named pipe.
execute_ctl_command()	Both	Execute command.
process_ctl_stat()	LINUX_CPU	Process control status packet.
process_data()	Both	Process received burst packet (not used).
process_rx_burst_packet()	Both	Process received burst data packet (not used).
get_cmd_dest()	Both	Retrieve destination agent ID based on the information extracted from packet header.
packet_router()	Both	Forwards a packet if the function is executed on an agent other than the expected destination.
send_cmd_prr_reconfig()	LINUX_CPU	Creates CMD_RECONFIG_PRR reconfiguration command packet and sends via named pipe.
send_ctl_command()	LINUX_CPU	Create control packet by name and send it. Takes text versions of PRM and command names, loads configuration files, creates and sends the packet (not implemented).
process_rx_packet()	Both	Process the received packet (i.e. execute a command, process status or data packet).

4.5.8.1.1. pr_packet.c : create_burst_ctl_packet()

The create_burst_ctl_packet(), despite its name, creates burst and single word packets.

Table 16: create_burst_ctl_packet()

Argument	Description
In: int command	Command ID
In: int is_command	Flag indicating whether it is a control command or status packet
In: int dest_prr	Destination PRR ID
In: int dest_ha	Destination Hardware Administrator ID
In: char *data_buffer	Pointer to packet buffer
In: int burst_size	Burst of packet payload following the header in bytes
In: char **tx_buffer	Address of transmit packet buffer
In: int *tx_pkt_len	Length of valid data in the transmit packet buffer
Out: int status	Returns LIBPR_SUCCESS

The create_burst_ctl_packet() uses set_packet_field() macro to assemble overhead and append data. The resulting packet buffer pointer is updated for use by the calling function. Below is the snippet of code that uses the macro.

```
set_packet_field( pkt_hdr, cmd_param,     command);
set_packet_field( pkt_hdr, is_command,    is_command);
set_packet_field( pkt_hdr, class,         PKT_CLASS_CTL);
set_packet_field( pkt_hdr, type,          type);
set_packet_field( pkt_hdr, dest_prr,      dest_prr);
set_packet_field( pkt_hdr, dest_ha,       dest_ha);
set_packet_field( pkt_hdr, payload_short, burst_size);
```

The PAYLOAD_SHORT field is used for packet size in this case. Technically, when a short packet is sent, this field can be used to send parameters. However, in the first version of the project no such command was created. In future, it is suggested to supersede this function with create_ctl_stat_packet(), which was unit tested, but not integrated into the project. Here is the header creation code from this function:

```
// Burst packet
if (burst_size > 0) {
```

```
        set_packet_field( pkt_hdr, type,             PKT_TYPE_BURST);
        set_packet_field( pkt_hdr, payload_short,    burst_size);

        // Copy data portion of the packet to the space following the
header
        memcpy(BURST_CTL_PKT_DATA(pkt_hdr), payload, burst_size);
    // Short packet
    } else {
        set_packet_field( pkt_hdr, type,             PKT_TYPE_SINGLE);
        set_packet_field( pkt_hdr, payload_short,    payload_short); //
2 Bytes
    }
      set_packet_field( pkt_hdr, cmd_param,        cmd_param);
      set_packet_field( pkt_hdr, is_command,       is_command);
      set_packet_field( pkt_hdr, class,            PKT_CLASS_CTL_STAT);
      set_packet_field( pkt_hdr, dest_prr,         dest_prr);
      set_packet_field( pkt_hdr, dest_ha,          dest_ha);
```

The set_packet_field() macro is defined in pr_packet.h:

```
      #define set_packet_field( packet_pointer, field_name,
new_field_value ) \
      packet_pointer->field_##field_name = new_field_value
```

4.5.8.1.2. pr_packet.c: packet_router()

The packet_router() function serves the following two purposes:

- Used by the DP and HCApp to route packets based on decoded packet header flags (class and command ID) and the processing agent ID;
- If a command in a packet is meant to be executed by the processing agent, this function launches its execution.

The packet_router() has a different API depending on whether it is compiled for LINUX_CPU or HA_CPU. This is controlled by LINUX_C and STANDALONE_C directives. The first three arguments are the same, while the HA_CPU adds additional void callback function pointer to a mailbox handle. This callback allows for architecture specific code defined in libpr/pr_sys.c to obtain a mailbox handle without defining mailbox specific headers and types.

Table 17: packet_router()

Argument	Description
In: char *rx_buffer	Pointer to the received packet buffer
In: int rx_pkt_len	Length of valid data in the received packet buffer
In: int agent_id	ID of agent application
In: int *is_dest_reached	Flag indicating whether destination agent has been reached
In: void * (*get_mailbox_handle)(void))	Callback function pointer to a mailbox handle. Used only when STANDALONE_C is defined.
Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

```
#ifdef LINUX_C
int packet_router(char *rx_buffer,
                  int rx_pkt_len,
                  int agent_id,
                  int *is_dest_reached) {
#endif // LINUX_C
#ifdef STANDALONE_C
int packet_router(char *rx_buffer,
                  int rx_pkt_len,
                  int agent_id,
                  int *is_dest_reached,
                  void * (*get_mailbox_handle)(void)) {
#endif // STANDALONE_C
```

Agent IDs defined in pr_comm.h are AGENT_ID_CONSOLE, AGENT_ID_DAEMON, AGENT_ID_HA_CPU, AGENT_ID_HA, and AGENT_ID_PRR. Since the packet_router() is currently supported on the DP and HCApp, the corresponding agent IDs are AGENT_ID_DAEMON and AGENT_ID_HA_CPU respectively. These two directives are statically passed and compiled as inputs of the packet router.

The function begins by setting a default behavior to forward the input packet as seen from the UML activity diagram in Figure 46. get_cmd_dest() returns the destination of the packet based on the extracted header information. A forward_packet flag is set if the destination is different from the ID of the application that is executing this code. LINUX_C and STANDALONE_C directives are used for the selection of the correct API calls depending on whether the function is executing on LINUX_CPU or HA_CPU. See the code snippet at the end of this section. For a Linux application such as DP,

send_packet_to_ha() function is called to forward the packet to HA/PRR via mailbox. For HCApp on HA_CPU, for the same destination (i.e. HA/PRR) the same API function is called. However, their respective implementations are different.

Such architecture dependent functions are defined in pr_sys.c and selected by TERRAE's tools before compilation. As a result, the send_packet_to_ha() on Linux uses Netlink sockets to send the packet to mailbox, while the same function on HA_CPU uses MicroBlaze's FSL driver write function calls.

If packet_router() is executed by HCApp and the packet is destined for DP, send_packet_to_os() function is called. It triggers HA_CPU's low-level mailbox driver XMbox_Write() to pass the packet to OS. After forwarding the packet, the is_dest_reached flag is cleared indicating to the function's caller that the packet and its command should not be executed locally.

```
if (forward_packet) {
            switch (cmd_dest) {
#ifdef LINUX_C
            case AGENT_ID_HA_CPU:
            case AGENT_ID_HA:
            case AGENT_ID_PRM:
                  send_packet_to_ha(rx_buffer, rx_pkt_len, agent_id);
                  *is_dest_reached = 0;
                  break;
#endif
#ifdef STANDALONE_C
            case AGENT_ID_HA:
            case AGENT_ID_PRM:
                  send_packet_to_ha((u32*)rx_buffer, rx_pkt_len);
                  *is_dest_reached = 0;
                  break;
            case AGENT_ID_DAEMON:
                  send_packet_to_os(rx_buffer, rx_pkt_len,
get_mailbox_handle);
                  *is_dest_reached = 0;
                  break;
#endif
            default:
                  printf("%s: Unknown command destination %d\r\n",
__FUNCTION__, cmd_dest);
                  *is_dest_reached = 0;
                  return LIBPR_FAIL;
            }
      } else {
            *is_dest_reached = 1;
}
```

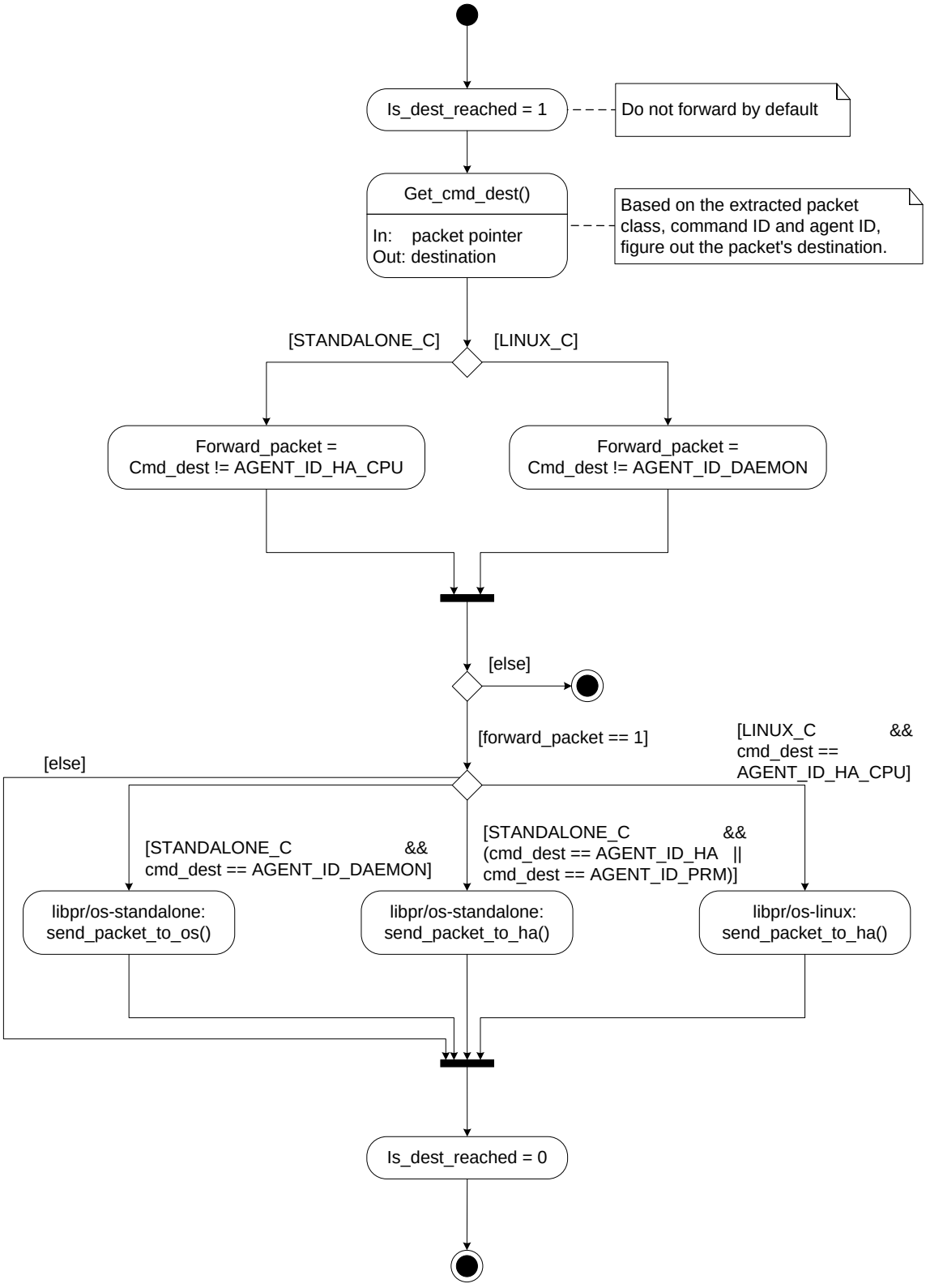


Figure 46: pr_packet.c : packet_router() UML activity diagram

4.5.8.1.3. pr_packet.c : process_rx_packet()

The process_rx_packet() function is used for processing a received packet. Its prototype is different depending on whether it is compiled for the Linux or HA_CPU agent, as seen from the source code below. It takes in a pointer to the packet buffer, and also a void callback function pointer to mailbox handler, similar to packet_router() (see the packet_router() description). For Linux, there is also an additional pointer to PR status table. Figure 47 visualizes the function in UML activity diagram.

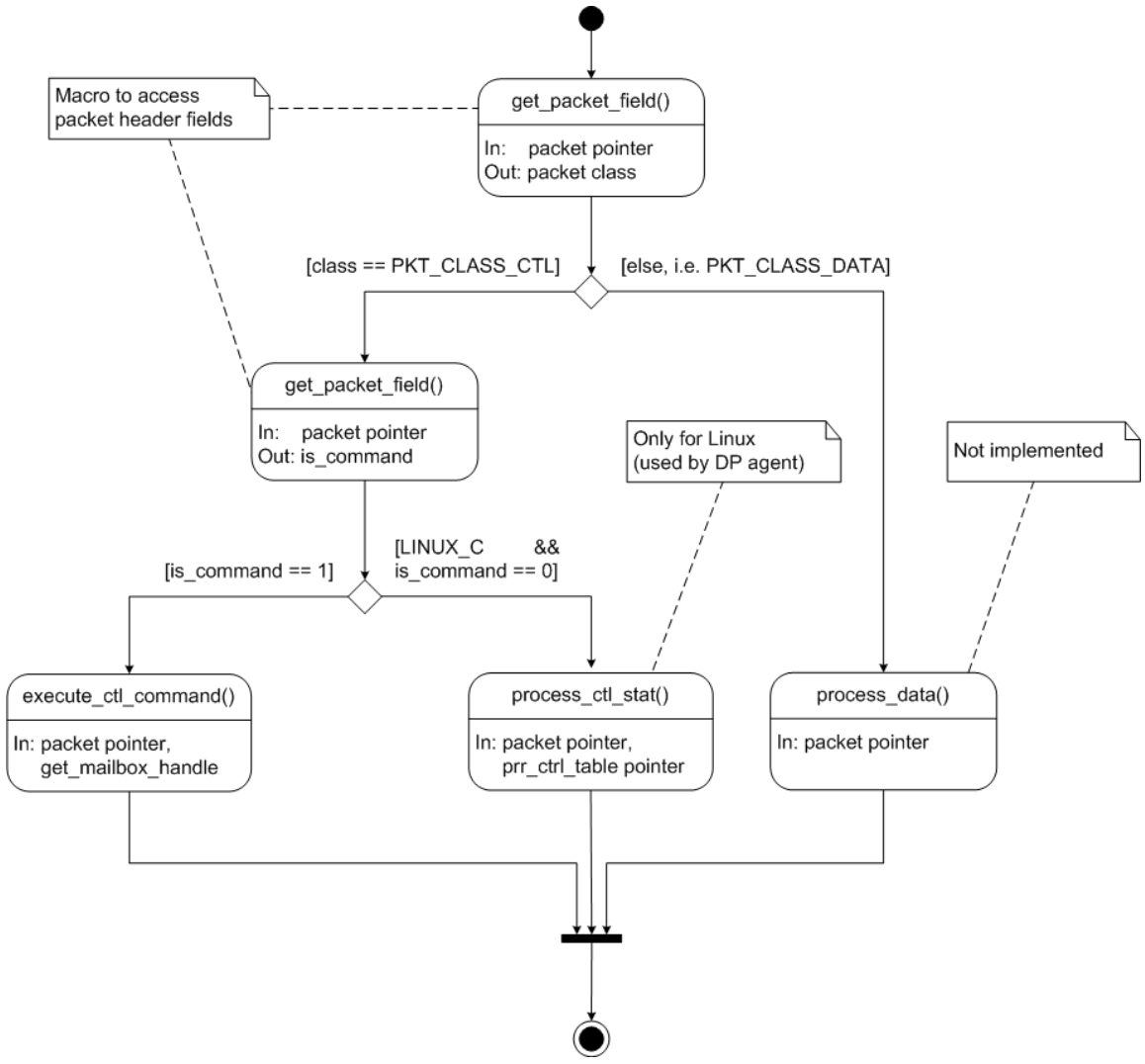


Figure 47: pr_packet.c : process_rx_packet() UML activity diagram

Table 18: process_rx_packet()

Argument	Description
In: char * rx_packet	Pointer to the received packet buffer
In: prr_ctrl_table_t * prr_ctrl_table	Pointer to PR status table. Used only when LINUX_C is defined
In: void * (*get_mailbox_handle)(void))	Callback function pointer to a mailbox handle
Out: int status	Returns LIBPR_SUCCESS

```
#ifdef LINUX_C
int process_rx_packet(char * rx_packet,
                      prr_ctrl_table_t * prr_ctrl_table,
                      void * (*get_mailbox_handle)(void)) {
#endif
#ifdef STANDALONE_C
int process_rx_packet(char * rx_packet,
                      void * (*get_mailbox_handle)(void)) {
#endif
```

4.5.8.2. pr_status.c Package

The pr_status package contains functions for PR status table management. All are targeted for LINUX_CPU. Their descriptions are provided below.

4.5.8.2.1. pr_status.c : init_prr_ctrl_table()

This function initializes PR status table. Currently, it hard codes the values for two PRR used in TERRAE. However, it can be extended to read system's configuration from template text files.

Table 19: init_prr_ctrl_table()

Argument	Description
In: prr_ctrl_table_t **prr_ctrl_table	Address of pointer to the created PR status table
Out: int status	Returns LIBPR_SUCCESS

4.5.8.2.2. pr_status.c : print_prr_ctrl_record()

This function prints a single PR status table entry to stdout.

Table 20: print_prr_ctrl_record()

Argument	Description
In: prr_ctrl_record_t *prr_ctrl_record	Pointer to PR status table record
Out: int status	Returns LIBPR_SUCCESS

4.5.8.2.3. pr_status.c : print_prr_ctrl_table()

This function prints a PR status table consisting of multiple records.

Table 21: print_prr_ctrl_table()

Argument	Description
In: prr_ctrl_table_t *prr_ctrl_table	Pointer to PR status table
Out: int status	Returns LIBPR_SUCCESS

4.5.8.2.4. pr_status.c : prr_ctrl_get_num_records()

This function returns a number of records in the PR status table.

Table 22: prr_ctrl_get_num_records()

Argument	Description
In: prr_ctrl_table_t *prr_ctrl_table	Pointer to PR status table
Out: int num_records	Returns a number of PR status table's records

4.5.8.2.5. pr_status.c : find_prr_ctrl_record()

This function returns a pointer to PR status table's record indexed by (ha, prr).

Table 23: find_prr_ctrl_record()

Argument	Description
In: prr_ctrl_table_t *prr_ctrl_table	Pointer to PR status table
In: int ha	Hardware Administrator ID
In: int prr	PRR ID
Out: prr_ctrl_record_t *	Returns a pointer to the found PR status table record

4.5.8.2.6. pr_status.c : update_prr_ctrl_table()

This function updates the PR status table based on the received packet header information and is_dest_reached flag.

Table 24: update_prr_ctrl_table()

Argument	Description
In: prr_ctrl_table_t *prr_ctrl_table	Pointer to PR status table
In: char *rx_packet	Pointer to packet buffer
In: int rx_pkt_len	Packet length in bytes
In: is_dest_reached	Flag indicating whether the received packet has reached its destination
Out: int status	Returns LIBPR_SUCCESS

4.5.8.3. pr_user_comm.c Package

The pr_user_comm package contains functions for communication between the DP, and CLT or user applications in Linux's user space. Currently, it defines functions for named pipe initialization, reading and writing. All are targeted for LINUX_CPU. See Table 25 for function descriptions.

Table 25: pr_user_comm.c package functions

Function	uP Target	Description
init_named_pipe()	LINUX_C	Creates named pipe
read_pipe()	LINUX_C	Reads from named pipe
write_pipe()	LINUX_C	Writes to named pipe

4.5.8.4. pr_comm.c Package

The pr_comm package contains functions used for communication between agents; for example, Netlink socket send/receive function for accessing mailbox from the

Linux side. It is based on the code originally presented by K. K. He in the Linux Journal [85], and updated for use in TERRAE with kernel 2.6.20 [89].

4.5.8.4.1. pr_comm.c : get_nl_protocol() (target: LINUX_CPU)

This function is used for lookup of Netlink protocol number in case multiple Netlink connections are required. As it stands, only a single connection is used; thus, this function always returns 31, as defined in libpr/pr_comm.h.

Table 26: get_nl_protocol()

Argument	**Description**
In: int agent_id	ID of agent application (i.e. AGENT_ID_CONSOLE, AGENT_ID_DAEMON),
In: int nl_direction	Direction of communication (i.e. NETLINK_TX, NETLINK_RX),
Out: int nl_protocol	Returns Netlink protocol number. Always the same for TERRAE. #define NETLINK_PR_DAEMON_TX 31 #define NETLINK_PR_DAEMON_RX 31

4.5.8.4.2. pr_comm.c : init_netlink() (target: LINUX_CPU)

This function reserves a buffer of MAX_PAYLOAD bytes, creates a Netlink socket and associates it with the specified protocol number.

Table 27: init_netlink()

Argument	**Description**
In: int nl_protocol	Netlink protocol number
Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

4.5.8.4.3. pr_comm.c : send_nl_msg_to_kernel() (target: LINUX_CPU)

The send_nl_msg_to_kernel() is used for forwarding messages from user space application to kernel space of a driver associated with the registered Netlink protocol number (i.e. mailbox driver).

Table 28: send_nl_msg_to_kernel()

Argument	Description
In: char *data_buffer	Pointer to data buffer to be sent
In: int data_len	Length of valid data in the buffer
In: int nl_protocol	Netlink protocol number
Out: int status	Returns LIBPR_SUCCESS

As seen from the source code below, send_nl_msg_to_kernel() starts by calling init_netlink(), which opens a socket of Netlink type and retrieves agent's process ID to be used by kernel to send messages back to the DP. Then send_nl_msg_to_kernel() fills out the fields of the Netlink message header, appends packet buffer and sends the message using a standard socket sendmsg() function. The agent's PID is passed to kernel via the Netlink header.

```
int send_nl_msg_to_kernel(char *data_buffer,
                          int data_len,
                          int nl_protocol) {
      if (!netlink_initialized) {
            init_netlink(nl_protocol);
      }

      // Clear previous buffer
      memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));

      // Create outgoing message
      nlh->nlmsg_len   = NLMSG_LENGTH(data_len);  // Does data_len + 16
      nlh->nlmsg_type  = 0;
      nlh->nlmsg_pid   = app_pid;
      nlh->nlmsg_flags = 0;

      // Copy data portion of the packet to the space following the
header
      memcpy(NLMSG_DATA(nlh), data_buffer, data_len);

      iov.iov_base    = (void *)nlh;
      iov.iov_len     = nlh->nlmsg_len;
      msg.msg_name    = (void *)&dest_addr;
      msg.msg_namelen = sizeof(dest_addr);
      msg.msg_iov     = &iov;
      msg.msg_iovlen  = 1;

      sendmsg(sock_fd,&msg,0);

      // Keep the socket open
```

```
      return LIBPR_SUCCESS;
} // send_nl_msg_to_kernel
```

4.5.8.4.4. pr_comm.c : receive_nl_msg_from_kernel() (target: LINUX_CPU)

The receive_nl_msg_from_kernel() function is used for retrieving messages from a Netlink socket receive buffer located in user space and associated with the registered Netlink protocol number (i.e., the mailbox driver).

Table 29: receive_nl_msg_from_kernel()

Argument	Description
In: char **rx_buffer	Address of receive buffer pointer
In: int *rx_data_len	Pointer to the length of valid data in the created receive buffer
In: int nl_protocol	Netlink protocol number
Out: int status	Returns LIBPR_SUCCESS

The message has been previously placed there by the mailbox driver's bottom half. It uses a standard socket receive function, recvmsg(), accomplishing a non-blocking read. The buffer pointer is updated and returned by address to the calling function [89].

```
int receive_nl_msg_from_kernel(char **rx_buffer,
                           int *rx_data_len,
                           int nl_protocol) {
    …

      //
      // Set recvmsg() flags
      // i.e. Non-blocking read: MSG_DONTWAIT
      //
      int recvmsg_flags = MSG_DONTWAIT;

      if (!netlink_initialized) {
            init_netlink(nl_protocol);
      }

      // Clear previous buffer
      memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));


    …
      iov.iov_base    = (void *)nlh;
```

```
      iov.iov_len     = nlh->nlmsg_len;
      msg.msg_name    = (void *)&dest_addr;
      msg.msg_namelen = sizeof(dest_addr);
      msg.msg_iov     = &iov;
      msg.msg_iovlen  = 1;

      // Non-blocking receive
      status = recvmsg(sock_fd, &msg, recvmsg_flags);
        if (status == -1) {
            if (VERBOSE) {printf("%s recvmsg(): nothing received\n",
__FUNCTION__);}
            *rx_buffer   = NULL;
            *rx_data_len = 0;
      } else {
            nlmsg_rx_data = NLMSG_DATA(nlh);
            nlmsg_rx_data_len = 4;      // TBD: Assume 4 bytes for now
    …
            *rx_buffer   = nlmsg_rx_data;
            *rx_data_len = nlmsg_rx_data_len;
      } // !if (status == -1)

      return LIBPR_SUCCESS;
} // receive_nl_msg_from_kernel
```

4.5.8.5. pr_utils.c Package

The pr_utils package has functions for formatted packet printing to stdio. See Table 30 for function descriptions.

Table 30: pr_utils.c package functions

Function	uP Target	Description
print_data_packet_header()	Both	Prints data packet header
print_ctl_stat_packet_header()	Both	Prints command/status control packet
print_packet_bin()	Both	Prints header in ASCII and char formats

4.5.8.6. os-linux/pr_sys.c Package

The os-linux/pr_sys.c package is sourced for Linux based applications (i.e. CLT, DP and user applications on LINUX_CPU). It contains APIs that encapsulate interface-specific low-level driver calls. The generic names of the APIs allow for function name reuse when compiled against different targets.

4.5.8.6.1. os-linux/pr_sys.c : pipe_send_packet_to_ha()

The pipe_send_packet_to_ha() function writes the contents of the input character buffer into a named pipe using a standard write_pipe() call.

Table 31: os-linux/pr_sys.c : pipe_send_packet_to_ha()

Argument	Description
In: char *tx_buffer	Pointer to transmit buffer
In: int tx_pkt_len	Length of valid bytes in the transmit buffer
Out: int status	Returns output of write_pipe()

4.5.8.6.2. os-linux/pr_sys.c : send_packet_to_ha()

The send_packet_to_ha() function looks up the Netlink protocol number and sends the contents of the input character buffer into the corresponding socket.

Table 32: os-linux/pr_sys.c : send_packet_to_ha()

Argument	Description
In: char *tx_buffer	Pointer to transmit buffer
In: int tx_pkt_len	Length of valid bytes in the transmit buffer
In: int agent_id	ID of agent application used for determination of Netlink protocol number
Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

4.5.8.6.3. os-linux/pr_sys.c : receive_packet()

The receive_packet() function looks up the Netlink protocol number and calls the non-blocking receive_nl_msg_from_kernel() to extract a packet from the Netlink message buffer.

Table 33: os-linux/pr_sys.c : receive_packet()

Argument	Description
In: char *tx_buffer	Pointer to transmit buffer
In: int tx_pkt_len	Length of valid bytes in the transmit buffer
In: int agent_id	ID of agent application used for determination of Netlink protocol number

Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

4.5.8.7. os-standalone/pr_sys.c Package

The os-standalone/pr_sys.c package is sourced for applications running on a platform without an OS (i.e., HCApp on HA_CPU). It contains APIs that are encapsulating interface specific low-lever driver calls. The generic names of the APIs allow for function name reuse when compiled against different targets.

4.5.8.7.1. os-standalone/pr_sys.c : send_packet_to_ha()

The send_packet_to_ha() function determines the class of packet (i.e., control versus data), and writes the packet to the corresponding FSL interface of Hardware Administrator connected to HA_CPU.

Table 34: os-standalone/pr_sys.c : send_packet_to_ha()

Argument	Description
In: u32 *tx_buffer	Pointer to transmit buffer
In: int tx_pkt_len	Length of valid bytes in the transmit buffer
Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

4.5.8.7.2. os-standalone/pr_sys.c : send_packet_to_os()

The send_packet_to_os() function determines the class of packet (i.e., control versus data), and writes the packet to the corresponding FSL interface of Hardware Administrator connected to HA_CPU.

Table 35: os-standalone/pr_sys.c : send_packet_to_os()

Argument	Description
In: char *tx_buffer	Pointer to transmit buffer
In: int tx_pkt_len	Length of valid bytes in the transmit buffer
In: void * (*get_mailbox_handle)(void))	Callback function pointer to a mailbox handle
Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

4.5.8.7.3. os-standalone/pr_sys.c : receive_packet_from_ha()

The receive_packet_from_ha() function calls non-blocking FSL read functions to retrieve packets from the appropriate Hardware Administrator's interface (i.e., control versus data).

Table 36: os-standalone/pr_sys.c : receive_packet_from_ha()

Argument	Description
In: char *ha_rx_buffer	Pointer to receive buffer
In: int *num_ha_rx_bytes	Pointer to the length of the received message
In: int class	Packet class (i.e. control vs. data)
Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

4.5.8.7.4. os-standalone/pr_sys.c : prr_reconfig()

The prr_reconfig() function is simply a wrapper for the download_bitstream() function, which in turn performs PRM reconfiguration.

Table 37: os- standalone/pr_sys.c : prr_reconfig()

Argument	Description
In: char *bitfilename	Pointer to a bitstream file name
In: int dest_ha	Destination Hardware Administrator
In: int dest_prr	Destination PRR
Out: int status	Returns LIBPR_SUCCESS or LIBPR_FAIL

4.5.8.8. os-standalone/SysACE_Header.c Package

The SysACE_Header.c file contains functions related low-level ICAP and compact flash (CF) IP management during reconfiguration. Please refer to Table 38 below for their descriptions.

Table 38: os-linux/SysACE_Header.c

Function	uP Target	Description
init_sysace()	HA_CPU	Initializes SysACE driver for CF

init_hwicap()	HA_CPU	Initializes HW ICAP driver
get_prr_bm_gpio_addr()	HA_CPU	Finds address of bus macro GPIO based on PRR ID
XHwIcap_CF2Icap()	HA_CPU	Programs device with the specified bitstream file name
XHwIcap_ReadHeader()	HA_CPU	Parses bitstream header
download_bitstream()	HA_CPU	Turns off GPIOs and calls XHwIcap_CF2Icap() to reconfigure device

4.6. TERRAE Toolset

This section introduces additional tools that were created in order to assist the development and testing of TERRAE. Each agent application has its own set of build scripts written either in Bash or using Makefile syntax. Also, additional C applications were created to assist incremental testing of each subsystem during its integration. Moreover, several versions of EDK hardware were put together to assist this process. During that time the address offsets of peripherals were kept constant, thereby minimizing modifications in BSP and allowing reuse of the same software executable on multiple platforms. Finally, for aiding the collaborative development and to guarantee consistent releases, a set of release scripts in Bash that integrate TERRAE and Intraframework were created.

4.6.1. Development Flow

The project development was split between two workstations/environments allowing two people working in parallel and promoting cross-testing between TERRAE and Intraframework, as seen in Figure 48. The first environment with Windows and PlanAhead 10.1 was installed on two separate workstations. One of them (WS1) was used for implementing Intraframework with a single bus single processor DPR demo system [13]-[14]. The second workstation (WS2) with a Linux virtual machine (VM) was used for developing the dual bus asymmetric multiprocessing TERRAE. It included ground-up development of a dual bus hardware platform, integration of HA IP and static versions of PRMs, configuration of Linux, development of the mailbox driver, PR library and applications, and the related integration testing and fine tuning of both frameworks.

We also decided to use a Windows environment for taking the dual bus dual processor TERRAE EDK project through the PlanAhead flow for generation of DPR-enabled TERRAE. The reason is that we faced tool issues that prevented completing the bitstream generation with PlanAhead 10.1 on Linux (see Section 4.7.3). While debugging the Linux installation we continued to use PlanAhead on Windows as a workaround. As a result, the development of both TERRAE and its DPR-enabled counterpart was split between the Linux VM and Windows host on WS2. At a later point in time, the PlanAhead part of the flow was moved to WS1's Windows for convenience. An efficient flow for project file handoff between the developers was developed, as shown on the Figure 48 data flow graph. Table 39 summarizes specs for both workstations. Note that once the issue with PlanAhead installation is resolved, the full flow could be performed in Linux without modifications.

Table 39: Development workstations

	Workstation 1 (WS1) – Windows 7 host and XP VM	**Workstation 2 (WS2) – Linux VM**	**Workstation 2 (WS2) – Windows host**
Purpose	a) Intraframework development with initial single bus system; b) Integration of TERRAE EDK project into ISE, and taking it through PlanAhead PR flow to obtain DPR-enabled TERRAE.	a) TERRAE development with dual bus dual MicroBlaze hardware architecture; b) Embedded Linux and all TERRAE software development; c) Static and DPR-enabled TERRAE testing.	Initial DPR-enabled TERRAE platform development and testing.
OS	Windows XP 32-bit VM on Windows 7 Pro	Ubuntu 10.04 32-bit (2.6.32-25-generic kernel) virtual machine	Windows 7 Pro SP1 64-bit
CPUs	Intel Core i5 680 dual core @3.6GHz, 4MB cache, 4 threads	Intel Core i7 L620, 64-bit, dual 2GHz, 4MB cache, 4 threads	
RAM	VM: 3GB Total: 8GB	VM: 1.5GB (scaled up to 3GB during synthesis runs) Total: 8GB	
Target	Xilinx ML505 development board	Xilinx ML505 development board	
Tools	Windows XP VM: Xilinx ISE, 9.2.04, Xilinx EDK 9.2.02, PR patch v14, PlanAhead 10.1	Xilinx ISE 9.2.04, Xilinx EDK 9.2.02	PlanAhead 10.1, ISE 10.1, PR patch v14

The description of the development flow is captured in Figure 48. First, a dual bus asymmetric multiprocessing system hardware with two MicroBlaze processors is developed in EDK on Linux workstation. It is integrated with RTL of Hardware Administrator (HA) IP and PRMs that are re-used from the Intraframework. The resulting system is the hardware platform of TERRAE, which brings LINUX_CPU and HA_CPU MicroBlaze processors for running software.

Several tests are performed at this point. LINUX_CPU's access to DDR2 and on-chip block RAMs are tested with TestApp_Memory application. Also, interrupt controllers and communication between the processors via mailbox are checked using TestApp_MBox_Int, a modified version of Xilinx dual processor reference application [93]. For both applications refer to Section 4.6.3. At this point, the operation of HCApp can also be checked by downloading it to the HA_CPU, and manually writing 32-bit control words into mailbox's PLB0 interface with a debugger. Finally, the verified TERRAE hardware is labeled as "golden" and frozen to prevent any changes to peripheral addresses and consequent kernel regenerations.

Next the embedded Linux development flow for LINUX_CPU is started, as shown on the left side of Figure 48. First, the PetaLinux kernel is configured and built for the TERRAE hardware with the help of PetaLinux toolchain, which produces linux.bin kernel image file without a file system. Then, applications and drivers are developed, compiled and installed into romfs. The latter is appended to the kernel image resulting in final OS binary image.bin.

In parallel, a board support package (BSP) is generated for HA_CPU and the HCApp is built. Next, the system integration is validated with Linux and all TERRAE agent applications. One exception is that partial reconfiguration API is replaced with an empty stub to avoid locking up the system in the absence of DPR. At this point the static TERRAE EDK hardware project is ready to be taken through the PlanAhead PR Flow.

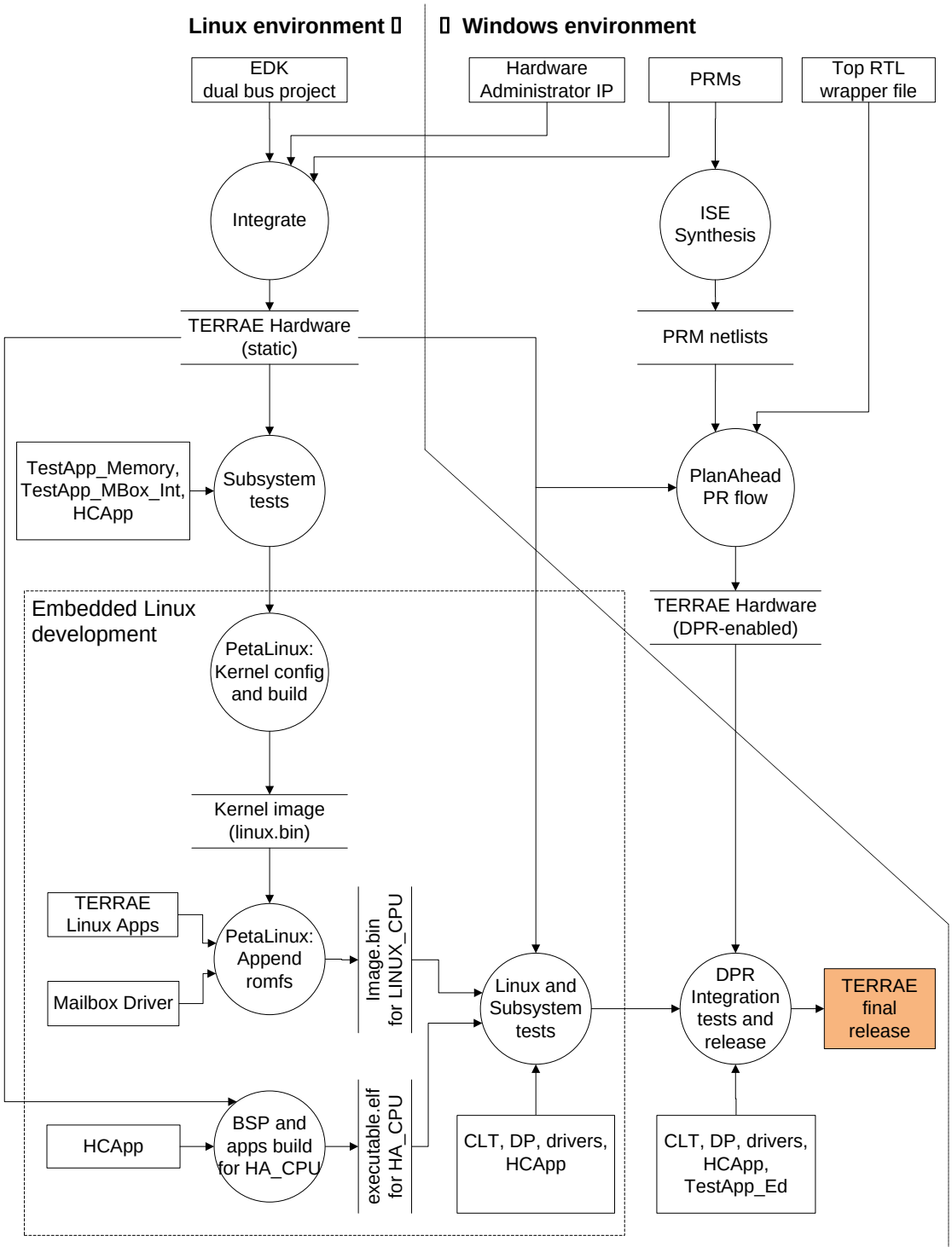


Figure 48: TERRAE development flow, data flow diagram

In the Windows environment, the TERRAE EDK project is wrapped in a shell with the help of Xilinx ISE tool, as shown on the right side of Figure 48. Here the boundaries for each PRR are defined. Then, as suggested by the Early Access PR flow [88], the EDK's digital clock manager (DCM) block is moved to the top level wrapper. Because in EDK the DCMs are automatically instantiated based on the selected configuration of clock_generator IP, when moving the clock management to the top level, two DCM blocks have to be manually chained together in order to obtain the desired frequencies and phases (see system clock diagram in Figure 37). Furthermore, for each port that links static and dynamic parts, the bus macros are inserted. Also, the locations for clock buffers are specified. Finally, the pre-synthesized netlists of each PRM are assigned to their respective PRRs and the whole project is taken through synthesis, mapping, place and route, and bitstream generation. The resulting static and partial bitstreams are first validated with TestApp_Ed, and then packaged together for the final TERRAE integration testing.

The final TERRAE test involves the DPR-enabled TERRAE, Linux kernel with applications, and HCApp running through real reconfiguration scenarios. Please refer to the created Quick Start User Guide [94]. Again, due to the IP address ranges having been frozen in the "golden" version of TERRAE's EDK project, any application or kernel can be modified and fine-tuned without changing the hardware.

4.6.2. Build Tools for Agent Applications and Drivers

This section describes command line Bash shell scripts and makefiles created for assisting the building process of the following software agent applications and drivers: CLT, DP, HCApp, and mailbox driver [89].

4.6.2.1. Command Line Tool Scripts

TERRAE does not use shared library for *libpr*. Instead, each application is compiled separately selecting the required library functions. This is not ideal as it generates larger executables, but allows for implementation simplicity. For example, with this approach before compilation it is possible to create a symbolic link to libpr/os/pr_sys.h depending on a target processor. For Linux applications such as CLT and DP, do_make.cmd links libpr/os/ to libpr/os-linux/ before running Makefile

compilation. For applications on HA_CPU such as HCApp, the TERRAE scripts link libpr/os/ to libpr/os-standalone/ instead. This allows for writing *libpr* such that it only has to include header files from libpr/os/ folder and at the same time providing specific APIs for a target processor.

The CLT's scripts are summarized in Table 40. Do_make.cmd does the linking of libpr/os/ to libpr/os-linux/. Then it builds the application by calling a default target in Makefile_Petalinux. Finally, it installs the compiled executable to Linux romfs at ../../petalinux-dist/romfs/bin/prctl (with respect to $PETALINUX/software/user-apps/prctl/ application development path).

Do_make_pc.cmd and Makefile_PC can be used for compiling the application for PC. These scripts were used during debugging and are not completely verified. Finally, running CLEAN removes object and executable files from the application directory. It can be optionally run before compilation.

Table 40: Command Line Tool scripts

File	Description
CLEAN	Removes object and executable files to cleanup directory
do_make.cmd	Customizes *libpr* by creating a symbolic link to the proper pr_sys.h, runs Makefile_Petalinux and installs the compiled application to Linux romfs.
do_make_pc.cmd	Customizes *libpr* by creating a symbolic link to the proper pr_sys.h and runs Makefile_PC.
Makefile_PC	Makefile that builds the necessary *libpr* files for PC. This file can be used for testing the code on PC.
Makefile_Petalinux	Makefile that builds the necessary *libpr* files. It has targets for library cleanup and romfs installation. This file was generated by PetaLinux toolchain and customized for TERRAE.

4.6.2.2. Daemon Process Scripts

The DP has the similar scripts as the CLT agent. An additionally listed do_make_pr_config_sharedlib.cmd was used for debugging of shared library compilation flow. However, it was later superseded by do_make.cmd. It was not fully tested and left there as a reference.

Table 41: Daemon Process scripts

File	Description
CLEAN	Removes object and executable files to cleanup directory
do_make.cmd	Customizes *libpr* by creating a symbolic link to the proper pr_sys.h, runs Makefile_Petalinux and installs the compiled application to Linux romfs.
do_make_pc.cmd	Customizes *libpr* by creating a symbolic link to the proper pr_sys.h and runs Makefile_PC.
do_make_pr_config_sharedlib.cmd	Customizes *libpr* by creating a symbolic link to the proper pr_sys.h and compiles it into a shared library. Then compiles DP, links it to the library and installs both to Linux romfs. This function is used for shared library testing only and has not been completely verified.
Makefile_PC	Makefile that builds the necessary *libpr* files for PC. This file can be used for testing the code on PC.
Makefile_Petalinux	Makefile that builds the necessary *libpr* files. It has targets for library cleanup and romfs installation. This file was generated by PetaLinux toolchain and customized for TERRAE.

4.6.2.3. HCApp Application Scripts

The HCApp's build scripts are summarized in Table 42. Do_make.cmd is used for trial compilation for standalone HA_CPU using system.make file found in Xilinx project hardware directory. Similar to CLT and DP's scripts, it starts off by re-linking libpr/os/ path. Then it calls system.make's pr_haif_program target to build the application for HA_CPU.

```
make -f system.make pr_haif_program
```

Table 42: HCApp scripts

File	Description
CLEAN	Removes object and executable files to cleanup directory
do_make.cmd	Customizes *libpr* by creating a symbolic link to the proper pr_sys.h and runs Makefile
Makefile	Makefile that builds the necessary *libpr* files

The developed flow uses do_make.cmd only for quick checking of compilation errors. Then as a follow-up step, the HCApp is compiled again by right-clicking on HCApp in the Applications tab of XPS and selecting Build Project from the drop-down menu. Finally, the hardware bitstream's RAMs are initialized with the executable code by launching Update Bitstream from Device Configuration menu. As a result, the final bitstream contains the HCApp, which it starts from system bootup.

4.6.2.4. Mailbox Driver Scripts (Linux)

The Linux mailbox driver contains the similar script files as other TERRAE applications. The directory also has do_release.cmd script used for automating the release process making the driver backward compatible with the original v1.00.

Table 43: Mailbox driver scripts (Linux)

File	Description
CLEAN	Removes object and executable files to cleanup directory
do_release.cmd	Driver release script
Makefile	Original makefile used by v1.00 of the driver; not recommended for PetaLinux
Makefile_petalinux	Makefile that builds the driver for PetaLinux use
README	Describes installation instructions and other useful information

4.6.3. Additional Applications for Subsystem Integration Testing

This section describes C applications that were developed or modified for testing the integration of TERRAE and Intraframework.

4.6.3.1. TestApp_Memory

The TestApp_Memory is an embedded application that can be automatically generated for the EDK project. It is a part of a standard Xilinx toolset, which were used for read-write access testing of the external DDR2 RAM.

4.6.3.2. TestApp_MBox_Int for Testing Inter-Processor Communication

The TestApp_MBox_Int is a modified version of Xilinx dual processor reference application [93]. The original code is targeting both processors with the help of the #ifdef

directive. It sends and receives ten messages between processors utilizing mailbox and interrupts. The code was customized during debugging for testing interrupts, blocking and non-blocking mailbox reads, heap and stack variability, byte endianness, and race conditions for messages from two processors printing to the same UART.

4.6.3.3. TestApp_Ed for Testing Intraframework

The TestApp_Ed is an application originally developed by E. Chan [14] for validating functionality of PRMs. It provides an interactive interface allowing reconfiguring PRMs, polling their status, and essentially stepping through PRMs' state machine. It was also frequently used during the bringup of DPR-enabled TERRAE.

4.6.4. TERRAE/Intraframework Release Flow Scripts

The final release of the TERRAE for public distribution is accomplished with the help of Bash scripts that were created to standardize and automate the process. The descriptions are provided in Table 44. Also, a release script for the mailbox Linux driver was implemented, as described in Section 4.6.2.4.

Table 44: TERRAE release scripts

File	Description
do_bitstream_check.cmd	Check and rename bitstreams for compatibility with TERRAE
do_create_release.cmd	Main release script
do_dir_struct_check.cmd	Checks and links PlanAhead project directories
setup_environment_vars.cmd	Loads environment variables

4.6.5. Additional EDK Platforms for Subsystem Integration Testing

One of the milestones towards building functional TERRAE hardware was a single bus static system. Among other IP, it included two processors, two interrupt controllers and a mailbox with both of its interfaces connected to the same bus. The resulting platform enabled incremental testing and resolution of both hardware and software. For instance, it helped to pinpoint a mailbox IP bug that assigned both of its interfaces to a single bus, even when it was connected to two (see Section 4.7.1). It

also allowed the elimination of the effect of a PLB2PLB bridge on both processors printing to the same UART.

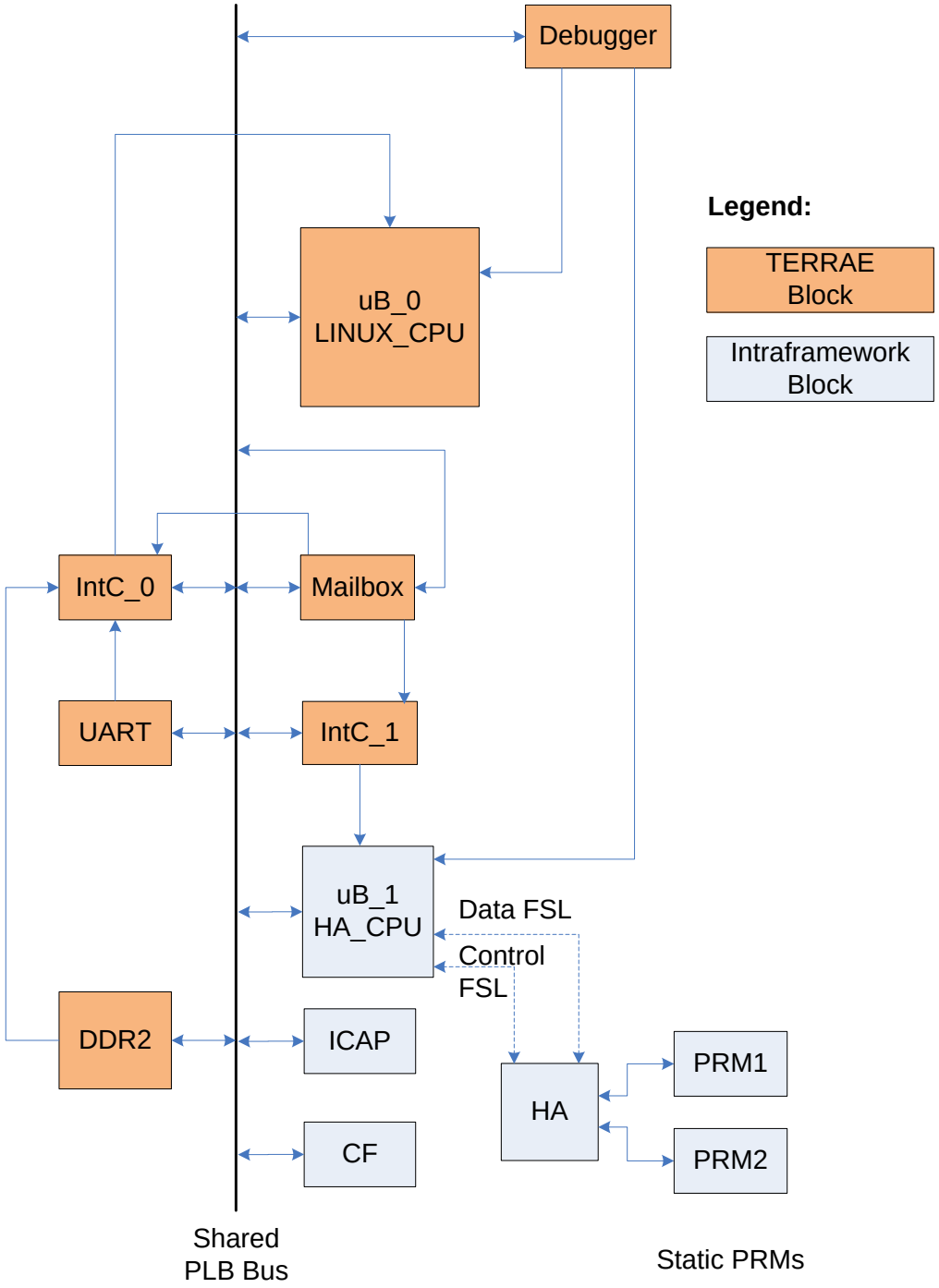


Figure 49: Shared bus EDK platform for IPC and mailbox testing

4.7. Issues and Workarounds

This section describes some of the interesting issues that were faced during the project implementation. These were largely architectural limitations set by Xilinx or bugs in the tool-chains. Also, solutions and workarounds that were developed to overcome these problems that would have hampered the successful implementation of this project are illustrated.

4.7.1. Dual Bus Mailbox Connection Bug Fix

Problem:

Mailbox is connected to 2 PLB busses that have one MicroBlaze processor on each bus. Mailbox is inaccessible from the 2nd PLB. Tools: EDK 9.2.04, and mbox_v1_00_a mailbox driver.

Fix:

It was found that the EDK mbox_v1_00_a driver generated address and bus interfaces referring to the 1st bus for both sides of the mailbox. The fix makes sure that MicroBlaze_0 can access Mailbox's SPLB0 address and Microblaze_1 can access Mailbox's SPLB1 address. The patch has been applied to the following driver's Tcl script $XILINX_EDK/sw/XilinxProcessorIPLib/drivers/mbox_v1_00_a/data/mbox_v2_1_0.tcl, with the fix listed in Appendix A.

4.7.2. Adding BRAM Larger than 64K

Problem:

The BRAM allocated for larger than 64K is not accessible in EDK 9.2.

Workaround:

To work around this issue two LMB controllers covering a contiguous range were instantiated, as seen in Figure 50 and Figure 35.

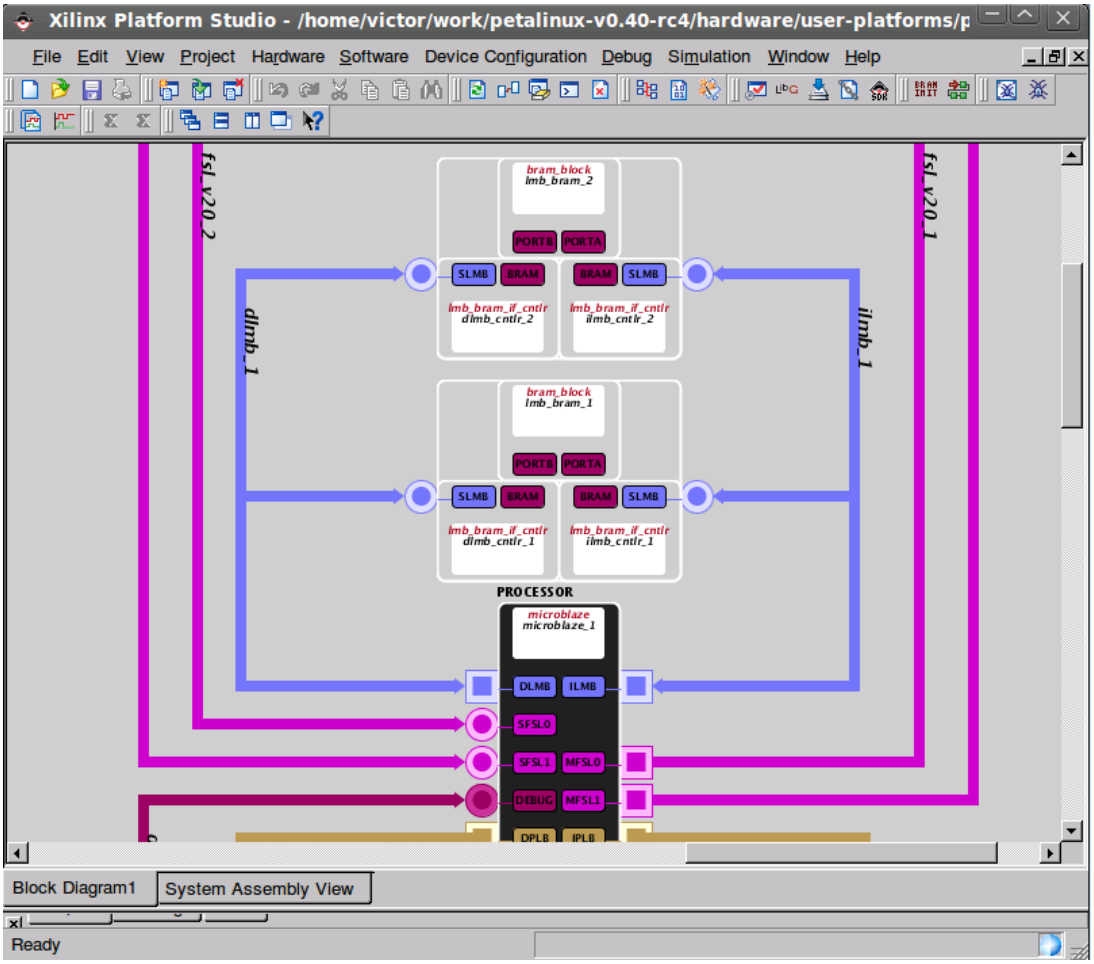


Figure 50: Using two LMB controllers for BRAM above 64K

4.7.3. Xilinx ISE/PlanAhead 10.1 Linux Problems and Mitigation

Problem:

During the NgdBuild step, the PlanAhead 10.1.8 Linux installation issues the following error and aborts: "invalid target" error: WARNING:NgdBuild:148 - Invalid target device "xc5vlx50t" in "-p" option value "xc5vlx50tff1136-1".

Workaround:

Virtex-5 LX50T device should be supported by this version of ISE/PlanAhead. So, the following steps were taken in attempt to resolve the issue. The supported device list and xml files were checked, such as $xilinx_path/10.1/PlanAhead/parts/xilinx/virtex5lx/DeviceParts.xml. Also, the project was attempted to be built for earlier devices,

with several installations and patch update sequences. Eventually, it was decided to move the addition of DPR support for TERRAE step to Windows environment, which did not cause any difficulties. Details are described in Section 4.6.1.

4.7.4. DDR2 RAM Operating Frequency

Problem:

DDR2 memory stopped passing the read-write access test after the move of clock managers to the top level during a PR flow.

Workaround:

Since clock_generator had to be removed from EDK, and instead DCM_ADV instantiated at the top level (see Section 4.3), originally only one DCM_ADV instance was created. However, its ports limited the design to generating only 200 MHz and 100 MHz with 90 degrees phase clocks, as the onboard oscillator produced 100 MHz. However, DDR2 controller supported 125 MHz, even though nowhere in its documentation was this listed as a requirement; moreover, it suggested the use of the default 100 MHz frequency for ML505. Except when the generation of a fresh EDK project with 100 MHz was attempted, a warning was raised. Thus, to circumvent this issue a second DCM_ADV instance was added and clocks were connected as shown in Figure 37.

4.7.5. IP Compatibility

Problem:

EDK 9.2.04 included newer IP than what was supported by PetaLinux v0.40 RC4. Moreover, same versions of Xilinx tool installations for Windows and Linux on different workstations did not behave consistency when faced with the same versions of IP.

Workaround:

The IP for TERRAE and Intraframework was downgraded to the supported versions on both workstations. Also, the PLB2OPB bridge had to be temporarily used

for adding a compatible OPB ICAP module. Eventually, clean reinstallations following the same update sequences, and using PetaLinux-compatible IP versions helped in resolving these problems.

4.7.6. Printing to JTAG UART

Problem:

Cannot print to JTAG UART on the 2nd PLB from HA_CPU. Following the dual processor application node suggestions [93] resulted in the code getting stuck at printf(), i.e. XUartLite_SendByte().

Workaround:

This issue has not been fully solved. To work around it, the UART on the 1st PLB was used for stdout from both processors. It is accessible from the second bus via the bridge.

4.7.7. Compact Flash Lockup during a Read

Problem:

Compact Flash locks up with ML505 board after reading several sectors during PRM reconfiguration.

Workaround:

This is not a Xilinx tool problem; however, it is listed here for the sake of completeness. Repeated tests pinpointed the issue to be likely due to the improper CF unmounting procedure. The following sequence proved to be working. After copying the files from a PC to CF via a card reader while in Linux VM, first unmount the card from the VM. As a result, Windows host automatically mounts it. So, unmount it from Windows as well. Finally, connect the card to the ML505 board.

4.7.8. USB JTAG Cable Lockup with Virtual Machine

Problem:

After downloading a bitstream via JTAG and forgetting to disconnect, starting an XMD debugger session locks the cable. The existing software cable unlocking functions do not work, requiring a physical cable reconnection.

XMD prints:

```
ERROR set configuration. strerr=Connection timed out.
```

iMPACT prints:

```
Checking cable driver.
...
write cmdbuffer failed 20000020.
write cmdbuffer failed 20000020.
Loopback test failed. ...
Cable connection failed.
```

Workaround:

Cable unlocking was attempted by running *cleancablelock* in IMPACT 9.2 and *xclean_cablelock* in XMD 9.2, which did not resolve the problem. As a workaround the following sequence has worked:

- Exit XMD and IMPACT;
- Disconnect the USB JTAG cable from the VM via menu: Virtual Machine -> Removable Devices -> XILINX cable -> Disconnect;
- Physically disconnect and reconnect the USB JTAG cable;
- Connect the USB JTAG cable to the VM via menu: Virtual Machine -> Removable Devices -> XILINX cable -> Connect;
- Start IMPACT and download the bitstream.

5. Analysis and Discussion of Results

This chapter begins by explaining TERRAE's operation with an example. Then it discusses the system utilization and performance. It continues by describing the desired improvements to the TERRAE architecture software, and Xilinx tools. Finally, this chapter is concluded by discussing the potential of TERRAE for the future adaptive hardware concurrent systems.

5.1. TERRAE Operation

The chapter begins with a brief illustration of how TERRAE is to be used in a typical operation, by showing the system starting up and reconfiguring one of the PRMs using TERRAE's API via a console.

First, FPGA is configured with a static bitstream. Next, the Linux kernel is downloaded into DDR2 RAM, which is accessible by the LINUX_CPU. Furthermore, the HCApp is downloaded into a Xilinx (BRAM) attached to HA_CPU. For simplicity, the Xilinx Microprocessor Debugger (XMD) was used and a script to automate the process was created. After completion of the script, the HCApp begins its operation by initializing hardware such as mailbox, interrupt controller and bus macros. Linux prompts to login. Both processors print to the same console since both can access the same UART, i.e., HA_CPU via the bridge.

In Linux, the mailbox driver is loaded by executing the following commands (Note: the '#' at the beginning of the lines is just the Linux OS prompt):

```
# mknod /tmp/mailbox_driver c 32 0
```

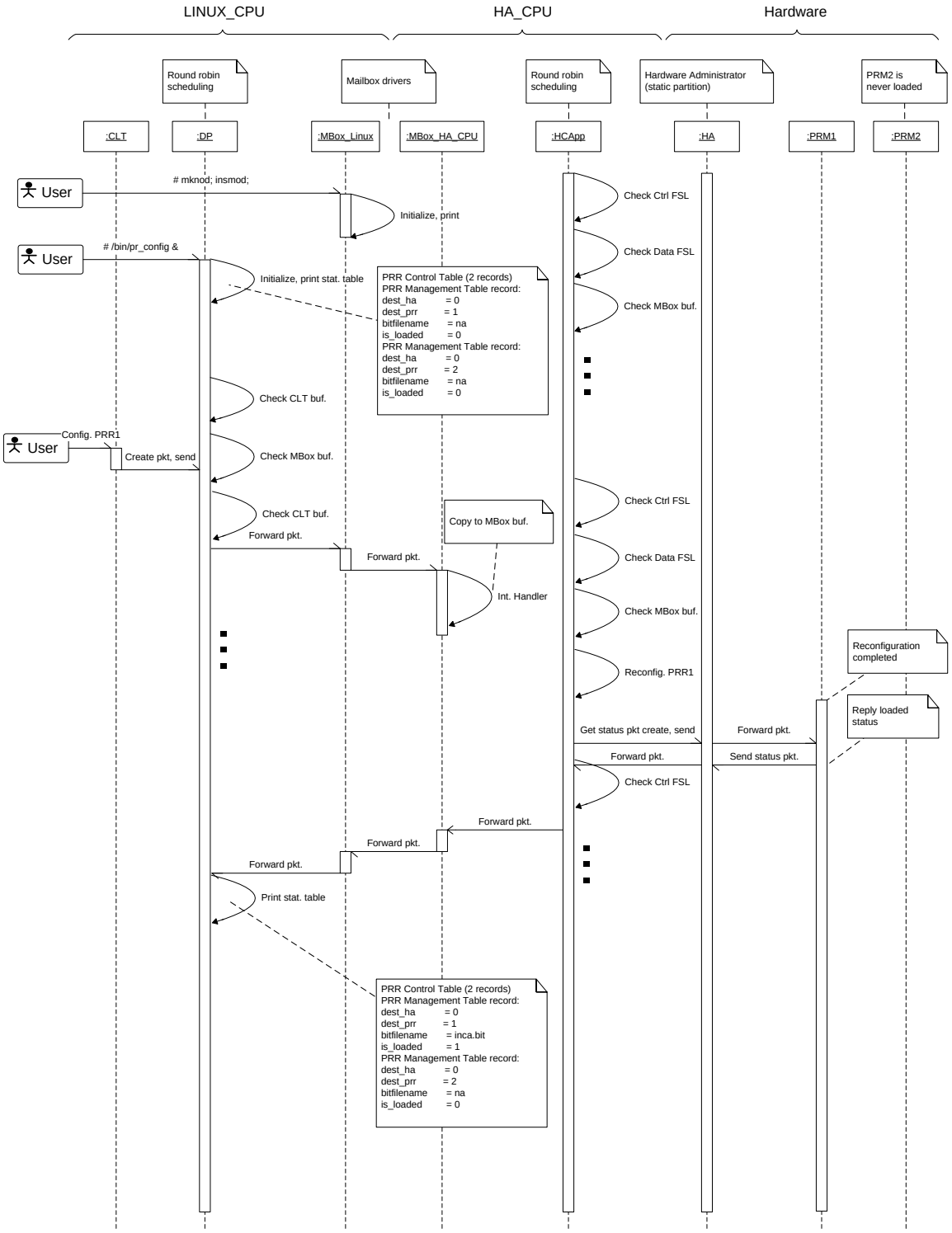


Figure 51: TERRAE operation UML sequence diagram

insmod /lib/modules/kernel/drivers/misc/mailbox_driver.koThe driver prints initialization messages, as seen Figure 51 UML sequence diagram:

```
Initializing interrupts..... Done
Allocating MBox tx/rx buffers..... Done
Registering IOs..... Done
Requesting IO ports.....
- Real address 0   : 0x71010000.... Done
Registering driver to kernel..... Done
Creating netlink socket..... Done
```

To load a daemon process, execute the following (Note: the '&' submits the command as a background task):

```
# /bin/pr_config &
```

It prints status table of available PRRs. Note that nothing has been loaded yet.

```
PRR Control Table (2 records)
PRR Management Table record:
dest_ha           = 0
dest_prr          = 1
bitfilename       = na
is_loaded         = 0
PRR Management Table record:
dest_ha           = 0
dest_prr          = 2
bitfilename       = na
is_loaded         = 0
```

Next download the inca.bit bitstream into PRR#1 at HA#0:

```
# /bin/prctl -bit inca.bit -dest 0 1
```

The above command creates a control command CMD_RECONFIG_PRR packet (12 bytes consisting of 4 byte header and 8 bytes filename) and sends it to the DP, which forwards it to HCApp via the Mailbox. HCApp recognizes the reconfiguration command by looking at its header with the help of one of libpr's functions and, instead of forwarding it to one of PRMs, executes a function to load the bitstream from CF to ICAP. After completion, the HCApp creates a control command CMD_PRR_SUMMARY packet and sends it to the PRR#1 via control FSL bus in order to check its status. The PRR replies back with a single word (i.e., 4 bytes) status packet, which is forwarded to DP via

HCApp and the Mailbox, which finally prints the status table. Note the change in status for PRR#1.

```
Reconfigure (HA, PRR) (0, 1) with inca.bit
PRR Control Table (2 records)
PRR Management Table record:
dest_ha           = 0
dest_prr          = 1
bitfilename       = inca.bit
is_loaded         = 1
PRR Management Table record:
dest_ha           = 0
dest_prr          = 2
bitfilename       = na
is_loaded         = 0
```

This serves as a simple validation of the system connectivity and reconfiguration process. The sequence of the described commands, along with the follow up responses, is illustrated in Figure 50.

5.2. TERRAE Utilization and Performance

The dual bus TERRAE implementation with 5-stage pipeline MicroBlaze processors, instruction accelerators, debugger module, two example counter PRMs, and several non-TERRAE peripherals is summarized in Table 46. It synthesizes to 11.8K registers and 12.7K LUTs, which results in 41% and 44% utilization of the Virtex-5 LX50T (see Table 45). However, these results include extra non-TERRAE IP. Also, the place-and-route (PAR) reports the maximum sys_clk frequency of 117 MHz.

If non-TERRAE blocks and PRMs are excluded, and a 3-stage pipeline is used instead, the utilization is 8.1K (28%) and 9.7K (33%) for flip-flop and LUT respectively, as reflected in Table 47 and Table 48. Similarly, PlanAhead reports 299 (1%) and 569 (1%) registers and LUTs for each counter PRM (see Table 49). The system runs at 125 MHz without reported timing violations.

Table 45: TERRAE hardware utilization on Virtex-5 LX50T

Resource	Count	Total	%
Slice	5,905	7,200	82
Register	11,818	28,800	41
LUT	12,863	28,800	44
BRAM	59	60	98
DSP48	7	48	14
DCM_ADV	2	12	16
Equivalent gate count			8,239,308

Table 46: EDK IP and options

IP	Options
microblaze_0	5-stage pipeline, barrel shifter, 64-bit int. multiplier, int. divider, MSR instructions, pattern comparator, 2xFSLs (1 unused)
microblaze_1	5-stage pipeline, 32-bit int. multiplier, MSR instructions, pattern comparator, 2xFSL
Debugger	2 ports
Extra IP	xps_mch_emc flash, util_bus_split, 2x xps_timer, PLB math IP
PRM IP	2x Counters

Table 47: TERRAE utilization on Virtex-5 LX50T, without PRMs

Resource	Count	Total	%
Slice	4,644	7,200	64
Register	8,127	28,800	28
LUT	9,672	28,800	33
BRAM	59	60	98
DSP48	6	48	12
DCM_ADV	2	12	16
Equivalent gate count			8,175,182

Table 48: TERRAE IP without PRMs

IP	Options
microblaze_0	3-stage pipeline, barrel shifter, 32-bit int. multiplier, 1xFSL
microblaze_1	3-stage pipeline, barrel shifter, 2xFSL
Debugger	None
Extra IP	None
PRM IP	None

Table 49: Counter PRM utilization

Resource	Count	Total	%
Slice	190	7,200	2
Register	299	28,800	1
LUT	569	28,800	1
Equivalent gate count			20,552

5.3. Desired TERRAE Architecture Improvements

This section discusses the desired improvements to the TERRAE architecture to make it more robust and adaptable.

5.3.1. Faster Reconfiguration and Internet Connectivity

Sections 3.2.2 and 3.2.4.2 discussed the reasons for choosing ICAP and HA_CPU as an internal reconfiguration solution for TERRAE. This was done more for convenience than for optimization. The reconfiguration performance was not measured in this thesis, as there are number of papers that already analyze this. Some projects showed a speed up when using a custom reconfiguration controller IP and/or DMA engines [73]-[74], [91]. Utilizing such modules in TERRAE should decrease the reconfiguration times.

Other conveniences provided by HA_CPU were acting as a gateway between PLB and dual FSL links connecting to HA, and also as a decision making about reconfiguration. If both of these features are moved to hardware, there is no need for the HA_CPU. In fact, it would remove software overhead and, thus, the bottleneck created by this processor.

Moreover, if the custom controller can be directly connected to the CF such that the reconfiguration bandwidth does not affect the PLB1, it would remove the original reason for having the secondary bus. Therefore, the architecture can be further optimized by using a single system bus.

Also, even though the internet connectivity was listed as a requirement for TERRAE, this feature was not addressed in this thesis. Nevertheless, it is an intuitive step that would unleash a number of benefits that come with Linux—remote system upgrades, maintenance and operation of standalone and distributed control systems such as security, fire alarm, and robotics for a number of industries from automotive and home automation to industrial and aerospace.

5.3.2. Linux Processor Alternative

Depending on system requirements to boost the performance of embedded Linux, it can be moved to a hard processor IP available with selected device families. See Section 3.2.4.1 for a list of processors.

5.3.3. Dynamic Partially Reconfigurable Architecture

This thesis has examined hardware concurrent architecture with a high-level administrative OS layer that promotes concurrency. The resulting system is able to adapt to its external environment by means of interchangeable hardware modules and reconfigurable interfaces. The next step would be to enable partial reconfiguration of the architecture itself—that is, to provide the necessary tools and connectivity framework for rapid development, and allow switching between multiple architectures at runtime without affecting the modules within the same and other subsystems.

Why is this useful? There are many cases where changing connectivity between modules can be beneficial. Rather than using a static and often space-inefficient crossover switch, bus, or network-on-a-chip (NoC), the DPR allows for modification of architecture “on-the-fly”, such as changing switch types, splitting busses for independent operation, balancing bandwidth, and switching protocol versions. This useful feature would provide an intelligence behaviour that allows the system to reconfigure its architecture as the environment changes; thus, enabling greater system efficiency and flexibility.

As an example, refer to Figure 52 of a large AHCS in which the six PRRs on the top left side of the system are identical to the bottom six. Their connectivity can be changed “on-the-fly”. Assume from the start both groups have the same dual bus architecture as for the top left group. However, at a later time the bottom group has been swapped by a bus and a pipeline without affecting the top left group. Similarly, the three PRRs on the right side of the figure initially have three direct links each, as seen from the top and bottom right PRRs. Later the middle PRR has been switched to a single fat pipe at runtime. Note that the PRRs with reconfigurable architecture have an irregular shape which may not be ideal for routing; they are drawn this way for visualization purposes.

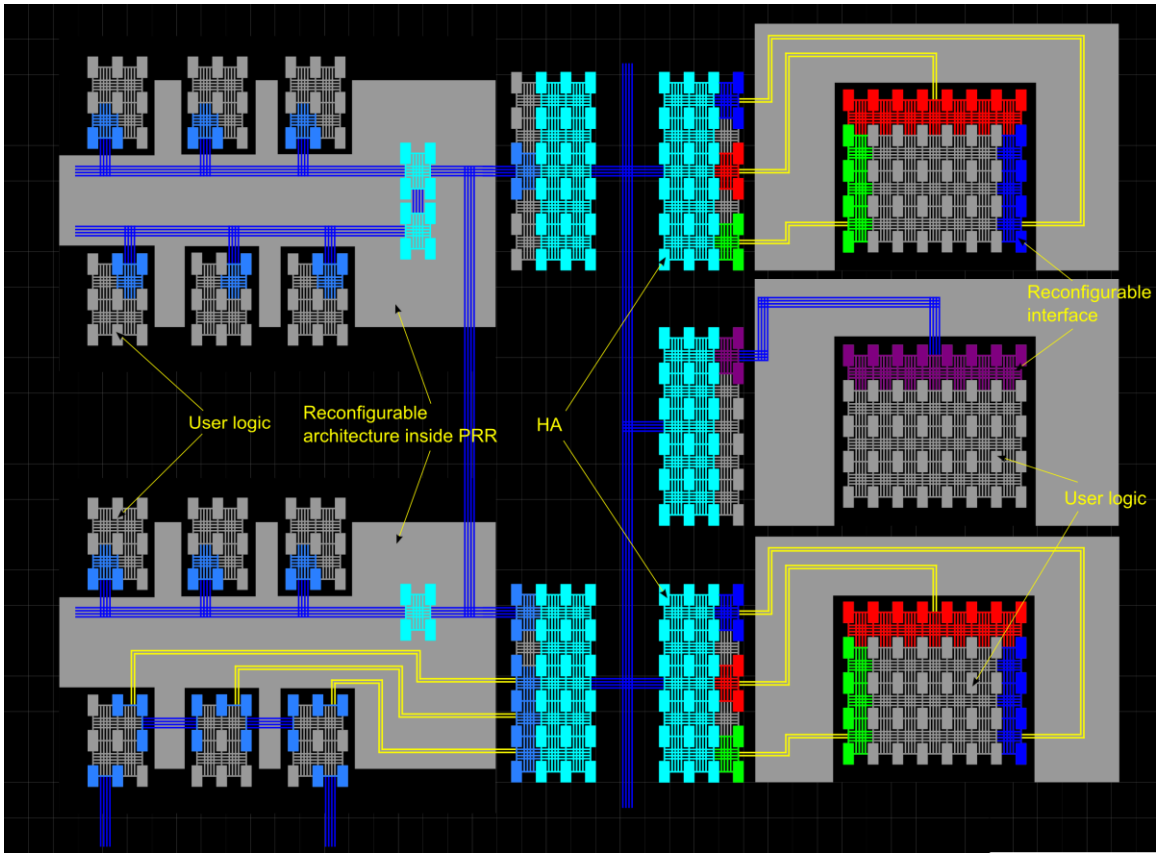


Figure 52: Dynamic partially reconfigurable architecture

5.4. Desired TERRAE Software Improvements

This section will discuss desired improvements to TERRAE agents and drivers.

5.4.1. Bursting Multiple Packets

As previously mentioned, due to a conscious decision for implementation simplicity of the first TERRAE version, the current limitation is that only one packet can be transmitted at a time between PRMs and OS. Special care must be taken to ensure that it is processed before another packet is sent to avoid overwriting the buffers. To overcome this shortcoming, it is important to look at its root cause and complexity. The Intraframework uses sideband signaling to imply a start of packet (SOP) based on a handshake between PRM and HA, after which a single packet is pushed into the FSL

FIFO. The Intraframework itself is limited to one packet in the FSL. If packet bursting is to be supported, there is a need for additional information to be used for packet delineation, either via a special inband header pattern or sideband signals.

Referring to Figure 20 in Section 3.3.3.4, TERRAE performs the FSL-to-PLB translation in software on HA_CPU. Because of the hardware/software boundary, it is advantageous to pre-buffer multiple packets in the FSLs. In this case the delineation can be done in hardware before sending the packets into the FSL FIFOs, possibly in the HA itself. Alternatively, if the HA_CPU is removed completely, as discussed in Section 5.3.1, the functionality can also be concentrated in a separate module between FSLs and the mailbox.

Now with packet boundaries defined and bursts of packets arriving to Linux, care must be taken with flow control and protecting the receive buffers. First, HA and hardware blocks should watch FIFO fill levels and not push more data than can be absorbed. Second, the Linux agent applications must use resource buffer locking techniques. Third, for both hardware and software buffers at each end, it would be convenient to allow partial packets into the FIFO for better packing. This feature would require residue buffering for reconstructing the received packets. The above mentioned techniques can be used in both directions between Linux and PRM.

5.4.2. Template Based Design and Text File Configuration

It would be beneficial for system developers to have a library of common base architectures as a starting point for their designs. Also, the framework should be able to be easily extended for supporting them. As it stands today, TERRAE maintains this information in a set of header files; however, it would benefit from a text file-based configuration, so that there would be no need for recompilation after the addition of a new architecture. The configuration files can include the list of all HAs and PRRs in the system, their properties, supported commands and command IDs. Some aspects of this work has already been started, as seen from the “-clt” switch to the CLT agent in Table 2 and Figure 22 for command lookup by name. However, the completion of this work has been left to future TERRAE updates.

5.5. Desired Tool and Platform Improvements

This section goes over some of the deficiencies that were faced with Xilinx tools during the TERRAE's implementation, and proposes suggestions for their improvement.

5.5.1. Integrated DPR Flow

The TERRAE was developed with the help of ISE 9.x, EDK 9.x, and PlanAhead 10.x, as mentioned in Table 39. The Early Access PR flow [88] suggests first developing the system EDK, followed by a creation of a top level HDL wrapper in text editor, and moving some IP there, such as the clock manager. Once the EDK project has been integrated, it essentially becomes a part of the ISE project and managed as such. Moreover, if the design becomes a candidate for DPR late in the project once the source versioning and directory structure have already been specified, there would be a need for restructuring. This is cumbersome and should be automated instead. Also, the ISE integration should ideally be avoided. It was mentioned that the latest partition-based tool flow has already tackled these deficiencies [23].

5.5.2. Back-to-Back Partially Reconfigurable Regions/Partitions

In order to support a partially reconfigurable architecture (see Sections 2.4 and 5.3.3) and modules, there is a need for seamless interface boundary without a static area between a PRM inside a PRR and the connecting architecture that is also inside another PRR. For instance, when an interface of the PRM is changed to support a different bus protocol, the number of ports would also likely change. The architecture would also be updated to accommodate for this change. If there is a static region connecting the architecture and PRR, it must account for all port variations across all PRMs targeting this PRR; thus, it is undesirable. This thesis has not examined the level of tool and device support; however, it notes it as a desired feature for continuing the future development of adaptive hardware concurrent systems.

5.6. Thesis Colophon

This section summarizes the work produced in this thesis. Table 50 lists the published papers, posters, technical reports and source code. Table 51 outlines the utilized hardware, software, and the developed features, methodologies and skills that were required for the successful completion of the thesis.

Table 50: Published materials leading to this thesis

Type	Published material	Description
Paper	[13] E. Chen, V.G. Lesau, D. Sabaz, L. Shannon and W.A. Gruver, "FPGA Framework for Agent Systems Using Dynamic Partial Reconfiguration," *Proc. Conf. on Industrial Applications of Holonic and Multi-Agent Syst.* (HoloMAS), 2011, pp. 94-102.	The definition and implementation of Intraframework; component-based design for hardware; hardware agent concept for DPR-FPGA-based designs.
Paper	[75] V.G. Lesau, E. Chen, D. Sabaz and W.A. Gruver, "Embedded Linux for Concurrent Dynamic Partially Reconfigurable FPGA Systems", *Proc. Conf. on NASA/ESA Adaptive Hardware and Systems* (AHS), 2012, pp. 99-106.	The definition and implementation of TERRAE and adaptive hardware concurrent control system; dual bus architecture for decoupling concurrent hardware from OS; embedded Linux kernel and user space modifications; API for command line communication with PRMs.
Poster	[57] V.G. Lesau, W.A. Gruver and D. Sabaz, "Embedded Linux for Hot Swapping Partially Reconfigurable FPGA Cores," poster presented at SFU Exchange 2010, Vancouver, May 6, 2010.	The initial introduction of the dual bus architecture and embedded Linux for TERRAE.
Tech. Report	[92] V.G. Lesau, *Experience Running PetaLinux on Xilinx ML505 Development Board with a Split Windows/Linux Environment*, tech. rep., Oct. 19, 2009.	The report summarizing the necessary steps for: installing tools, creating hardware, and modifying and building PetaLinux kernel for Xilinx ML505 board.
User Guide	[94] V.G. Lesau, *TERRAE: A Framework for Adaptive Hardware Concurrent Systems User Guide*, user guide (v0.11), Aug. 15, 2012.	Quick start user guide for booting TERRAE given the DPR-FPGA bitstream and Linux kernel image.
Source Code	[84] C. Foucher and V.G. Lesau. (2012, Feb. 9). *Open source Xilinx mailbox Linux drivers* [Online]. Available: http://sourceforge.net/projects/mboxlinux	The open source project for Xilinx Mailbox IP driver; adding support for interrupt handling and Netlink communication between the kernel and user spaces.
Source Code	[89] V.G. Lesau. (2012, Jan. 27). *Partially Reconfigurable Hardware Project* [Online]. Available: http://sourceforge.net/projects/prhardware	TERRAE open source project including hardware and software.

Table 51: Developed features, methodologies and required skills

Thesis Component	Developed Features, Methodologies and Required Skills	Tools and Languages
TERRAE hardware	Architecture; Component-Based Design for hardware agents; FPGA resource allocation; IP development; processor debugging.	Tools: Xilinx ISE, EDK, PlanAhead, IMPACT, XMD, and Early Access DPR patch. Languages: VHDL. Dev.Board: Xilinx ML505 development board.
Evaluation Hardware	Porting PetaLinux to XUP V2Pro; evaluation of embedded Linux and external reconfiguration.	Tools: PetaLogix PetaLinux and embedded Debian Linux. Dev.Board: Digilent XUP V2Pro; Technologic Systems TS-7300 with Cyclone II FPGA and ARM9 external CPU [95].
Embedded Linux	Development and debugging of Linux kernel drivers, Netlink communication, user space applications, BSP, and romfs.	Tools: PetaLogix PetaLinux toolchain, Emacs, VIM, gedit, hexdump, objdump. Languages: C, Bash, Tcl, Makefile.
Embedded Software	Development of BSP, drivers, and software agent applications.	Tools: Xilinx EDK, XMD. Languages: C, Bash, Tcl, Makefile.
Publishing	UML; data flow graphs.	Tools: Inkscape, InkSeine, UML, MS Office, Visio, Libra Office.

5.7. Conclusion

5.7.1. Synopsys

The research described in this thesis illustrates that DPR-FPGAs provide a path for systems engineers towards implementing adaptive and fully concurrent systems. By utilizing software paradigms as componentization of hardware bitstreams behind hardware interfaces, which can be done with DPR-FPGAs, the computer architecture can be separated from user logic. This enables a hardware or software engineer to develop hardware logic for highly concurrent systems quite simply, thus reducing the design time and the hardware development time for FPGAs. It is also emphasized that an adaptive control system will benefit not only from partially reconfigurable modules and interfaces, but also the architecture.

Now, the opportunity exists to develop tools that bring the same benefits of threaded soft architectures to concurrent hard architectures. In general, FPGA-based control systems have architectures designed for a specific problem domain. However, this introduces reengineering and development complexities. Solutions such as that described by Michel, et al. [72] lack greater flexibility and control by not decoupling the control and data interfaces from user logic. These interfaces also allow for simpler and more effective integration with higher level systems. Also, embedding an OS (Linux) that utilizes these interfaces allows our architecture to more effectively cooperate with the external environment. Furthermore, with the help of a two-bus architecture the processor with the OS is decoupled from the processor responsible for low-level reconfiguration-supporting tasks. The entire system is also integrated on a single Virtex-5 FPGA. The result is a flexible solution with more high-level functionality integrated in the same system-on-chip.

The current implementation promotes scalability by allowing for the addition of more PRRs without impeding the operation of OS. This thesis introduced a system with HA connected to two PRRs; however, it can be expanded to manage more ports. When all ports have been used, extra HAs have to be added. MicroBlaze 7.00.a supports up to 16 FSL duplex channels. With each HA using two FSLs, additional MicroBlazes are required for every 8 HAs. The dual bus architecture keeps communication between two PRMs that are connected to different HA-MicroBlaze pairs local to the DPR subsystem bus, if such communication is required. As a result, increasing the number of PRRs, and changing their connecting architecture within the DPR subsystem, does not burden the OS, thereby encouraging system scalability.

5.7.2. Looking Forward: Future of Adaptive Hardware Concurrent Systems and Migration of Operating Systems into Hardware

As discussed in this thesis, thread architectures impose limitations on control systems due to the non-determinism associated with threads (see Section 2.8). Thus, it can be concluded that control systems that utilize an OS, such as Linux, will benefit from the migration of OS to hardware. Rather than Linux being “accelerated”, which emphasizes dependency of hardware upon non-deterministic software, it can instead be run truly concurrently without impeding the system.

Moving OS to hardware would mean dealing with a number of architectural issues. First, a thread scheduler would need to be eliminated. In particular, instead of dealing with threaded architectures that manage time slicing of software code, the hardware OS would have to incorporate a spatial IP coordinator for bitstream placement on the reconfigurable fabric. Second, the processor-based architecture of OS would also change based on the fact that in hardware, the virtual memory and paging are no longer relevant. Third, the classical CPU-dependent computer architecture with a bus to memory can now be replaced with structures that support greater concurrency.

The above challenges can now be accommodated more effectively using the techniques and services demonstrated in this thesis. Virtual memory, paging, protected mode programming, thread scheduling and the supporting sub-systems to these OS services are no longer relevant in a DPR-FPGA environment. Instead, hardware modules can be wrapped by hardware interfaces and operate independently, while residing on top of a reconfigurable computer architecture. Furthermore, this architecture could entail multiple busses and bridges as necessary for the operating environment, without the restrictions imposed by a CPU. Finally, an administrative system, such as TERRAE, which could also be migrated to hardware, would provide a management layer. In summary, with the work described in this thesis, it is now possible to overcome difficulties associated with the “hardening” of OS and building complex adaptive control systems.

As a final thought on the Hard OS possibility, there arises a question: what kind of an FPGA would be required for accomplishing its implementation? Linux will be used as an example for making a rough estimate. As it stands today, it exceeds 15 million lines of code (v3.2); however, only 9.7 million is ANSI C [41], [96]. Also, 1.6 million of it is processor architecture-specific. Furthermore, at least 5.6 million lines are dedicated to drivers and processor architecture, which are ignored for this estimate because they can be loaded on demand. Moreover, a number of subsystems are used for thread management, such as virtual memory, synchronization, and scheduler (see the highlighted modules in Figure 53). As previously mentioned, such functionality will no longer be needed in the concurrent fabric; instead, it will be replaced by spatial managers, which too could be realized in hardware. It is roughly assumed the thread

management subsystems take another one million lines. With this we are left with around 1.5 million lines of optimized kernel code.

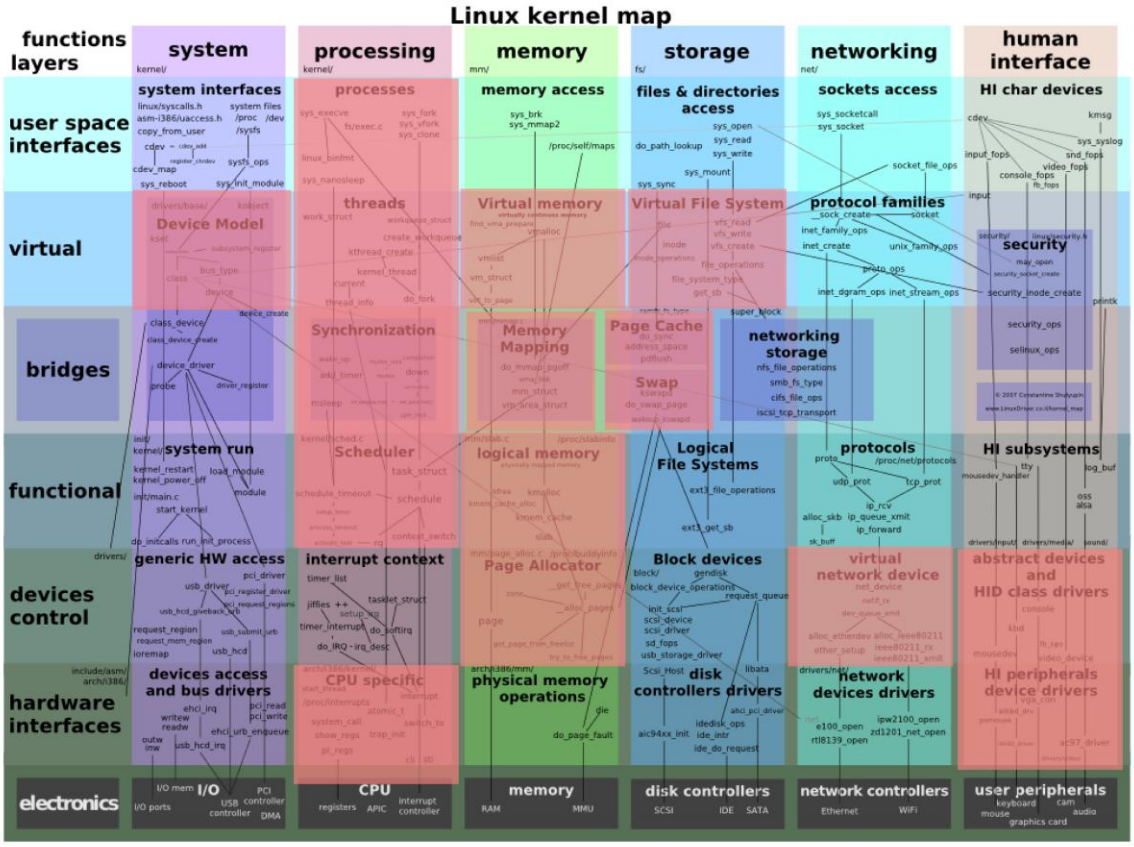


Figure 53: Linux kernel map highlighting thread management subsystems [42]

For the purpose of roughly estimating the conversion between lines of code and FPGA resources, several examples written in Handel-C high-level programming language will be used. Based on case studies with Handel-C for Canny Edge Detection [97], JPEG2000 [98], and integer-based MIPS [99], the ratio between the LUT and register counts, and the number of lines of Handel-C code, was around 1x, 4x, and 7x respectively. It is understood that FPGA utilization is highly dependent upon architectural features of the project. Thus, taking the larger number, and without considering resource efficiencies of the newer devices, the amount of reconfigurable resources required for implementing the Linux kernel is approximated to be at 10 million registers and LUTs.

As it stands today, the largest Virtex-7 V2000T 28 nm device contains nearly 300K slices with 4 flip-flops and 8 LUTs per slice [100], which gives around 1.2 million registers and 2.4 million LUTs. This illustrates that the today's most advanced DPR-FPGAs are an order of magnitude smaller than needed for migration of the optimized Linux kernel into hardware (i.e., a factor of 10). However, with the 50% reduction in circuit size of every smaller technology node, it becomes evident that the today's technology node would need to reach approximately 7 nm to accomplish the discussed hardware features. It has been indicated that the 7 nm node will become feasible within the next 10 years [101]-[102]. Therefore, given the expected advances in lithography, it would become possible to load a hard form of today's Linux or similar OS on a single FPGA device for concurrent management of control systems.

The advent of DPR-FPGA technology opens new possibilities for the development of intelligent control systems. With the help of methodologies demonstrated in this thesis, the flexibility of software can now be brought into hardware. Also, a hardware implementation of the OS itself would improve the determinism and performance of the system. Furthermore, the expected increase in fabric density of reconfigurable devices in the near future makes them more attractive for the migration of the system's intelligence from software to the circuit level. Thus, now is the prime time to begin the development of technologies supporting hardware concurrency for adaptive hardware concurrent systems.

References

[1] D. Husak, “Network processors: a definition and comparison,” white paper (CP00WP201), C-Port Corp., 2000.

[2] N. Farrington, “Data center switch architecture in the age of merchant silicon,” *IEEE Symp. on High Performance Interconnects* (HOTI), Aug. 2009, pp. 93-102.

[3] “Enabling 100-Gbit OTN muxponder solutions on 28-nm FPGAs,” white paper (WP-01126-1.0), Altera Corp., Apr. 2010.

[4] G. Brebner, "Live in-service modification of optical network elements implemented with Xilinx FPGAs," *Nat. Fiber Optic Eng. Conf.*, OSA Technical Digest (CD-ROM) (Optical Society of America, 2011), paper NWC3.

[5] N.W. Bergmann, J.A. Williams, J. Han and Y. Chen, “A process model for hardware modules in reconfigurable system-on-chip,” *Proc. Dynamically Reconfigurable Syst. (DRS) Workshop, 19th Int. Conf. Architecture of Comput. Syst.* (ARCS), Mar. 2006, pp. 205-214.

[6] E. Lübbers and M. Platzner, “ReconOS: an RTOS supporting hard- and software threads,” *17th IEEE Int. Conf. on Field Programmable Logic and Applications* (FPL), Aug. 2007, pp. 441-446.

[7] D. Andrews, R. Sass and E. Anderson, “Achieving programming model abstractions for reconfigurable computing,” *IEEE Trans. on Very Large Scale Integration (VLSI) Syst.*, vol. 16, no. 1, pp. 34-44, 2008.

[8] E. Lübbers and M. Platzner, “ReconOS: multithreaded programming for reconfigurable computers,” *ACM Trans. Embed. Comput. Syst.* (TECS), vol. 9, no. 1, pp. 1-33, 2009.

[9] Ismail and L. Shannon, “FUSE: front-end user framework for OS abstraction of hardware accelerators,” *IEEE Symp. on Field Programmable Custom Computing Mach.* (FCCM), May 2011, pp. 170-177.

[10] H.K.-H. So and R.W. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," *Int. Conf. on Field Programmable Logic and Applications* (FPL), 2006, pp. 1-6.

[11] H.K.-H. So, "BORPH: An OS for FPGA-based reconfigurable computers," Ph.D. dissertation, Sch. Electrical Eng. and Comput. Sci., U. of California, Berkeley, CA, 2007.

[12] E.A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33-42, 2006.

[13] E. Chen, V.G. Lesau, D. Sabaz, L. Shannon and W.A. Gruver, “FPGA Framework for Agent Systems Using Dynamic Partial Reconfiguration,” *Proc. Conf. on Industrial Applications of Holonic and Multi-Agent Syst.* (HoloMAS), 2011, pp. 94-102.

[14] E. Chen, “High-Level Abstractions for FPGA-based Control Systems to Improve Usability and Reduce Design Time,” Ph.D. dissertation, Sch. Eng. Sci., Simon Fraser Univ., Vancouver, BC, 2011.

[15] D.W. Page, “Dynamic Data Re-Programmable PLA,” U.S. Patent 4524430, June 18, 1985.

[16] (2012, June 2). *FPGA Architecture for the Challenge* [Online]. Available: http://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html

[17] L. Sekanina, “Reconfigurable Hardware,” in *Evolvable Components: From Theory to Hardware Implementations*, Springer, 2004, ch. 2.3, pp. 13-14.

[18] *XC6200 FPGA Advance Product Specification*, product specification, Xilinx Inc., June 1996.

[19] *AT40K/AT40KLV Series FPGA*, datasheet (0896CS-FPGA), Atmel Corp., Apr. 2002.

[20] “Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs,” white paper (WP-01137-1.0), Altera Corp., June 2010.

[21] “Flexible Waveform Processing with the Xilinx Zynq-7000 Extensible Processing Platform,” white paper (WP404 v1.0.1), Xilinx Inc., Mar, 6, 2012.

[22] *Fast Configuration of PCI Express Technology through Partial Reconfiguration*, application note (XAPP883 v1.0), Xilinx Inc., Nov. 2010.

[23] *Partial Reconfiguration User Guide*, UG702 (v14.1), Xilinx Inc., Apr. 24, 2012.

[24] *Xilinx Introduces Breakthrough Virtex-II Pro FPGAs to Enable New Era of Programmable System Design*, Xilinx Press Release #0204, Xilinx Inc., Mar. 4, 2002.

[25] *Stratix V Device Family Overview*, datasheet (SV51001-2.3), Altera Corp., Feb. 2012.

[26] (2012, Aug. 25). *Zynq-7000 Extensible Processing Platform* [Online]. Available: http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm

[27] (2011, Dec. 8). *Xilinx Ships First Zynq-7000 Devices, the World's First Extensible Processing Platform* [Online]. Available: http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1637670

[28] *Early Access Partial Reconfiguration User Guide*, UG208 (v1.1), Xilinx Inc., Mar. 6, 2006.

[29] A. Gordon-Ross, C. Conger and A.D. George, “Design framework for partial run-time FPGA reconfiguration,” *17th IEEE Int. Conf. on Field Programmable Logic and Applications* (FPL), 2007, pp. 1-6.

[30] G.O. Ruiz, “Dynamic Partial Reconfiguration and Video Distribution in a Reconfigurable Device,” M.Sc. dissertation, Barco Medical Imaging Division, Univ. de Bourgogne, France, 2009.

[31] "Mind and Brain," in *How People Learn: Brain, Mind, Experience, and School: Expanded Edition*, The National Academies, 2000, ch. 5, pp. 114-128.

[32] (2012, Aug. 25). *Human Connectome Project* [Online]. Available: http://www.humanconnectomeproject.org

[33] W.R. Ashby, *Design for a Brain: the Origin of Adaptive Behavior*, John Wiley & Sons Inc., 1960.

[34] P. Koopman, "Design constraints on embedded real time control systems," *System Design and Network Architecture Conf.*, 1990, pp. 71-77.

[35] S. Baskiyar, N. Meghanathan, "A Survey of Contemporary Real-time Operating Systems," *Informatica*, vol. 29, no. 2, pp. 233-240, 2005.

[36] (2006, June 6). J. Turley. *Operating Systems on the Rise* [Online]. Available: http://www.eetimes.com/discussion/other/4025674/Operating-systems-on-the-rise

[37] (2007, Apr. 26). *Snapshot of the embedded Linux market—April, 2007* [Online]. Available: http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Snapshot-of-the-embedded-Linux-market-April-2007

[38] (2012, Aug. 25). *Linux for Devices* [Online]. Available: http://www.linuxfordevices.com

[39] (2012, Aug. 25). *TiVo Inc.* [Online]. Available: http://www.tivo.com

[40] V. Yu, "Choosing an Embedded Operating System," white paper, Moxa Inc., Aug. 18, 2005.

[41] J. Corbet, G. Kroah-Hartman and A. McPherson, "Linux Kernel Development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It," white paper, The Linux Foundation, Mar. 2012.

[42] (2012, Aug. 25). *Linux Kernel Map* [Online]. Available: http://www.makelinux.net/kernel_map

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

[44] A.S. Tanenbaum, J.N. Herder and H. Bos, "Can We Make Operating Systems Reliable and Secure?" *Computer*, vol. 39, no. 5, pp. 44-51, 2006.

[45] M. Hatle. (1999, Feb.). *History of Linux for the PowerPC* [Online]. Available: http://gate.crashing.org/doc/ppc/doc003.htm

[46] (2012, Aug. 25). *Debian for PowerPC* [Online]. Available: http://www.debian.org/ports/powerpc

[47] (2003, Aug. 25). *[microblaze-uclinux] [ANN] Microblaze uClinux demo released* [Online]. Available: http://itee.uq.edu.au/~listarch/microblaze-uclinux/archive/2003/08/msg00054.html

[48] (2009, June 9). *MicroBlaze CPU architecture merged into kernel.org* [Online]. Available: http://www.petalogix.com/news/microblaze-architecture-merged-into-kernel.org

[49] (2012, Aug. 25). [Online]. Available: http://www.petalogix.com

[50] (2009, June 22). *Re: Corrections for download URL link and website* [Online]. Available: http://permalink.gmane.org/gmane.linux.uclinux.microblaze/8997

[51] J. Williams and N. Bergmann. "Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip," *Proc. of Int. Conf. on Engineering of Reconfigurable Syst. and Algorithms* (ERSA), June 2004, pp. 163-169.

[52] (2011, Oct. 19). *Hardware-Accelerated Linux Router* [Online]. Available: http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/Guide

[53] "Accelerating High-Performance Computing with FPGAs," white paper (WP-01029-1.1), Altera Corp., Oct. 2007.

[54] "FPGA Acceleration in HPC: A Case Study in Financial Analytics," white paper (v1.0), XtremeData Inc., Nov. 2006.

[55] (2012, Aug. 25). *Embedded Linux/Microcontroller Project* [Online]. Available: http://www.uclinux.org/ports

[56] (2012, Aug. 25). *BlueCat Embedded Linux* [Online]. Available: http://www.lynuxworks.com

[57] V.G. Lesau, W.A. Gruver and D. Sabaz, "Embedded Linux for Hot Swapping Partially Reconfigurable FPGA Cores," poster presented at SFU Exchange 2010, Vancouver, May 6, 2010.

[58] (2012, Aug. 25). *Technology Center* [Online]. Available: http://www.altera.com/technology/tc-index.html

[59] *7 Series FPGAs Overview*, product specification (DS180, v1.11), Xilinx Inc., May, 2012.

[60] *Embedded Processor Block in Virtex-5 FPGAs*, UG200 (v1.8), Xilinx Inc., Feb. 24, 2010.

[61] (2012, Aug. 25). *Dual-Core ARM Cortex-A9 MPCore Processor* [Online]. Available: http://www.altera.com/devices/processor/arm/cortex-a9/m-arm-cortex-a9.html

[62] (2011, Oct. 11). *Altera Introduces SoC FPGAs: Integrating ARM Processor System and FPGA into 28-nm Single-Chip Solution* [Online]. Available: http://www.altera.com/corporate/news_room/releases/2011/products/nr-soc-fpga.html

[63] *Intel Atom Processor E6x5C Series. Product Preview Datasheet*, datasheet (324602-001US), Intel Corp., Dec. 2010.

[64] "Strategic Considerations for Emerging SoC FPGAs," white paper (WP-01157-1.0), Altera Corp., Feb. 2011.

[65] *MicroBlaze Processor Reference Guide*, UG081 (v12.0), Xilinx Inc., Mar. 1, 2011.

[66] *PicoBlaze 8-bit Embedded Microcontroller User Guide*, UG129 (v2.1), Xilinx Inc., June 22, 2011.

[67] (2012, Aug. 25). *Nios II Processor: The World's Most Versatile Embedded Processor* [Online]. Available: http://www.altera.com/devices/processor/nios2/ni2-index.html

[68] (2012, Aug. 25). *MP32 Processor Brings the MIPS Ecosystem to Custom Embedded Systems* [Online]. Available: http://www.altera.com/devices/processor/mips/mp32/proc-mp32.html

[69] (2012, Aug. 25). *OR1200 OpenRISC processor* [Online]: http://opencores.org/openrisc,or1200

[70] G.J. Brebner, "A virtual hardware operating system for the Xilinx XC6200," *Proc. 6th Int. Workshop on FPGAs* (FPL), 1996, pp. 327-336.

[71] G.J. Brebner, "The Swappable Logic Unit: A Paradigm for Virtual Hardware," *IEEE Symp. on Field Programmable Custom Computing Mach.* (FCCM), 1997, pp. 77-86.

[72] H. Michel, F. Bubenhagen, B. Fiethe, H. Michalik, B. Osterloh, W. Sullivan, A. Wishart, J. Ilstad and S.A. Habinc, "AMBA to SoCWire network on chip bridge as a backbone for a dynamic reconfigurable processing unit," Proc. NASA/ESA Conf. on Adaptive Hardware and Syst. (AHS), 2011, pp. 227-233.

[73] 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," *IEEE Int. Parallel and Distributed Process. Symp.* (IPDPS), 2007.

[74] J.C. Hoffman and M.S. Pattichis, "A High-Speed Dynamic Partial Reconfiguration Controller Using Direct Memory Access Through a Multiport Memory Controller and Overclocking with Active Feedback," *Int. J. Reconfig. Computing*, vol. 2011.

[75] V.G. Lesau, E. Chen, D. Sabaz and W.A. Gruver, "Embedded Linux for Concurrent Dynamic Partially Reconfigurable FPGA Systems", *Proc. Conf. on NASA/ESA Adaptive Hardware and Systems* (AHS), 2012, pp. 99-106.

[76] J. Lohn, G. Larchev and R. DeMara, "A Genetic Representation for Evolutionary Fault Recovery in Virtex FPGAs," *5th Int. Conf. on Evolvable Systems* (ICES), Mar. 2003.

[77] (2012, Aug. 25). [Online]. Available: http://www.gaisler.com

[78] (2012, Aug. 25). *µC/OS-II Ports. Micrium* [Online]. Available: http://micrium.com/page/downloads/ports/xilinx

[79] (2012, Aug. 25). *PrKERNELv4 (µITRON4.0)* [Online]. Available: http://www.esol.com/embedded/prkernelv4.html

[80] (2012, Aug. 25). *FreeRTOS. Xilinx MicroBlaze Port Demonstrated on a Spartan-6 FPGA* [Online]. Available: http://www.freertos.org/Free-RTOS-for-Xilinx-MicroBlaze-on-Spartan-6-FPGA.html

[81] *Using EDK to Run Xilkernel on a MicroBlaze Processor*, UG758 (v7.0), Xilinx Inc., Aug. 11, 2010.

[82] (2012, Aug. 25). *Mentor Nucleus* [Online]. Available: http://www.mentor.com/embedded-software/nucleus

[83] (2012, Aug. 25). *Express Logic ThreadX* [Online]. Available: http://rtos.com/products/threadx/MicroBlaze

[84] C. Foucher and V.G. Lesau. (2012, Feb. 9). *Open source Xilinx mailbox Linux drivers* [Online]. Available: http://sourceforge.net/projects/mboxlinux

[85] K.K.He, "Why and How to Use Netlink Socket," *Linux Journal*, no. 130, Feb. 2005.

[86] (2012, Aug. 25). [Online]. Available: http://www.ubuntu.com

[87] (2012, Aug. 25). *ISE Design Suite* [Online]. Available: http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm

[88] *Early Access Lab: PR Implementation Using PlanAhead*, Xilinx Inc., 2009.

[89] V.G. Lesau. (2012, Jan. 27). *Partially Reconfigurable Hardware Project* [Online]. Available: http://sourceforge.net/projects/prhardware

[90] *Virtex-5 FPGA User Guide,* UG190 (v5.4), Xilinx Inc., Mar. 16, 2012.

[91] L. Shaoshan, R.N. Pittman and A. Forin, *Energy Reduction with Run-Time Partial Reconfiguration*, tech. rep. (MSR-TR-2009-2017), Microsoft Research, Sep. 2009.

[92] V.G. Lesau, *Experience Running PetaLinux on Xilinx ML505 Development Board with a Split Windows/Linux Environment*, tech. rep., Oct. 19, 2009.

[93] *Dual Processor Reference Design Suite*, applicat. note (XAPP996 v1.3), Xilinx Inc., Oct. 6, 2008.

[94] V.G. Lesau, *TERRAE: A Framework for Adaptive Hardware Concurrent Systems User Guide*, user guide (v0.11), Aug. 15, 2012.

[95] *TS-7300, High-Security Linux FPGA Computer*, datasheet, Technologic Systems Inc., June 2009.

[96] T. Leemhuis, "Kernel Log: 15,000,000 lines, 3.0 promoted to long-term kernel." (2012 August). [Online]. Available: http://www.h-online.com/open/features/Kernel-Log-15-000-000-lines-of-code-3-0-promoted-to-long-term-kernel-1408062.html

[97] D.V. Rao, S. Patil, N.A. Babu and V. Muthukumar, "Implementation and Evaluation of Image Processing Algorithms on Reconfigurable Architecture using C-based Hardware Descriptive Languages," *Int. J. Theoretical and Appl. Comput. Sci.*, vol. 1, no. 1, pp. 9-34, 2006.

[98] J. Jussel and C. Sullivan, "Software-Compiled System Design: A Methodology for Field Programmable System-on-Chip Design," Celoxica Ltd., Apr. 15, 2003.

[99] T. Ramdas, L.-M. Ang and G. Egan, “FPGA Implementation of an Integer MIPS Processor in Handel-C and its Application to Human Face Detection,” *IEEE Region 10 Conf.* (TENCON), vol. 1, 2004, pp. 36-39.

[100] *Virtex-5 Family Overview*, product specification (DS100, v5.0), Xilinx Inc., Feb. 6, 2009.

[101] (2011). *The International Technology Roadmap for Semiconductors. Lithography. 2011 Edition* [Online]. Available: www.itrs.net/Links/2011ITRS/2011Chapters/2011Lithography.pdf

[102] M. Tyson. (2012, May 15). *Intel currently developing 14nm, aiming towards 5nm chips* [Online]. Available: http://hexus.net/tech/news/cpu/39381-intel-currently-developing-14nm-aiming-towards-5nm-chips

Appendices

Appendix A.

A Fix for Incorrect PLB Address Generation for Xilinx Mailbox IP driver (mbox_v1_00_a)

Please refer to Section 4.7.1.

Tool: EDK 9.2.04

Driver: mbox_v1_00_a

Fixed File: $XILINX_EDK/sw/XilinxProcessorIPLib/drivers/mbox_v1_00_a/data/mbox_v2_1_0.tcl

Bug description:

Mailbox is connected to 2 PLB busses that have 1 MicroBlaze processor on each bus. During the libgen's generate procedure the mbox_v1_00_a's Tcl script uses PLB0's address for both processors.

The fix makes sure that MicroBlaze 1 can access Mailbox's SPLB0 address and Microblaze 2 can access Mailbox's SPLB1 address.

Instructions:

For modifications to the driver configuration Tcl script please find comments that start with #VGL# in the fixed mbox_v1_00_a/data/mbox_v2_1_0.tcl

```
##############################################################
# Copyright (c) 2007 Xilinx, Inc. All Rights Reserved.
# You may copy and modify these files for your own internal use solely
with
# Xilinx programmable logic devices and  Xilinx EDK system or create IP
# modules solely for Xilinx programmable logic devices and Xilinx EDK
system.
# No rights are granted to distribute any files unless they are
distributed in
# Xilinx programmable logic devices.
##############################################################
#uses "xillib.tcl"

proc generate {drv_handle} {
    xdefine_include_file $drv_handle "xparameters.h" "XMBOX"
"NUM_INSTANCES"
    xdefine_mbox_config_files $drv_handle "xparameters.h" "xmbox_g.c"
"XMbox"

#VGL#-----------------------------------------------------------------
#VGL# Fix for /* Canonical definitions for peripheral XPS_MAILBOX_0 */
section
#VGL# of xparameters.h file.
#VGL#-----------------------------------------------------------------
#VGL# When this TCL file is called for the 2nd MicroBlaze, its
#VGL# xdefine_canonical_xpars function needs to be called with
#VGL# C_SPLB1_BASEADDR and C_SPLB1_HIGHADDR parameters instead of
#VGL# C_SPLB0_BASEADDR and C_SPLB0_HIGHADDR.
#VGL# This is done by checking whether mailbox's SPLB1 port is connected
```

```
#VGL# to the same bus as the 2nd MicroBlaze before calling
#VGL# xdefine_canonical_xpars.
#VGL# Similarly, the code checks whether mailbox's SPLB0 port is
connected
#VGL# to the same bus as the 1st MicroBlaze before calling
#VGL# xdefine_canonical_xpars.
#VGL#--------------------------------------------------------------------
#VGL# The line below is replaced with the reset of the section
#VGL#    xdefine_canonical_xpars $drv_handle "xparameters.h" "Mbox"
"DEVICE_ID" "C_SPLB0_BASEADDR" "C_SPLB0_HIGHADDR" "C_NUM_CHANNELS"

#VGL# Debug printing
#VGL#    set file_name "xparameters.h"

    set sw_proc_handle [xget_libgen_proc_handle]
    set hw_proc_handle [xget_handle $sw_proc_handle "IPINST"]

    set periphs [xget_sw_iplist_for_driver $drv_handle]
    foreach periph $periphs {
        # PLB0
        set plb [xget_hw_busif_value $periph "SPLB0"]
        set plb_buses [xget_proc_dplb_buses $hw_proc_handle]

#VGL# Debug Print
#VGL#    set file_handle [xopen_include_file $file_name]
#VGL#    puts $file_handle [format "PLB0: plb=%s, plb_buses=%s" $plb
$plb_buses]
#VGL#    close $file_handle

        set if0_isplb 0
        foreach plb_bus $plb_buses {
            if { $plb_bus == $plb } {
                set if0_isplb 1
            }
        }
        if { $if0_isplb == 1 } {
            xdefine_canonical_xpars $drv_handle "xparameters.h" "Mbox"
"DEVICE_ID" "C_SPLB0_BASEADDR" "C_SPLB0_HIGHADDR" "C_NUM_CHANNELS"
        }

        # PLB1
        set plb [xget_hw_busif_value $periph "SPLB1"]
        set plb_buses [xget_proc_dplb_buses $hw_proc_handle]

#VGL# Debug Print
#VGL#    set file_handle [xopen_include_file $file_name]
#VGL#    puts $file_handle [format "PLB1: plb=%s, plb_buses=%s" $plb
$plb_buses]
#VGL#    close $file_handle

        set if1_isplb 0
        foreach plb_bus $plb_buses {
            if { $plb_bus == $plb } {
                set if1_isplb 1
            }
```

```
        }
        if { $if1_isplb == 1 } {
            xdefine_canonical_xpars $drv_handle "xparameters.h" "Mbox"
"DEVICE_ID" "C_SPLB1_BASEADDR" "C_SPLB1_HIGHADDR" "C_NUM_CHANNELS"
        }
    }
#VGL# End of the Fix
#VGL#---------------------------------------------------------------------

}

#
# Create configuration C/H files as required by Xilinx drivers
#
proc xdefine_mbox_config_files {drv_handle hfile_name cfile_name
drv_string} {

…
(Note: Some code has been removed to save space)
…

        # Next check IF1
        set mbox_baseaddr    0
        set mbox_use_fsl     0
        set mbox_send_fsl    0
        set mbox_recv_fsl    0
        set if1_isfsl        0
        set if1_isplb        0

        set if1_isfsl [xget_value $periph "PARAMETER"
"C_INTERFACE_1_IS_FSL"]
        if { $if1_isfsl == 1 } {
            # -- FIXME This is incomplete
            # -- Ultimately need to set the variables mbox_send_fsl and
mbox_recv_fsl in this branch
            # Verify if this interface is connected to this processor
            set mbox_baseaddr    0
            set mbox_use_fsl     1
            set mbox_send_fsl    1
            set mbox_recv_fsl    1
            set if0_isfsl        1
            set if0_isplb        0

        } else {

            # Verify if this interface is connected to this processor
            ## Note:  CR 442125 If this is fixed the following code is
an easier way to verify connectivity
            ## set master_ifs [xget_hw_connected_busifs_handle
$mhs_handle "mb_plb_0" "MASTER"]
            #set plb [xget_hw_busif_value $periph "SPLB0"]
            #foreach master_if $master_ifs {
            #    set master [xget_hw_parent_handle $master_if]
            #    if { $master == $hw_proc_handle } {
            #        set if1_connected 1
```

```
            #    }
            #}

            # Alternate scheme
            set plb [xget_hw_busif_value $periph "SPLB1"]
            set plb_buses [xget_proc_dplb_buses $hw_proc_handle]
            foreach plb_bus $plb_buses {
                if { $plb_bus == $plb } {
                    set if1_isplb 1
                }
            }

#VGL#---------------------------------------------------------------------
#VGL# Fix for /* Definitions for peripheral XPS_MAILBOX_0 */ of
xparameters.h
#VGL#---------------------------------------------------------------------
#VGL# Replaced C_SPLB0_BASEADDR with C_SPLB1_BASEADDR in the line below
#VGL#            set mbox_baseaddr [xget_value $periph "PARAMETER"
"C_SPLB0_BASEADDR"]
            set mbox_baseaddr [xget_value $periph "PARAMETER"
"C_SPLB1_BASEADDR"]
#VGL#---------------------------------------------------------------------
        }

        if { $if1_isfsl == 1 || $if1_isplb == 1 } {
 …
```

(There are no fixes below this line; for the rest of the code, please refer to Xilinx EDK 9.2i installation)