PADERBORN UNIVERSITY

DISSERTATION

RECONOS64
Hardware-Software Multithreading for Heterogeneous Platform FPGAs

by
LENNART CLAUSING

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF DR.-ING.

to the

FACULTY OF COMPUTER SCIENCE, ELECTRICAL
ENGINEERING AND MATHEMATICS

March 2025

SUPERVISOR:
Prof. Dr. techn. habil. Marco Platzner

REVIEWERS:
Prof. Dr. Marco Platzner
Prof. Dr. Sybille Hellebrand

EXAMINATION COMMITTEE:
Prof. Dr. Marco Platzner
Prof. Dr. Sybille Hellebrand
Prof. Dr. Lin Wang
Prof. Dr. Wolfgang Müller
Dr. Heinrich Riebler

ACKNOWLEDGMENTS

I want to thank many people for their support towards as well as during this dissertation. In the following lines, I would like to highlight some people in particular.

First, I would like to thank my supervisor, Prof. Dr. Marco Platzner, for his trust, impulses, and continuous advice, without which this thesis would not have been possible.

I would like to thank Prof. Dr. Sybille Hellebrand for serving as a reviewer for this thesis, as well as Prof. Dr. Lin Wang, Prof. Dr. Wolfgang Müller, and Dr. Heinrich Riebler as members of the examination committee.

I would further like to thank all current and former colleagues from the Computer Engineering Group at Paderborn University for the positive working environment and many inspiring discussions.

In addition, I would like to thank all students involved in the ReconOS and ReconOS64 projects under my supervision as student assistants or during Bachelor's and Master's thesis projects.

Finally, I would like to express my sincerest thanks to my parents and family. Without their continuous support, encouragement, and fostering of curiosity, my academic journey and, therefore, this thesis would not have been possible.

ABSTRACT

Heterogeneous computing devices comprising multiple processing systems combined with a programmable logic like an FPGA have become increasingly popular. Developing and runtime managing these heterogeneous platform devices, however, still poses multiple challenges. The programming paradigms for hardware and software applications fundamentally differ, while the runtime management requires additional attention. The first steps towards unifying design flows have been taken by creating high-level synthesis tools. However, these tools focus on the design process and lack runtime management features.

FPGA operating systems aim to close this gap by providing runtime management features and build tools. ReconOS, as an example of an FPGA operating system, allows bridging the hardware-software gap by integrating hardware-located computation units as threads into a host operating system, allowing for familiar inter-thread programming paradigms to be extended to FPGAs.

This thesis presents ReconOS64, a successor and fork of ReconOS, aiming at modern 64-bit platform systems-on-chip. Besides establishing support for this platform and incorporating new system generation processes, ReconOS64 provides additional features for both development and runtime management. By providing a flexible grouping mechanism for hardware-located threads, partial reconfiguration for exchanging individual threads at runtime is simplified. Further runtime flexibility was achieved by implementing a multi-clock architecture, allowing individual groups of threads to be executed at different frequencies, which can be adapted during runtime with a low reconfiguration overhead.

In addition, this thesis provides an overview of further development approaches on various systems based on ReconOS64 and ReconOS.

ZUSAMMENFASSUNG

Heterogene Architekturen, welche aus mehreren Recheneinheiten im Verbund mit einer programmierbaren Logik bestehen, gewinnen zunehmend an Verbreitung. Die Entwicklungsprozesse sowie die Steuerung solcher heterogenen Umgebungen stellen jedoch nach wie vor eine Herausforderung dar. Die Programmiertechniken für Hardware- und Software-Anwendungen unterscheiden sich bereits grundlegend; zudem benötigt die Steuerung zusätzliche Aufmerksamkeit. Die Verwendung von Hochsprachen-Synthesewerkzeugen (High-Level Synthesis) für das Hardwaredesign stellen einen ersten Schritt hin zu einem einheitlichen Entwurfsprozess dar. Allerdings beschränkt sich deren Einsatz auf den Designprozess, während das Management zur Laufzeit keine Berücksichtigung findet.

FPGA-Betriebssysteme versuchen diese Lücke durch die Bereitstellung von Management- und Entwurfswerkzeugen zu schließen. Ein Beispiel für ein solches System ist ReconOS, welches die Hardware-Software-Interaktion durch Anbindung von hardwareseitigen Recheneinheiten als Threads in einem Host-Betriebssystem ermöglicht. Dadurch werden die aus der Softwareentwicklung bekannten Methoden zur Kommunikation zwischen mehreren Threads auf FPGA-Systeme ausgeweitet.

Diese Arbeit stellt ReconOS64 vor, welches als Ableger von ReconOS auf aktuellen 64-bit Systemen verwendet werden kann. Neben der Unterstützung aktueller Plattformen sowie der Einrichtung eines neuen Designprozesses werden durch ReconOS64 zusätzliche Möglichkeiten sowohl zur Entwicklung wie auch zum Laufzeitmanagement geschaffen. Durch flexible Gruppierungen von Hardware-basierten Threads wird die Verwendung partieller Rekonfiguration zum Austausch einzelner Threads zur Laufzeit vereinfacht. Zudem erhöht die Entwicklung einer Multi-Taktsignal-Architektur die Flexibilität zur Laufzeit, da hierdurch einzelne Gruppen von Threads mit unterschiedlichen Taktsignalen versorgt werden können, welche zudem zur Laufzeit mit geringer Unterbrechungszeit verändert werden können.

Zusätzlich werden im Rahmen dieser Arbeit weitere Entwicklungsprojekte auf verschiedenen Systemen für ReconOS64 und ReconOS vorgestellt.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACP Accelerator Coherency Port
APU Application Processing Unit
ATF ARM Trusted Firmware
AXI Advanced eXtensible Interface
BRAM Block-RAM
CDDC Clock Divide Dynamic Change
CMA Contiguous Memory Allocator
COTS Commercial Off-the-Shelf
CPU Central Processing Unit
CSU Configuration Security Unit
DMA Direct Memory Access
DRP Dynamic Reconfiguration Port
DT Delegate Thread
EL Exception Level
FIFO first in, first out
FPGA Field-Programmable Gate Array
FSBL First-Stage Boot Loader
GPU Graphics Processing Unit
HBICAP High-Bandwidth ICAP
HDL Hardware Description Language
HLS High-Level Synthesis
HP High-Performance
HWT Hardware Thread
ICAP Internal Configuration Access Port
ILA Integrated Logic Analyzer
IP Intellectual Property
IRQ Interrupt Request
ISA Instruction Set Architecture
ISE Integrated Synthesis Environment
ISR Interrupt Service Routine
LPA Large Physical Address
LTS Long Term Support

MEMIF Memory Interface
MMCM Mixed-Mode Clock Management
MMU Memory Management Unit
MPSoC Multi-Processor System-on-Chip
NFS Network File Share
NOC Network on Chip
OSFSM Operating System Finite State Machine
OSIF Operating-System Interface
PCAP Processor Configuration Access Port
PFD Phase-Frequency Detector
PGD Page Global Directory
PL Programmable Logic
PLL Phase-Locked Loop
PMU Platform Management Unit
PR Partial Reconfiguration
PS Processing System
RDK ReconOS Development Kit
RPU Real-Time Processing Unit
RTOS Real-Time Operating System
SoC Sytem-on-Chip
TCL Tool Command Language
TLB Translation Lookaside Buffer
TTBR Translation Table Base Register
URAM UltraRAM
VCO Voltage-Controlled Oscillator
VM Virtual Machine

1

INTRODUCTION

In recent years, the field of computing has seen many changes regarding the type of workloads, the physical location of compute loads – centralized or at the edge – and ways to handle these tasks.

The traditional use of Central Processing Units (CPUs) for computing and Graphics Processing Units (GPUs) for graphic rendering was one of the first to be opened to new computing approaches. Many steps into more heterogeneous systems followed, such as using accelerator cards for specific workloads. It has become common for mass-market CPUs to integrate performance- and efficiency-oriented cores into the same chip, providing a heterogeneous computing environment. Similarly, adding accelerators for neural networks to mobile processors was a step in a more heterogeneous direction. The close cooperation or mergers between CPU and Field-Programmable Gate Array (FPGA) vendors like Intel and Altera or AMD and Xilinx highlight the importance of extending CPUs by additional types of computation devices.

An example of such heterogeneous computation devices is an FPGA-integrating Sytem-on-Chip (SoC). These systems integrate a tightly coupled CPU and FPGA unit, along with additional components, into a single chip package. However, the sole availability of heterogeneous systems does not bring a performance benefit for individual applications. Instead, it poses an additional overhead on the programmer's side since the development processes for traditional CPU-targeting software applications and FPGA-focused hardware designs are fundamentally different.

In previous years, there was a separation between application software and hardware logic design, which led to mostly separated development processes and languages, effectively utilizing the hardware and software components like two individual systems with a communication channel in between them.

The first step in the direction of more uniform development processes was the introduction of High-Level Synthesis (HLS) tools, which allowed the generation of hardware design descriptions using traditional programming languages from the software world, such as C/C++. Despite the basic concept of HLS being old [47], it took until the switch from Integrated Synthesis Environment (ISE) to Vivado for Xilinx before presenting its use for FPGA devices.

However, high-level synthesis alone only simplified part of the hardware-software development gap since it only allowed for describing algorithms using a programming language familiar to the software world. The additional development overhead of creating the heterogeneous design consisting of FPGA configuration, communication channels, and management features was still hindering development processes.

FPGA operating systems such as ReconOS aim to facilitate the integration of hardware and software development steps into a more uniform flow. They provide different mechanisms for integrating the system components from an architectural and programming point of view, separating the application development for hardware and software from the underlying system architecture.

The recent changes towards more powerful SoC devices based on newer architectures posed an additional difficulty by offering high amounts of logic inventory that can often be shared between multiple individual accelerator designs to achieve high spatial efficiency. Therefore, throughout this thesis, an integrated solution for the development and management of modern heterogeneous systems-on-chip is presented.

1.1 THESIS CONTRIBUTIONS

The main focus of this thesis is the development of ReconOS64, a system for the development and execution of multi-threaded hardware-software designs on modern Multi-Processor System-on-Chip (MPSoC) platforms. The key contributions include:

- Adaption of the ReconOS concept and its system generation tool flow to modern 64-bit platforms.
- Flexible resource allocation for an arbitrary number of hardware-located threads on a multi-threaded system design.
- Runtime adaptable clocking infrastructure for individual groups of hardware-located threads.

1.2 THESIS ORGANIZATION

The following chapters of this thesis focus on individual components and applications for ReconOS64, moving from a top-level overview to a more in-depth description.

- Chapter 2 provides general background information and an overview of other similar and different approaches for heterogeneous systems, including the ReconOS concept.

- Chapter 3 focuses on the ReconOS64 system architecture created throughout this thesis to support modern 64-bit-based heterogeneous systems. Besides architectural details and decisions, this chapter also covers its new build system and partial reconfiguration options. It also presents a runtime performance evaluation covering multiple aspects of the system.
- Chapter 4 focuses on the build- and runtime-related challenges, concepts, and approaches ReconOS64 includes to handle partial reconfiguration and clock management for workload adaptive system designs.
- In Chapter 5, use cases and projects based on ReconOS and ReconOS64 are presented.
- This thesis concludes with Chapter 6, additionally providing a perspective for future development paths of the ReconOS64 system.

2

BACKGROUND AND RELATED APPROACHES

This section aims to provide a general background regarding the challenges targeting heterogeneous Sytem-on-Chips (SoCs) that consist of a Processing System (PS) coupled with a Programmable Logic (PL) component inside a single chip. This tight coupling allows for fast data transfer and control mechanisms to be established between system components running on both parts. As a downside, due to the inherent differences in programming and operation between FPGA hardware and CPU software, creating a uniform approach for targeting both components from within a single toolchain is still an ongoing challenge.

2.1 HARDWARE-SOFTWARE COUPLING FOR FPGA SYSTEMS

Multiple authors have presented approaches for FPGA systems to enable hardware-software coupling. The approaches vary in their respective perception of the interaction level and their resource sharing methods. A very early approach to categorizing modes of hardware-software interactions for FPGAs was made in 1996 by Brebner [21]. In this work, the author describes two possible methods for interacting with FPGA-based systems: One option, titled *sea of accelerators*, is to integrate individual accelerators as memory-mapped units. The second approach, titled parallel harness, is described as cooperating units interacting with each other and the operating system. This definition summarizes the idea of FPGA operating systems, which aim at creating a system for coupling hardware and software components in various ways.

2.2 FPGA OPERATING SYSTEMS

One approach for FPGA operating systems is ReconOS [1] and subsequently ReconOS64, which will be covered in detail in Section 2.3 and Chapter 3. ReconOS, which was first presented in 2007 [41], was one of the earlier approaches. Its main idea is based upon the concept of threads, known from the software world, transferring the inter-thread communication and synchronization methods across the hardware-software boundary. This is accomplished by forwarding operating

system API calls from a Hardware Thread (HWT) located in the PL to the host operating system running on the PS side. This allows for cooperative multi-threading by using POSIX-based multi-thread synchronization methods. Later additions include virtual memory access for hardware-based threads.

Other approaches for reconfigurable systems integrate hardware-based components in different ways. The following projects are examples of FPGA operating systems. An overview of multiple FPGA OS concepts is presented in [26].

An early discussion and implementation of a reconfigurable hardware operating system (RHWOS) is presented in [65], demonstrating a partially reconfigurable system with variable-sized task frames for exchangeable user logic tasks on a Virtex-II FPGA.

BORPH [59] or *Berkeley Operating system for ReProgrammable Hardware*, presented another early concept of adding FPGA hardware resources to a Linux operating system, conceiving them as hardware processes and offering certain OS functions to them. The system is provided as a patch file for the Linux kernel version 2.4. While this enables a quick application for this specific kernel release, it lacks an extensible and generalized tooling approach, which is essential for the development process of heterogeneous applications.

Hthreads [10] presents a concept of hybrid threads utilizing operating system functions from both hardware as well as software locations. The API calls for hardware and software threads should be uniform, thus providing an identical interface from an application programmer's perspective.

FUSE [31], meaning *Front-end User Framework*, abstracts hardware-located elements as tasks in an application running on a PetaLinux-based system. It aims at runtime flexibility by providing a dynamically loadable kernel module for interfacing the FPGA tasks instead of the delegate threads of the ReconOS system.

SPREAD [66] (*Streaming-based Partially Reconfigurable Architecture and Programming Model*) focuses on streaming-based hardware and a thread model supporting switching between software and hardware execution. Besides an interface to the system bus, each hardware thread is connected to a shared memory controller.

R3TOS [32, 33], short for *Reliable Reconfigurable Real-Time Operating System*, aims at providing a fault-tolerant execution environment for hardware multitasking. It utilizes a Commercial Off-the-Shelf (COTS) Real-Time Operating System (RTOS) kernel extended by a scheduler, allocator, and reconfiguration system aware of hardware defects, avoiding damaged zones when initializing tasks. Communication is established via a Network on Chip (NOC).

The *Latency-insensitive Environment for Application Programming* or LEAP [28] is built around latency-insensitive communication channels between individual modules that can span across multiple FPGAs and

provide an in-order data transfer with a FIFO-based interface towards the programmer.

Examples of contemporary approaches targeting the UltraScale+ Multi-Processor System-on-Chip (MPSoC) devices include the *Zynq UltraScale+ framework for OpenCL HLS applications* or ZUCL [52]. While providing partial reconfigurability, runtime management, and a concept of hardware slots, ZUCL focuses on OpenCL kernels implemented via High-Level Synthesis (HLS) and communicating via hard-mapped physical memory addresses. In contrast, ReconOS and ReconOS64 focus on enabling regular inter-thread communication and synchronization by hardware threads created via Hardware Description Language (HDL) or HLS, including virtual memory access. Support for virtual memory and data paths exceeding 32 bit were added to its successor ZUCL2.0 [51]. The FOS or *FPGA Operating System* project is based on ZUCL, extending its runtime management features for different accelerator configurations at runtime [61, 62].

The *Xilinx Runtime* or XRT [9] is a library that by itself does not aim at providing a complete operating system solution but gains relevance as it is a vendor-provided set of components for management and data transfer on both PCIe-based data center systems as well as Zynq UltraScale+ MPSoC-based edge devices. XRT is an integrated component for projects using the PetaLinux-based build process. It can be used for creating Direct Memory Access (DMA) buffers between software and hardware using the *zocl* driver [8], which creates contiguous physical addresses obtained via the Linux Contiguous Memory Allocator (CMA) to be shared between the operating system and accelerator. Additional XRT management functions include bitstream loading and interrupt handling. The XRT system is also used as a backbone of the *Xilinx FPGA Resource Manager* XRM [7] for accessing individual hardware elements as compute units.

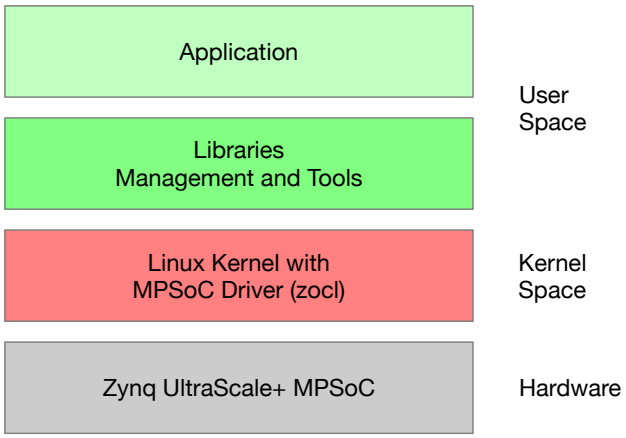

Figure 2.1: XRT Software Stack. Based on [9].

Figure 2.1 presents an overview of the XRT software stack, from the hardware layer to the intermediate Linux kernel layer where the *zocl* driver is included. The user space layer offers libraries utilizing the kernel driver to manage platform components such as the clocking system or collect measurements.

2.3 THE RECONOS SYSTEM

The main goal of the ReconOS system is to ease the process of generating applications for heterogeneous platform systems that include an FPGA in conjunction with some other general-purpose processing system. ReconOS offers an interface, a build tool system, and control mechanisms to the programmer. ReconOS64, the latest development step and the main product of this thesis, will be described in the next chapter. In preparation, this section will present an overview of the system and the history of the ReconOS development.

2.3.1 *Overview and Predecessors*

The ReconOS system was first presented in [41], targeting the eCos [27] RTOS and offering a unified programming model for hardware-located threads on a Xilinx Virtex-II and Virtex-4 FPGA in conjunction with an IBM PowerPC CPU. It included the concept of an Operating-System Interface (OSIF) for communicating between the hardware-based threads and the CPU. In addition, it introduced an Operating System Finite State Machine (OSFSM) to synchronize the hardware thread to the OS.

Later development steps included additional components, such as the Memory Interface (MEMIF) and virtual address translation via a hardware-based Memory Management Unit (MMU) and the switch to Linux as its primary host operating system [54]. Further steps provided switching of the toolset from Xilinx Integrated Synthesis Environment (ISE) to Vivado and first approaches towards clock configuration using Phase-Locked Loop (PLL) tiles.

Figure 2.2 presents a timeline of the development process, listing the primary target operating system, hardware platforms, and CPU architectures, as well as toolchains used. Additional operating systems and tool flows have been evaluated, which will be described in later sections. The FPGA systems and architectures marked green are currently supported by ReconOS or ReconOS64, while support for the red platforms has been removed in later versions.

Figure 2.3 presents a high-level architectural overview of the original ReconOS project's most recent version as presented in its architecture description [11]. The parts to the right and left of the reconfigurable slots resemble the OSIF and MEMIF components, responsible for forwarding operating system calls and enabling memory access.

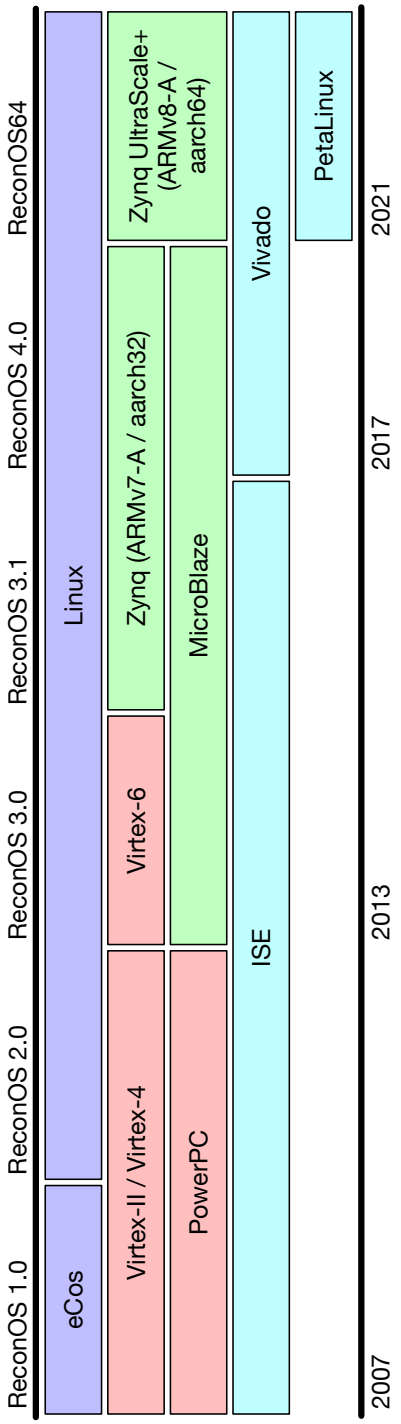

Figure 2.2: ReconOS development timeline and feature comparison, including ReconOS64. Based on [54].

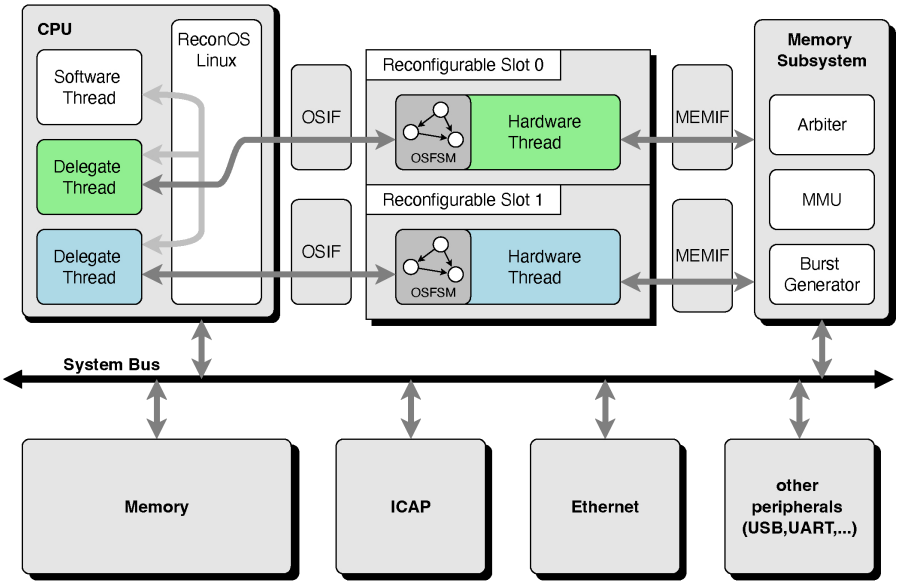

Figure 2.3: ReconOS system structure showing PS- and PL-located elements including connections and system components. Adopted from [11].

The basic concept of the hardware thread-based ReconOS execution model is enabled by the presence of lightweight Delegate Threads (DTs) for all hardware threads, shown within the CPU. A delegate thread works as a representative of the HWT facing the remaining software system. Towards the operating system, it acts like a regular thread, which means it can perform OS calls and is a member of the application tasks' virtual memory mapping. This mechanism sets the ReconOS approach apart from other approaches, which use programmable logic as accelerators. In ReconOS, user logic within the FPGA can equally participate in the cooperative multi-threaded environment. Despite creating one software delegate thread per hardware thread, the delegate thread is not executing any actual data processing. It is, therefore, consuming only small amounts of system resources for its interrupt-triggered delegate actions. Table 2.1 lists available ReconOS OSIF calls for inter-thread communication, including a message box system, mutexes, semaphores, and condition variables. The available calls are presented with their software-sided signature. The parameter list of each call includes a resource pointer `*rp` to identify the respective entry within the list of resources defined within the project configuration file.

Imposing thread-like interfaces to hardware-mapped application components creates the ability to interact between threads using standardized communication and synchronization methods known from the software world for a long time, such as POSIX-based API calls. Enabling synchronization primitives like mutexes or semaphores across the hardware-software boundary allows for tight cooperation between

Table 2.1: Inter-thread operating system calls delegated by ReconOS. Each resource is referenced by a resource pointer `*rp`. Based on [12].

ReconOS Call	POSIX Call	Blocking
`int mbox_put(*rp, uint32_t in)`	similar to `mq_send`	no
`uint32_t mbox_get(*rp)`	similar to `mq_receive`	yes
`int mbox_tryget(*rp, *output)`	similar to `mq_receive`	no
`int mutex_lock(*rp)`	`pthread_mutex_lock`	yes
`int mutex_unlock(*rp)`	`pthread_mutex_unlock`	no
`int mutex_trylock(*rp)`	`pthread_mutex_trylock`	no
`int sem_wait(*rp)`	`sem_wait`	yes
`int sem_post(*rp)`	`sem_post`	no
`int cond_wait(*rp, *mutex)`	`pthread_cond_wait`	yes
`int cond_signal(*rp)`	`pthread_cond_signal`	no
`int cond_broadcast(*rp)`	`pthread_cond_broadcast`	no

those SoC domains, both from an execution and a development point of view. The ReconOS library offers API calls for its supported functions for implementations based on software as well as using HLS and HDL definitions, allowing for a more unified development flow for hardware and software.

3

RECONOS64: DESIGN AND IMPLEMENTATION

Due to the constant change of FPGA-based systems with the tendency towards bigger, more complex heterogeneous architectures incorporating multiple different processing and logic elements, FPGA operating systems have to be adapted and extended to enable targeting such platforms. This chapter focuses on ReconOS64 as a successor of the ReconOS project as described in Section 2.3. While keeping the general concepts and functionality in place, ReconOS64 enables compatibility and extended features to meet the requirements for modern system-on-chip architectures of the Zynq UltraScale+ Multi-Processor System-on-Chip (MPSoC) series.
The following sections start with an overview of the new target platform before focusing on individual development details. After describing the memory management, the build system, and partial reconfiguration features, the later sections of this chapter present runtime evaluation measurements for various aspects of the overall system performance as well as its hardware utilization.

3.1 OVERVIEW AND DEVELOPMENT

ReconOS64 is based on the previous ReconOS [1] system, developed to implement new platforms and build toolchains while keeping most of the compatibility with applications from previous versions intact. However, due to multiple architecture-based modifications, with some of them introducing breaking changes to the system and its build process, a new name was chosen to represent the new and modern target architecture of 64-bit-based systems.

Figure 3.1 presents a more in-depth topology of a ReconOS64 system on a Zynq UltraScale+ MPSoC device. The left side reflects the Processing System (PS) running the host Linux system, separated into the kernel space running the ReconOS64 kernel driver for accessing hardware components and interrupts on the one hand and the user space running the main application embedding the ReconOS64 runtime together with all software and delegate threads. The user space application is similar to a regular multi-threaded software application with an added ReconOS64 runtime environment. It provides calls to

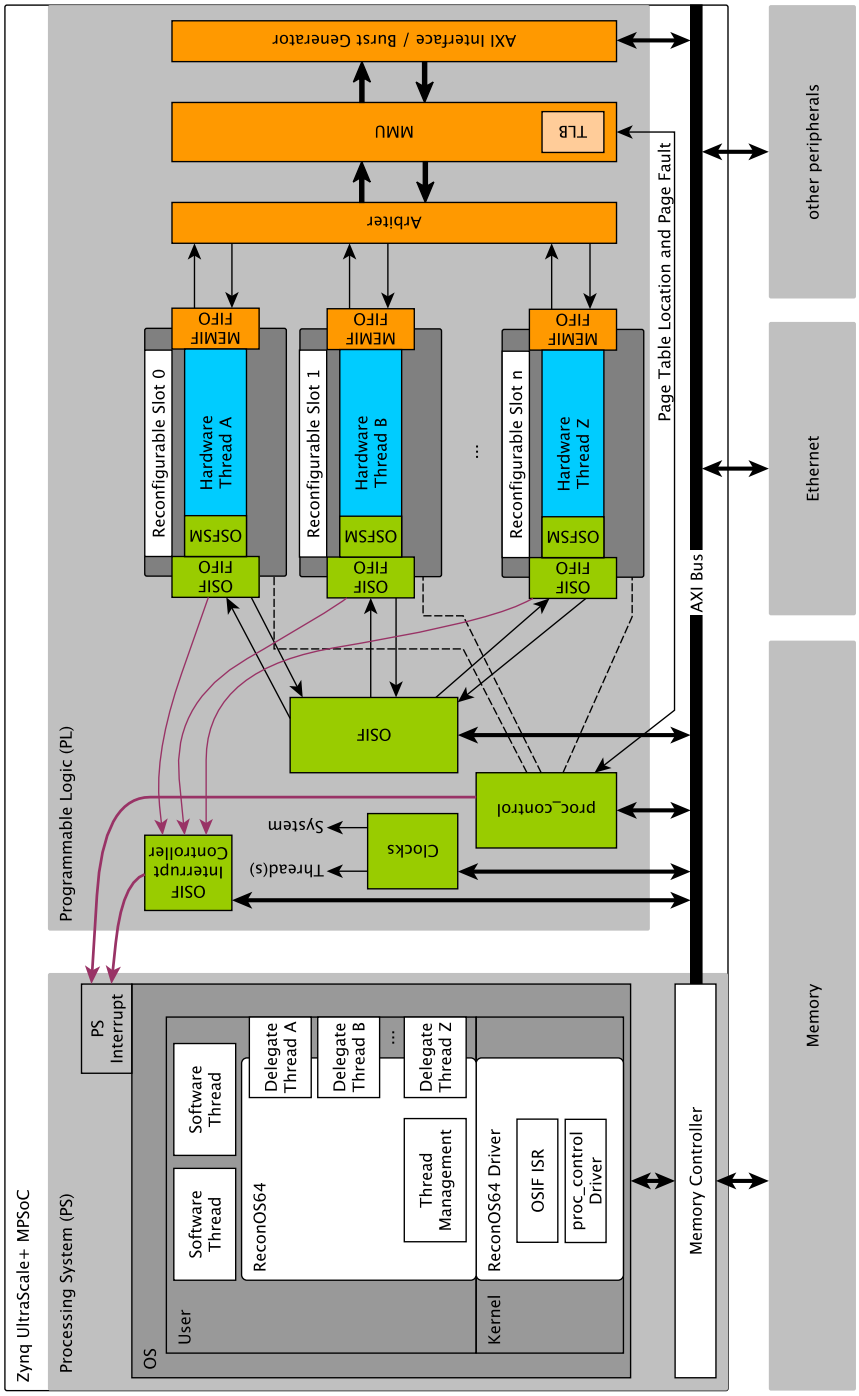

Figure 3.1: Detailed ReconOS64 system layout showing Processing System (PS) and Programmable Logic (PL) components including control, data and interrupt signal flows. OSIF components are shown in green, and MEMIF components in orange. Extended version of Fig. 2.3. First presented in [25].

control the hardware-located components, so besides the ability to spawn a software thread, the application receives the added capability to spawn a thread located in hardware. In addition, the runtime library provides methods for setting individual clock frequencies for hardware slot groups, which will be described in a later section.

The right side of Figure 3.1 shows the Programmable Logic (PL) with $n+1$ reconfigurable slots `0`, `1` and `n`. The content of the reconfigurable slots can either be exchanged via Partial Reconfiguration (PR) or can be defined statically. Each slot has a different Hardware Thread (HWT) A, B, or Z placed inside, including its respective Operating System Finite State Machine (OSFSM) for synchronizing software and hardware thread procedures. The OSFSM is a user-defined state machine that handles the cooperation between software and hardware threads. Figure 3.2 presents two different examples. While the left one is a simple FSM only separating data input, data processing, and data output phases, the right example is a more complex state machine with additional states for acquiring synchronization clearances for reading and writing via semaphores or mutexes, respectively.

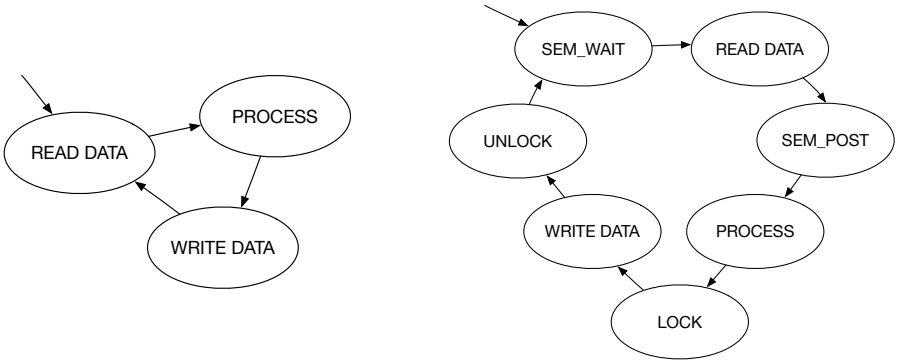

Figure 3.2: Two examples of ReconOS Operating System Finite State Machines (OSFSMs) of varying complexity. Based on [1].

Each HWT within its respective slot is connected to both Operating-System Interface (OSIF) and Memory Interface (MEMIF) structures, decoupled by first in, first out (FIFO) buffers each. Regarding the hardware design implementation, the FIFOs are each again split into two different unidirectional buffers, one of which contains data originating from the hardware thread while the second FIFO contains data targeting the respective thread. Since both of those buffers act as a single, bidirectional FIFO from a logical perspective, they are condensed into a single buffer in figures and descriptions. The buffers operate asynchronously, thus acting as a decoupler between the global system's and the independent thread's clock domain.

The green OSIF components to the left of the hardware threads in Figure 3.1 enable the exchange of messages and operating system

calls, including their return values between the delegate and hardware threads, as shown by the black arrows. The purple interrupt lines are gathered within the OSIF interrupt controller, and an interrupt is forwarded to the CPU once a hardware thread initializes an OS call. The interrupt-based delegation of calls allows for a low overhead since the delegation mechanism is only invoked from its waiting state once a call is signaled from within the PL.

The dotted black arrows represent `hwt_signal` signals that can be sent to each hardware thread for user-specific implementations. For example, they can indicate a yield signal to a hardware thread in an environment where partial reconfiguration can be invoked. Since these signal lines are outside of the regular OSIF message-passing mechanisms, they can evade blocking or queued-up calls within the OSIF FIFOs, thus reducing the reaction time of a hardware thread to this specific signal. An additional component related to runtime management is the *clock* hardware block, providing clock signals for all ReconOS64 components as well as all slot groups. A set of multiple independent, runtime-reconfigurable clocks can be generated and controlled via the ReconOS64 runtime system. A detailed description of the reconfigurable clocking resources will be provided in Section 4.2.

The orange components to the right of the reconfigurable slots of Figure 3.1 represent the MEMIF system for memory access using virtual addresses. Hardware threads issue read and write requests through the respective MEMIF FIFO, referring to the memory location by a virtual address from the application's context. The first component arbitrates different threads' requests in a round-robin way without prioritizing individual threads. One request and all related data to this transaction is then forwarded to the ReconOS64 Memory Management Unit (MMU), where the virtual address is replaced by the respective physical memory address, which is either retrieved from the Translation Lookaside Buffer (TLB) if it has been looked up before and has not been overwritten since then, or the physical address is generated via a multi-step page table lookup. The steps required for translating virtual addresses, as well as details regarding the target platform memory configurations, are described in Section 3.3.

The MMU has an additional signal path to the *proc_control* component, representing both the communication for the translation table base address as a starting point of the page table walk as well as signaling failed translation steps as page faults. The final MEMIF component is the memory interface and burst generator block. The burst generator translates the ReconOS64 read and write requests into Advanced eXtensible Interface (AXI) burst accesses. AXI accesses can span over multiple consecutive bursts from consecutive addresses, thus reducing the need to issue individual requests. Utilizing the maximum possible consecutive burst length for longer transactions ensures high

throughput, thus reducing the overhead of individual transactions. An evaluation of the MEMIF system is described in Section 3.7.

3.2 ZYNQ ULTRASCALE+ TARGET ARCHITECTURES

The switch from the previous 7-Series Zynq devices to the 8th generation or Zynq UltraScale+ device family changes the underlying PS architecture. Previous Zynq Sytem-on-Chip (SoC) devices included a Central Processing Unit (CPU) based on the ARMv7-A Instruction Set Architecture (ISA) (*aarch32*) like the ARM Cortex-A9 processor.

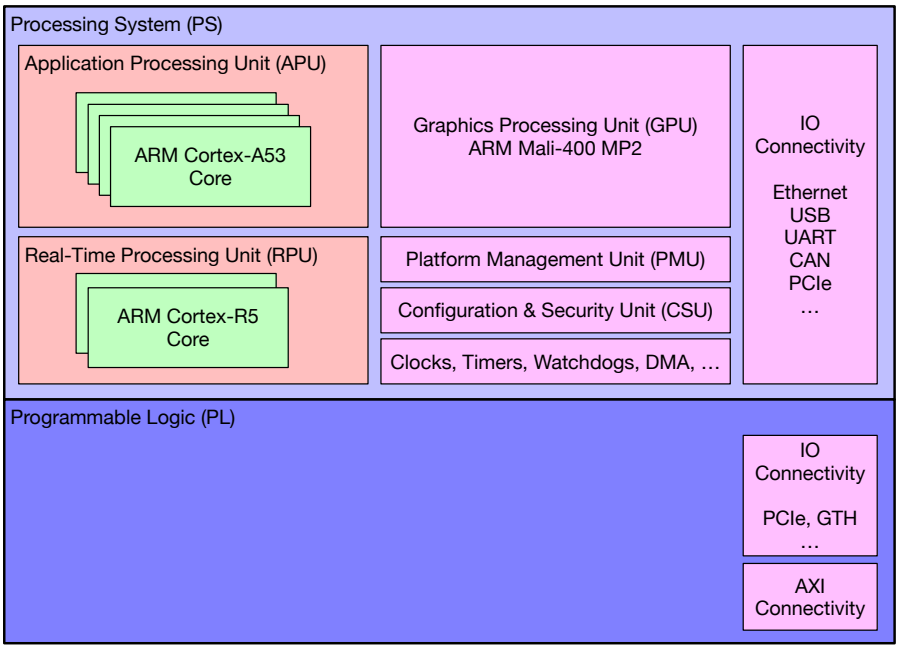

Figure 3.3: Zynq UltraScale+ MPSoC architecture overview displaying PS and PL components. Based on [5].

Starting from the 8-Series Zynq UltraScale+ devices, the main processor, referred to as Application Processing Unit (APU), changed to an ARM Cortex-A53 quad-core processor based on the ARMv8-A architecture, interchangeably denoted as *aarch64*. Additional heterogeneous units, such as the embedded ARM Mali-400 Graphics Processing Unit (GPU) or the ARM Cortex-R5 Real-Time Processing Unit (RPU), are not described in detail since they are currently not used for the ReconOS64 system.

Figure 3.3 presents an overview of the individual heterogeneous components. The top region represents the processing system, including the APU, RPU, GPU, and platform management units, besides examples of available external interfaces. The lower section shows the FPGA region together with its external interfaces and internal data ports. Depending on the specific MPSoC series, additional components

are included in either region.

The ARMv8-A architecture defines the available protection layers as individual Exception Levels (ELs) with increasing privileges, ranging from the most unprivileged EL0 to EL3 [20]. Figure 3.4 presents the available layers with increasing privilege from top to bottom. The Linux implementation for *aarch64* uses EL0 for user space applications, while the more privileged EL1 is used for kernel code [45]. Exception levels EL2 and EL3 can be used for Virtual Machine (VM) scenarios to provide hypervisor level above the individual guest kernels, with enhancing additions introduced to later architecture revisions, but none of them are used in ReconOS64 since it is executed in a non-virtualized environment.

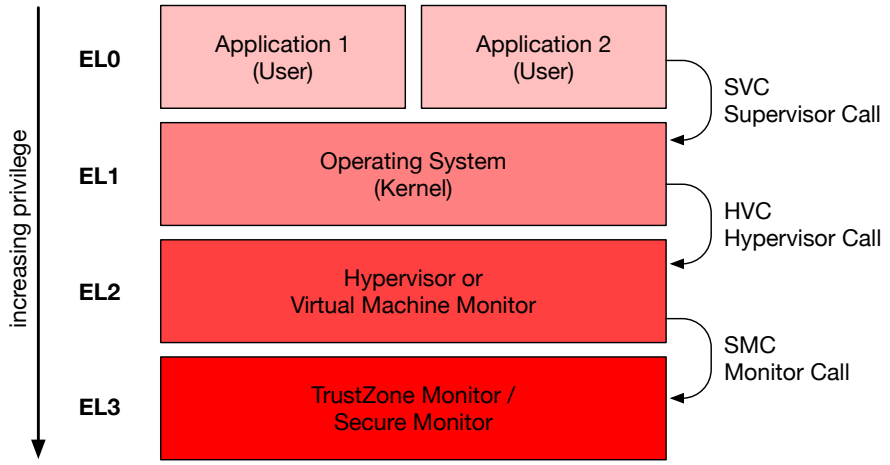

Figure 3.4: ARMv8-A exception levels including Linux kernel and user space. Based on [20, 45].

A significant change caused by the architecture switch to 64-bit from the perspective of ReconOS is the extension of the addressing space.

ARMv7-A systems offer a virtual address space of `0x00000000` to `0xFFFFFFFF` or 32-bit, thus limited to 4GiB [16, B3]. Despite offering an additional Large Physical Address (LPA) extension for 40-bit physical addresses [14, p. 22.4], the 32-bit addressing remained the default for this architecture and was used, for example, by ReconOS. With the introduction of the ARMv8-A architecture, the virtual address space got extended beyond 32-bit addresses [58], allowing for virtual addresses of up to 48 bit. This change results in increased sizes for `void*` and, in consequence, all other data type's pointers from 32 to 64 bit [15, Tab. 8-1] for *aarch64* applications.

Since the ReconOS system's mode of communication heavily depends on passing pointers to memory locations via OSIF messages before accessing them via MEMIF commands, ReconOS64 introduces a new default data type size of 64-bit for all communication messages,

thus allowing to fit the required pointer length into its message box system to exchange virtual addresses between hard- and software. All ReconOS system calls listed in table 2.1 that feature any input or output data have therefore been designed to use `uint64_t` as parameter and return types for ReconOS64. This structural change impacts both the OSIF communication since the message payload size was increased to 64-bit, as well as the MEMIF memory accesses, which also transfer 64 bits of data and use addresses exceeding 32 bits. To ensure aligned accesses, byte-addressed pointers for MEMIF transfers are aligned to 8 bytes by truncating the least significant three bit and replacing them with zero values.

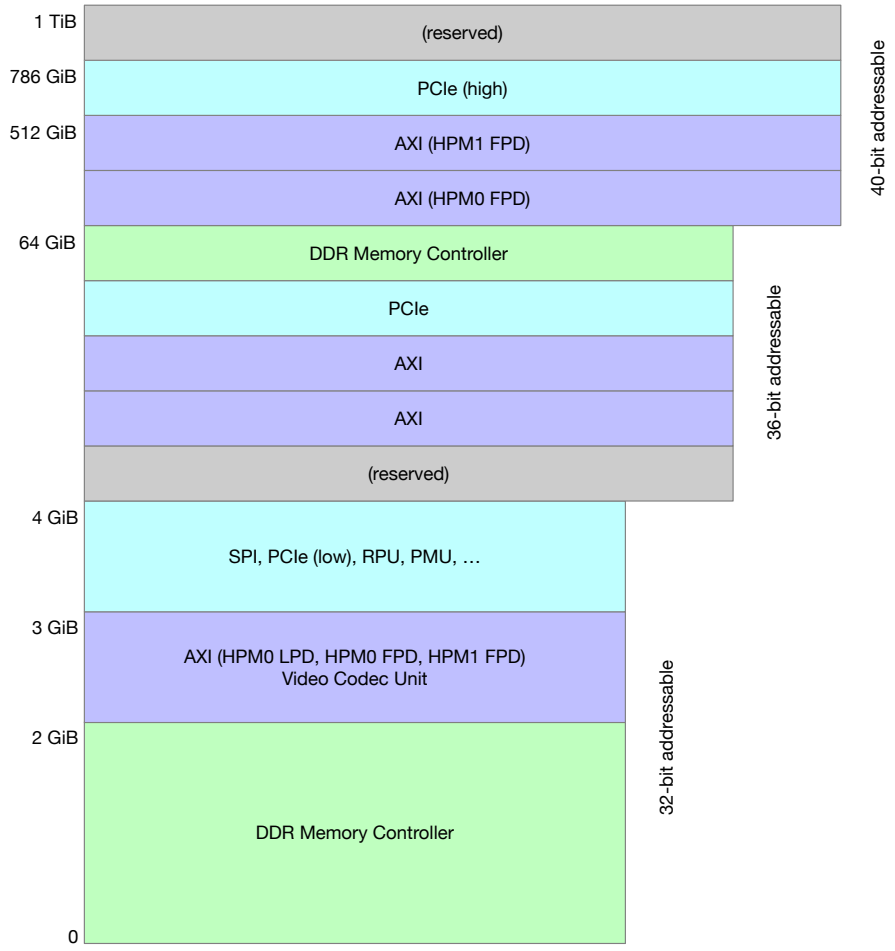

Figure 3.5: Zynq UltraScale+ memory map showing address ranges and minimum required address lengths. Adapted from [3, Fig. 10-1].

The Zynq UltraScale+ address map includes several ranges of addresses for system and flash memory as well as internal and external interfaces ranging up to 40-bit addresses, thus allowing for an addressable range of 1TiB. Addresses are extended to 48-bit, which is the address size used for the APU of the processing system [3, Chap. 10] and therefore also the physical address size used in ReconOS64. Fig-

ure 3.5 shows the memory map, including the cluttered DDR memory addressing, resulting in the lower 2GiB being addressable by 32-bit, while the remaining memory needs at least 36-bit physical addresses.

3.3 RECONOS64 MEMORY SYSTEM

3.3.1 *ARMv8 Memory Architecture and Linux Addressing Modes*

Bringing the ReconOS64 memory system to the new 64-bit hardware platform and architecture requires multiple aspects regarding addressing and virtual memory layout to be considered.

The ARMv8-A memory model used on the ReconOS64-supported systems allows for four main modes of address translation, depending on the virtual address length and page size, while the size of a physical address is fixed to be 48-bit [17, App. K7]. Additional ARMv8-A modes for backward compatible accesses structured similarly to ARMv7 exist but are out of the scope of ReconOS64 since the development aims at using the current capabilities of the architecture without imposing limitations caused by backward compatibility layers. In addition, further address translation modes specific to VM hypervisors on ELs of higher privileges than EL1 were added to later ARMv8-A extensions starting with ARMv8.2 but are out of scope as well since the Cortex-A53 CPU found in the Zynq UltraScale+ systems is limited to the ARMv8.0 architecture [19].

The four address translation modes are used by the architecture-specific implementation of the Linux host operating system [46] and should, therefore, be entirely supported by the ReconOS64 MMU. In order to keep compatibility with the default 4kiB page size that has been common for Linux operating systems, the first supported mode consists of 4kiB pages, referred to by a 39-bit virtual address, which will be translated into a 48-bit physical address using a three-step address lookup. This corresponds to the default memory mode used by PetaLinux-generated Linux systems for UltraScale+ targets. Figure 3.6 presents an overview of the address translation process for a user space virtual address on this configuration.

The second address translation option allows for using the longer virtual address format of 48 bit for 4kiB pages. The remaining two options allow for increasing the page size from 4kiB to 64kiB, thus resulting in 42-bit or 48-bit virtual addresses. Allowing for all combinations regarding page size and long or short addressing results in four implemented MMU designs that can be selected within the ReconOS64 configuration file. Figure 3.7 presents an overview of the various address formats as supported by the ReconOS64 MMU. Address components VA_{Ln} represent individual address bit ranges used during the translation at level n within the process.

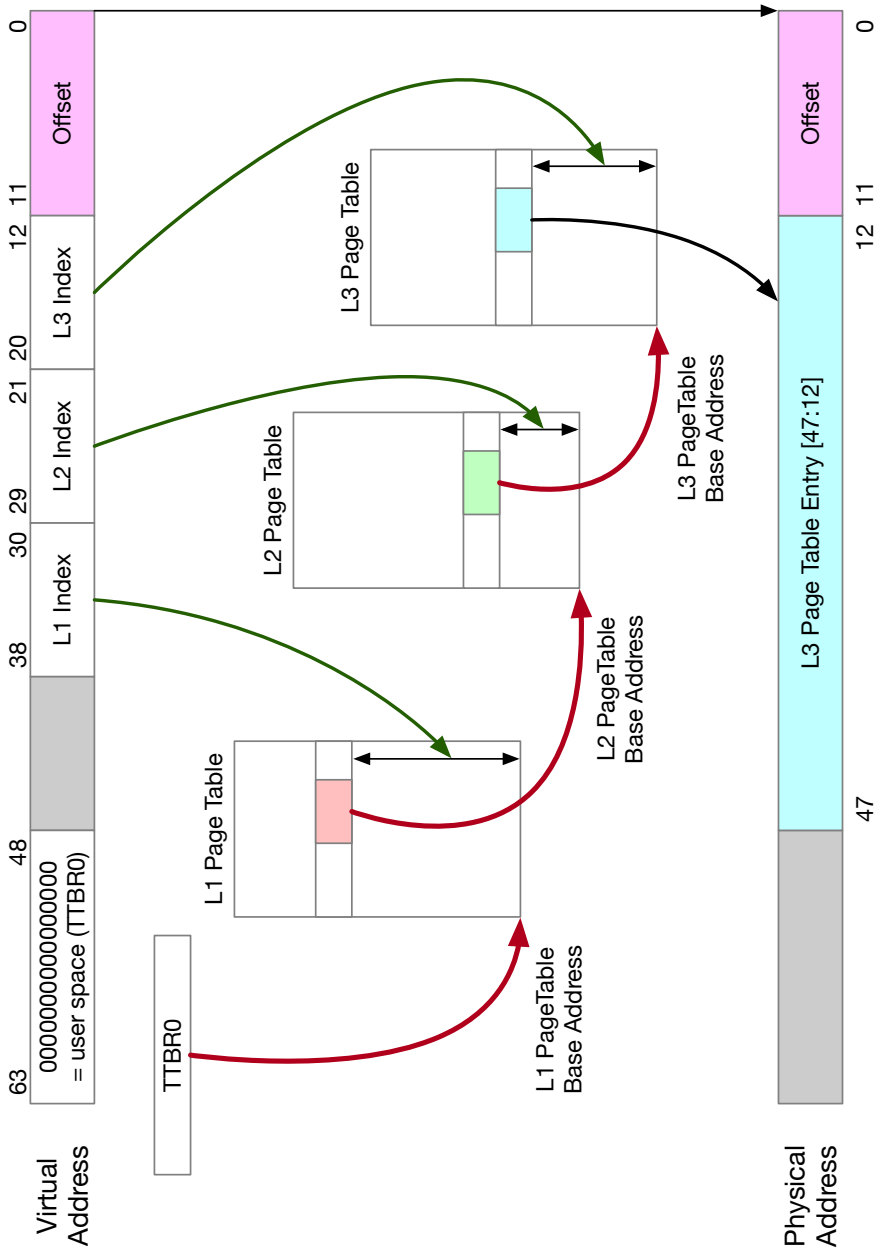

Figure 3.6: ARMv8-A page table walk example for a Linux user space address translated via a three-step lookup from a 39-bit virtual address on 4kiB pages. The Translation Table Base Register (TTBR) is fixed to TTBR0 for user space addresses. Adapted from [15, Fig. 12-8].

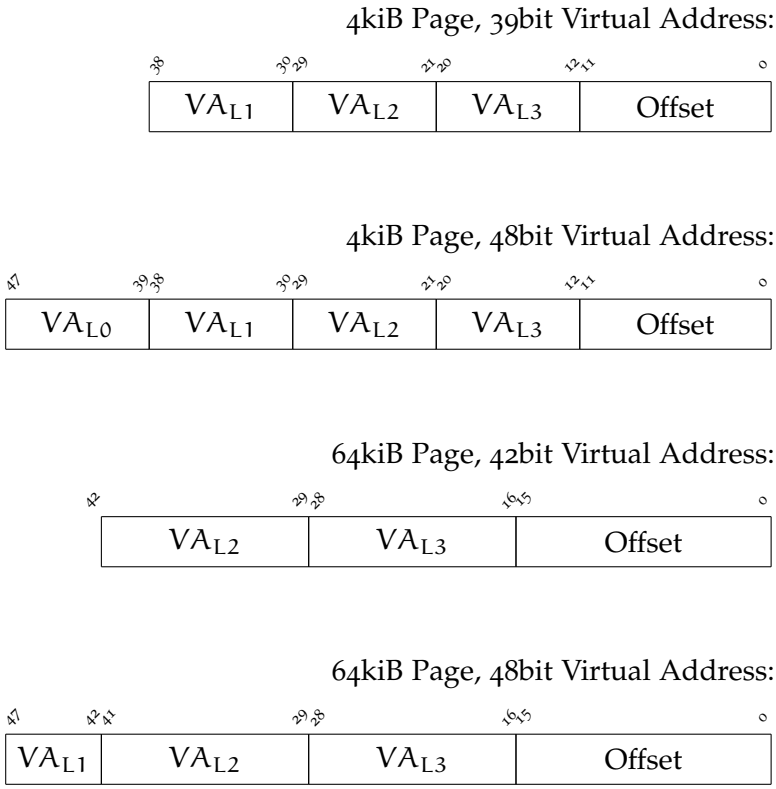

Figure 3.7: ARMv8-A virtual address translation formats of 4kiB and 64kiB pages with long and short addresses as supported by the ReconOS64 MMU. Based on [17, App. K7]

3.3.2 *Memory Management Unit*

The ReconOS64 MMU enables the address translation between the virtual address present in one of four different formats as defined earlier into a physical address. Since the MMU is embedded within the MEMIF path, the translation can happen on the fly and be transparent to the application. A hardware thread issues read and write commands to a virtual address like a regular software thread within an operating system using virtual memory would do. The MMU intercepts the address messages and looks up the respective physical memory location, injecting the result into the request before passing it to the AXI bus.

In order to support the individual configurations, four different variants of the MMU and TLB with configurable buffer depth are available within the unit's Intellectual Property (IP) core file. The ReconOS64 build toolchain includes the appropriate respective designs depending on the memory configuration passed to the build system within the project file.

All translations start with a common starting point, which is the base address of the first translation table. Its location can be retrieved from within the Translation Table Base Register (TTBR). For operating system environments that are non-virtualized and therefore only use ELs up to `EL1`, two TTBRs are available: `TTBR0_EL1` for user space virtual memory ranges and `TTBR1_EL1` for kernel space [15]. Therefore, `TTBR0` is the relevant base register for ReconOS64 in the context of a Linux-based user space application [45].

The address within the TTBR is used as the starting point for the first lookup of a page table walk. On Linux systems, this address can be retrieved by accessing the user space application's Page Global Directory (PGD) location. Each process within the Linux user space uses an individual PGD for its respective context [40]. This address is retrieved by the kernel module using a call to `virt_to_phys(current->mm->pgd)`, which points to the PGD of the currently active user space process, which in this case is the ReconOS64 application itself.

Figure 3.8 shows the individual steps and address segments involved in a 3-step lookup for a 4kiB page layout when translating a 39-bit virtual address into a 48-bit physical address. The virtual address includes the offset within the page, so for byte-addressing 4kiB of memory, it includes $\log_2 4096 = 12\text{Bit}$. For 64kiB pages, the offset would, therefore, be 16 bit. Depending on the virtual address length, the lookup can include two to four steps. For the provided example, a three-step lookup has to be used, so starting from the known base address of the first table, the MMU has to execute three memory read operations via the AXI bus. The initial query of the TTBR is omitted since the MMU already received and stored the PGD location when the ReconOS64 system was initialized.

For each lookup step, a 9-bit part of the virtual address is used in conjunction with a part of the address returned by the previous lookup in order to produce the following query address. Each lookup result includes two flag bits at the least significant position. `res[0]` specifies if the table entry is valid. If the ReconOS64 MMU receives a table entry marked invalid, it signals a failed translation by raising the `page_fault` signal. That signal will be forwarded via the kernel module and detected by the page fault handler within the user space ReconOS64 application. The fault address will be accessed from within the application context, thus ensuring its translation is present and accessible within the page table.

The ARMv8-A memory architecture additionally allows for situations where a page table walk can be aborted at an earlier stage if one of the intermediate lookup steps returns a block descriptor instead of a table descriptor [18]. The content of a lookup result is signaled by `res[1]` with a 1 indicating a table descriptor that can be further looked up in consecutive steps, while a 0 signals a block descriptor

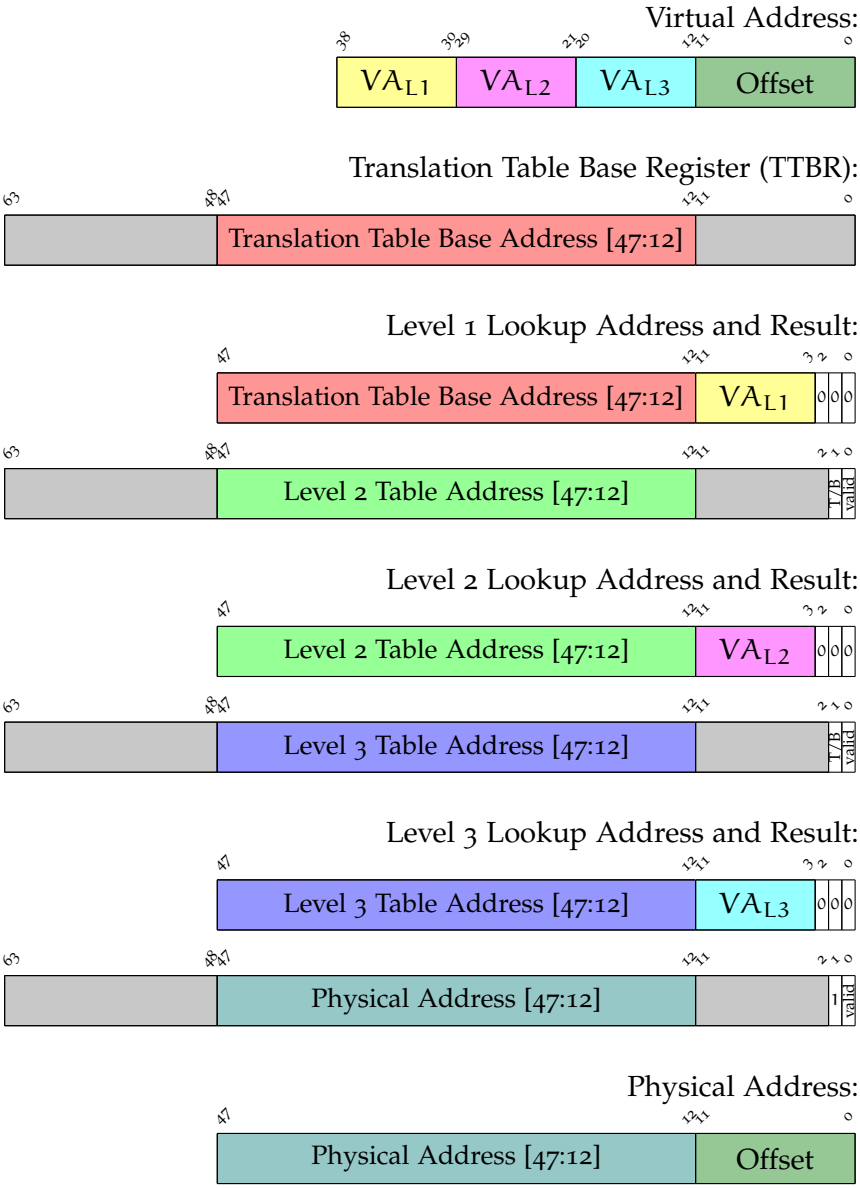

Figure 3.8: ARMv8 address translation for 39 Bit virtual addresses with a page size of 4kiB. Simplified by omitting additional bits not needed during the described translation process. Based on [17, App. K7]

that is the final result. This bit position is, therefore, marked as "T/B" in Fig. 3.8, while the final lookup step is expected always to return a non-block descriptor.

A runtime evaluation of the MMU and its lookup overhead, together with a discussion of the effect of page sizes, will be presented at the end of this chapter.

3.4 LINUX KERNEL MODULE

The ReconOS64 kernel module is used as a device driver to establish communication between the user space runtime application and hardware-located components. As shown in Figure 3.1, the kernel module can be separated into two main components, one for controlling the MMU and `proc_control` registers, and one for handling OSIF messages and interrupts. As described before, when initializing a ReconOS64 application, the kernel module receives requests to look up the PGD location of the respective user space application, which is then forwarded to the MMU as a starting point for address translations. If a translation failure is received via a page fault signal, the kernel module issues a command to access the address in question from the user space application. If this is a valid address in the context of the user space application, its translation should be made available in the MMU, but if the address is invalid or access is not allowed, the hardware-issued request results in an access violation when the user space application tries to access that address. This mechanism ensures that hardware-issued read or write requests are handled the same way regarding access violation exception signals as software-based threads.

The kernel module also provides two Interrupt Service Routines (ISRs) for triggering delegate thread actions or handling translation faults from the MMU. For the OSIF delegation, it registers an ISR to the Interrupt Request (IRQ) port connected to the OSIF interrupt handler inside the PL. When a delegation request is received, the kernel module invokes the respective delegation task in order to react to the request. In addition, the driver component for the `proc_control` registers the respective interrupt line for notifications of page faults.

Figure 3.9 presents the register of the `proc_control` module. The green 64-bit register at 0x00 contains the number of hardware slots, which will be statically generated during the build process. All red registers are related to the MMU. The PGD will be set by the kernel module. The page fault will be written by the MMU if a translation fails. The TLB statistics are written by the MMU, and the MMU reset can be issued by the kernel module if the MMU and TLB need to be reset. This could, for example, be used if a translation should be aborted or if the PGD changes and has to be reloaded by the MMU. The blue registers are bitmasks for individual hardware threads' reset

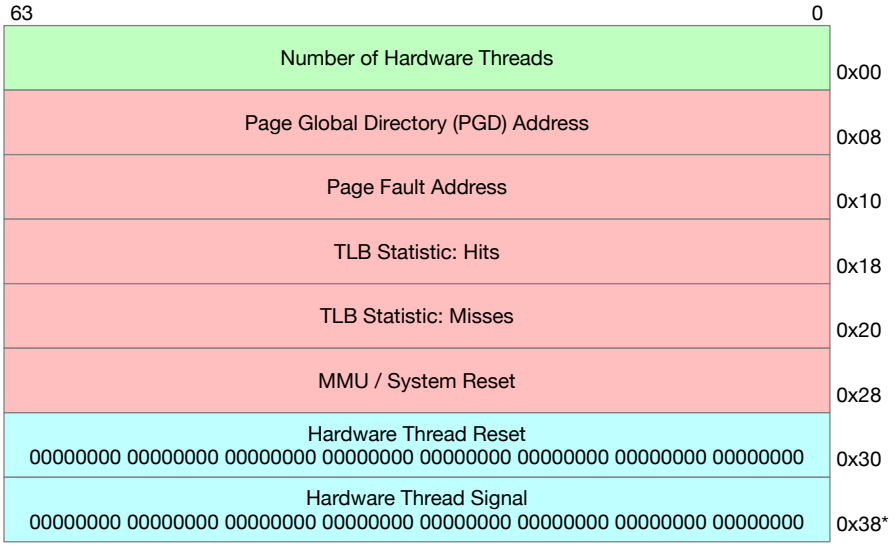

Figure 3.9: Registers of the `proc_control` hardware module. The address denoted with an asterisk can change if the number of hardware threads exceeds 64 due to a further reset register being added before the beginning of the signal registers.

and signal conditions, which can be set via the kernel module. Each hardware thread is assigned a bit in the 64-bit mask. If the number of threads should exceed 64, the reset register is extended into a second register at 0x38, thus shifting the beginning of the signal registers to 0x40.

The kernel module build process depends on the selected overall system build flow as described below. It is either built as a part of the PetaLinux build process or can be externally built for systems based on the Ubuntu Linux flow. For PetaLinux, it is included with the generated system image and can be enabled once a suitable bitstream, including ReconOS64 components, is programmed into the logic device.

3.5 RECONOS64 BUILD SYSTEM

The Zynq UltraScale+ MPSoC system added multiple heterogeneous components to the overall system, leading to a more extensive boot process compared to the 7-series Zynq devices. This also impacts the boot process of the overall system, which requires additional steps compared to previous SoC devices. For non-secure boot operation, the Zynq UltraScale+ Platform Management Unit (PMU) starts the Configuration Security Unit (CSU), which then loads and executes the First-Stage Boot Loader (FSBL) on the APU. The FSBL invokes the ARM Trusted Firmware (ATF) and a second-stage boot loader, commonly U-Boot, which then loads the Linux system [6, Ch. 7].

Manually building the individual software components along this boot path is not practical, so an integrated build tool was introduced to ReconOS64.

The following sections describe PetaLinux as a configurable build tool and later introduce a second tool approach using a pre-built Ubuntu Linux image as an alternative path for creating ReconOS64 systems.

3.5.1 *PetaLinux Build Flow*

ReconOS64 introduces the Xilinx PetaLinux [69] project for generating the Linux system components. This ensures an integrated flow of the individual build steps needed to boot a Linux environment on UltraScale+ series platforms. In addition, it allows for an easily configurable Linux kernel and root system. Besides the Linux components, such as the kernel and root file system, the earlier boot stages, including the second stage boot loader, as well as additional platform controller firmware components, are generated and built by PetaLinux.

Figure 3.10 provides an overview of the overall build flow. Blue elements represent vendor tools used, while green boxes depict components provided by ReconOS64. The sources on the left are separated between software threads, hardware threads, and system files. Software-placed threads, defined as C/C++ files, are compiled and combined with the runtime libraries for ReconOS64 thread management, resulting in the application binary including all threads. Hardware threads can either be defined using High-Level Synthesis (HLS) from C/C++ sources or via Hardware Description Language (HDL) descriptions. C/C++ hardware threads are converted using an intermediate Vivado HLS step, integrating the HLS library for embedding ReconOS64 calls in the high-level definition of threads. Threads defined via HDL files are directly integrated into the Vivado project, together with all pre-processed static IP cores of the ReconOS64 system. The Vivado bitstream generation process is controlled via Tool Command Language (TCL) scripts generated by the ReconOS Development Kit (RDK). Its output product is either a single bitstream or if partial reconfiguration is activated, a set of partial bitstreams and a single full bitstream. Partial reconfiguration is described in more detail in Section 3.6 as well as Chapter 4.

The PetaLinux section was newly introduced to the ReconOS system by ReconOS64. It replaces the previous manual compilation steps for the boot loader and Linux system, which were required for ReconOS on ARMv7. PetaLinux uses an exported Vivado hardware design file to infer parameters such as board type and processor configuration for creating the build project. The ReconOS64-specific components used during the PetaLinux tool flow include a PL base design bitstream

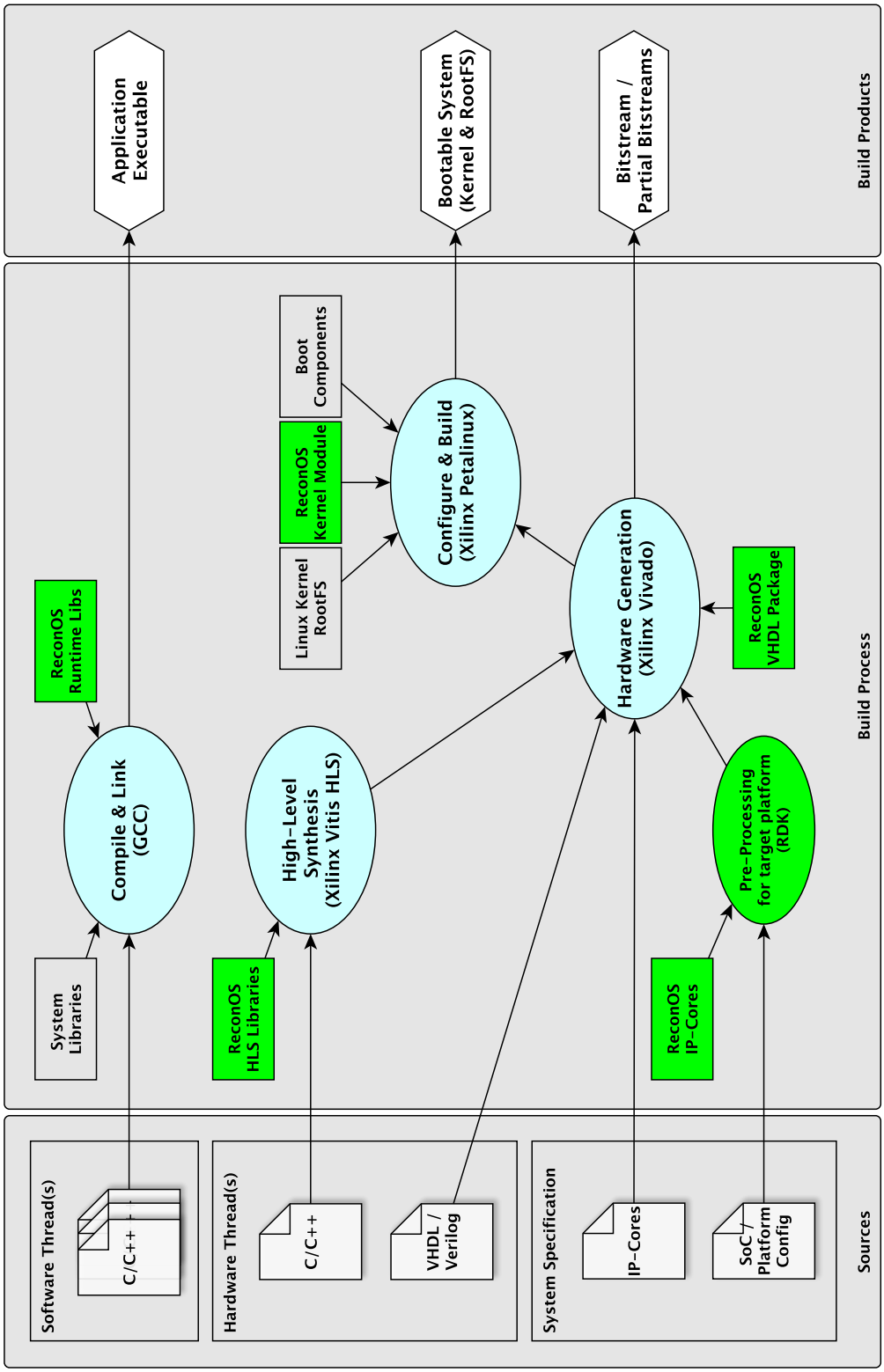

Figure 3.10: ReconOS64 application and system build flow illustrating the process for threads, system and hardware design. Blue items are individual vendor tools. Green items are ReconOS64 components. First presented in [25]. Adapted from [1].

holding the static ReconOS64 elements. The device tree entries for the memory-mapped components are included, so they are present in the system device tree at static addresses and can be looked up when loading the kernel module. The PetaLinux toolchain controls the build process for the ReconOS kernel module and ensures its compilation using the respective kernel sources. The module will be built and included in the system image generated as an output product. The kernel module is not configured to be loaded automatically at boot time since it requires a bitstream containing a ReconOS64 hardware design to be loaded into the PL. Since this might not be the case at all times, the module instead has to be loaded manually once the FPGA is configured and the hardware design is, therefore, present.

The use of PetaLinux replaces a set of manual processes used by previous ReconOS, thus reducing the overhead of creating and regenerating the host system. Graphical configuration utilities for the basic image configuration, the kernel, and the root file system exist, so all configuration steps, for example, for adding individual Linux components, can be accomplished using a guided editor. In addition, the use of network-based boot methods, such as including the root file system via Network File Share (NFS) protocol, can be selected, thus enabling faster prototyping turnaround times by reducing the overhead for creating boot media storage cards.

After the base system is built, there is usually no necessity to recreate the overall image to reflect application design changes on either hardware or software side. As long as the kernel module IO signatures remain untouched and the addressing layout, which is reflected in the device tree, is not changed, the PetaLinux application does not have to be updated between builds.

3.5.2 *Ubuntu Build Flow*

In contrast to the PetaLinux build flow, which originates from a hardware design template, a second and different approach for building the ReconOS64 system using a preexisting boot image was created as a supervised student project work at Paderborn University.

For a number of AMD development boards, including the ZCU102 and 104 board, pre-built Ubuntu images are provided for download [30]. The images are based on the current Long Term Support (LTS) release of the operating system. The images can be extracted to a memory card and provide a ready-to-run operating system for the respective device, including all boot components, which makes them an interesting quick-to-run option for integrating ReconOS64 on these supported platforms. The ReconOS64-specific components are defined as a device tree overlay using the generic `fpgamanager` provided by Linux [29], so the device nodes can be appended to the default device tree once the bitstream including the base ReconOS64 design is

loaded into the PL. A device tree overlay [39] is used to add nodes to a Linux kernel device tree. The ReconOS64 device tree overlay appends the device tree nodes for the OSIF, OSIF interrupt controller, and `proc_control` devices. A device tree entry for those nodes contains information regarding their physical AXI bus address and interrupts. These properties are read by the ReconOS64 kernel module during initialization to find the individual hardware modules' interfaces to connect. Since no system build tool is used for the Ubuntu-based flow, all non-standard extensions, such as the ReconOS64 kernel module, have to be manually compiled against the respective distribution's kernel sources.

The Ubuntu-based tool flow is still experimental and not intended as a replacement for the PetaLinux flow since it relies on the availability of pre-built images for any target hardware platform. Especially when using custom or uncommon devices or hardware configurations, no preexisting image is available, thus reducing the applicability of this approach to only a small set of boards. In contrast, the PetaLinux approach offers more flexibility with respect to the platform adaption and image configuration. However, for boards that are supported by Ubuntu images, this method adds a faster second way to integrate ReconOS64 without needing to build the kernel and root system first.

3.6 WORKLOAD ADAPTION BY PARTIAL RECONFIGURATION

ReconOS64 enables options for PR throughout the build flow for replacing individual hardware threads within a design at runtime. Partial reconfiguration at runtime can be used in different scenarios, for example, if an FPGA's utilization should be increased by offering acceleration to a number of different applications. In this case, the functionality of individual hardware threads can be exchanged at runtime, while the main application acts as a manager distributing the workload and preparing reconfigurable slots with requested tasks, similar to spawning tasks in software. Another use case can be to switch between multiple accelerators for the same function, which are optimized to different metrics, such as efficiency vs. latency. Depending on the current workload and power budget, different designs can be loaded into a reconfigurable slot.

This section focuses on the system implementation of the reconfiguration as well as the general build toolchain aspects. A more in-depth description of the layout considerations and control option for system designers is provided in Chapter 4.

3.6.1 *Partial Reconfiguration in ReconOS64 Projects*

If a ReconOS64 project has the flag for partial reconfiguration set inside its configuration file, the build process treats all hardware threads as separate entities besides the overall base hardware design. The static full base design, comprising the MEMIF, OSIF, and clock hardware elements, is created in a similar way as for non-reconfigurable projects. The hardware threads are, however, not directly integrated into the system design. Instead, empty placeholder hardware threads using a default OSFSM and all MEMIF and OSIF ports are inserted into the slots while the base system design is synthesized and implemented. During individual tool runs, the hardware threads are subsequently implemented as partial bitstreams. This way, the generated products from a PR flow include the non-partial bitstream for the base design, as well as individual partial bitstreams for the hardware threads. The generation of all output products is transparent from the thread implementation and requires no further configuration. From a developer's perspective, the only required setting to enable the use of partial reconfiguration is the declaration of the PR flag within the project configuration. This allows for a gradual development process, starting with a single-bitstream design, which can later be refined to include the ability to partially exchange hardware threads at runtime.

3.6.2 *Partial Bitstream Loading*

The Programmable Logic (PL) can be reprogrammed by loading either a full or partial bitstream through one of two available types of configuration ports. The Processor Configuration Access Port (PCAP) can be targeted from the PS side, while the Internal Configuration Access Port (ICAP) can be embedded as a tile within a hardware design.

The ICAP port offers higher transfer speeds, thus reducing the reconfiguration time, but cannot be accessed from the operating system software without adding custom logic. Therefore, the ReconOS64 application defaults to utilize the bitstream loading capabilities of the Linux `fpgamanager`, which internally accesses the PCAP interface, resulting in measured configuration times of 195ms for a full ZU7EV MPSoC bitstream on a ZCU104 development board [25, Tab.II].

In order to enhance the reconfiguration performance, a custom ICAP interface logic with control integration for ReconOS64 was developed as a bachelor's project by Klassen [34], presenting a speedup of 1.75 to 2.76 over a PCAP reconfiguration depending on the bitstream size [34, Fig. 6.3]. Internally, this approach uses the AXI High-Bandwidth ICAP (HBICAP) IP core [68] and extends it by providing methods for buffering and loading individual bitstreams from within an operating system software application. A selection of bitstreams can be held in individual slots of a contiguously allocated system

memory location, which additionally reduces reconfiguration times since the reconfiguration via ICAP can be started by a short trigger signal identifying the offset of the bitstream within the contiguous configuration data memory region. Figure 3.11 presents an overview of the architecture used for this project. The green components were added to control the blue HBICAP core. Red arrows represent the bitstream transfer from its memory location via the AXI bus to the HBICAP. The green arrow of the management block represents the control interface exposed to the AXI bus, which is controlled from the Linux user space via a kernel module.

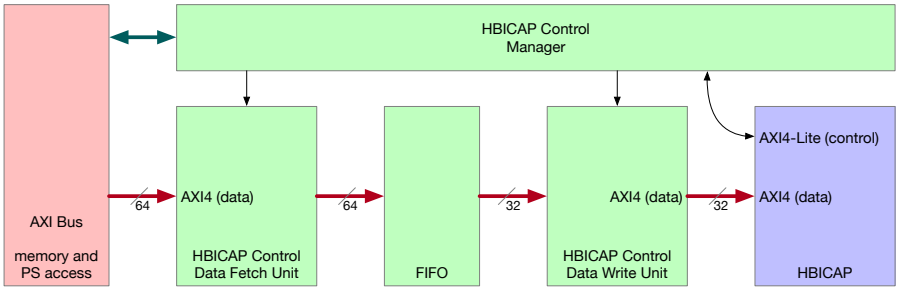

Figure 3.11: HBICAP Manager architecture. Based on [34].

Using the ICAP port imposed additional restrictions for partially reconfigurable systems. Since the ICAP is located in a fixed location, the surrounding region has to be statically programmed and should contain any interfacing logic. The respective hardware inventory can no longer be used as a *pblock* region. During the development of this approach, ZyPR [22] was published, which also provides a memory-mapped ICAP interface and is an extended port for current UltraScale+ systems based on ZyCAP [64], a previous work targeting 7-series Zynq SoC devices.

3.7 RECONOS64 RUNTIME PERFORMANCE MEASUREMENTS

The ReconOS64 system was evaluated as presented in [25]. The evaluation can be separated into the OSIF-centered measurements for system call roundtrip latencies, as well as the MEMIF-focused measurements of read and write operations between main memory and hardware threads. The ReconOS64 measurements are recorded on a ZCU104 board providing a ZU7EV UltraScale+ MPSoC with its ARMv8-A processing system. In order to provide a baseline, additional measurements were also taken with ReconOS in its 32-bit version on a Zedboard development board, which provides a Z-7020 SoC from the previous 7-series.

3.7.1 *OSIF Evaluation*

The OSIF component forwards operating system calls originating in a hardware thread through its communication system toward the delegate threads that represent the respective hardware thread facing toward the operating system. In addition to bringing POSIX-defined synchronization methods across the hardware-software boundary, the OSIF additionally implements a system for passing messages between hardware and software threads. In order to evaluate the latencies arising from the forwarding mechanism, a set of hardware and software threads was created for both the ReconOS as well as ReconOS64 system that interacts with a common subset of operating system calls as well as sending messages via the `mbox` messaging system. The selected methods include locking, unlocking, and trying to unlock a mutex; semaphore setting and reading; and sending and receiving messages.

During the measurements of both the ReconOS64 and ReconOS systems, the static system part, including the OSIF buffer and further IP cores, is set to run at 200MHz, while the hardware thread generating the requests is set to run at 100MHz. The processing system of the Z-7020 runs at 667MHz, while the ZU7EV ARM processor defaults to 1.2GHz.

```
initialize semaphore value > 0
for N:
    mbox_put(random data)
mbox_put(num_iterations, 1)
mbox_get({post,wait,mget,mput,lock,ulck,tlck}_cycles_sum)
mbox_put(num_iterations, 100000)
mbox_get({post,wait,mget,mput,lock,ulck,tlck}_cycles_sum)
```

Listing 3.1: OSIF evaluation main application

```
loop:
    N <= mbox_get(num_iterations)
    for N:
        post_cycles_sum += cycles( semaphore_post(...) )
        wait_cycles_sum += cycles( semaphore_wait(...) )
        mget_cycles_sum += cycles( mbox_get(...) )
        mput_cycles_sum += cycles( mbox_put(...) )
        lock_cycles_sum += cycles( mutex_lock(...) )
        ulck_cycles_sum += cycles( mutex_unlock(...) )
        tlck_cycles_sum += cycles( mutex_trylock(...) )
    mbox_put({post,wait,mget,mput,lock,ulck,tlck}_cycles_sum)
```

Listing 3.2: OSIF evaluation hardware thread

The hardware thread calls the operating system functions on variables and within a sequence, which ensures that each call can immediately be served and no blocking condition can arise. For example,

a semaphore that should be passed is ensured to be available, or a message queue to be read from holds a message before entering the test. Listing 3.1 shows the main application, starting with initializing a semaphore to a value that would not cause a block during a subsequent *semaphore_post*. After that, the desired number of iterations is sent via a message, first starting with a single iteration to eliminate possible overheads before issuing the number of 100,000 consecutive runs of the entire evaluation set. After the evaluation is finished, a set of messages is received, holding the accumulated cycles for every type of OS call evaluated. Listing 3.2 shows the associated hardware thread behavior, starting with a blocking call to receive the number of iterations. If an iteration count is received, the state machine advances and executes the individual OS calls. For each call, the time between the call issued and the *done* signal received is recorded and added to a cumulative cycle count for all $N = 100,000$ executions of the respective call. After finishing the given number of iterations, the accumulated cycle counters are returned to the main application via individual message boxes.

Due to the number of elapsed clock cycles being counted until an acknowledging signal is received, the measurements include some additional overhead for the acknowledge signal to travel back to the hardware thread, thus giving a realistic estimation from the thread's perspective while providing a pessimistic latency value relative to the moment the action is triggered inside the operating system.

Figure 3.12 shows the results for the test set of operating system calls, compared between the ReconOS and ReconOS64 measurements on the Z-7020 or ZU7EV SoC, respectively.

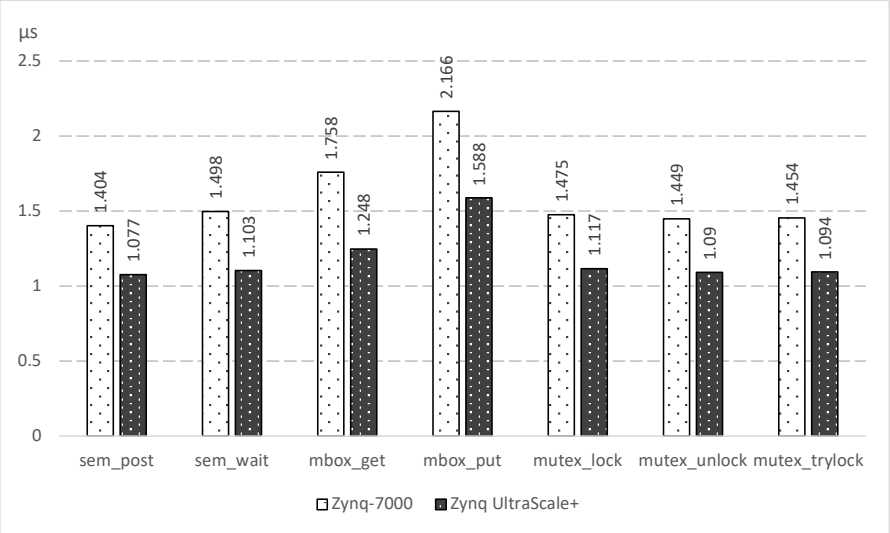

Figure 3.12: OSIF call and message forwarding roundtrip time comparison between 7-series Zynq Z-7020 SoC and 8-series Zynq UltraScale+ ZU7EV MPSoC. First presented in [25].

Table 3.1: ReconOS64 OS calls for evaluation on a ZU7EV device. Providing respective POSIX API calls for reference. First presented in [23].

ReconOS64 Call	POSIX Equivalent	Blocking	avg. time in µs
`mbox_put` (8B)	similar to `mq_send`	no	1.588
`mbox_get` (8B)	similar to `mq_receive`	yes	1.248
`mutex_lock`	`pthread_mutex_lock`	yes	1.117
`mutex_unlock`	`pthread_mutex_unlock`	no	1.090
`mutex_trylock`	`pthread_mutex_trylock`	no	1.094
`sem_wait`	`sem_wait`	yes	1.103
`sem_post`	`sem_post`	no	1.077

Since the message boxes are used in many cases to transfer memory pointers, the ReconOS64 system adapts its payload size to fit the pointer size of 8 bytes for 64-bit systems. Comparing the increased data size to the call latencies shows that despite the increased payload size, the transfer time decreased. While part of this is to be expected due to the higher processing system clock speed, the overall throughput is increased.

The second set of measurements was entirely conducted on the ReconOS64 system on the ZCU104 board, aiming at comparing the call roundtrip latency when being executed from within a hardware thread to the latency for the same call executed directly from within a software thread. Therefore, the same set of operating system calls was executed from a software thread; the results are shown in Figure 3.13.

The measurements show that the hardware-issued operating system calls have a higher roundtrip latency compared to the software-issued calls. This meets the expected results since the ReconOS64 system can only forward the calls to the respective delegation thread, which then, in turn, executes the calls as a placeholder for the hardware-bound thread implementation. Since the very same operating system call, therefore, has to be called by the delegate thread after the request was forwarded, and a possible result has to be returned back through the OSIF mechanism to the hardware thread, the delegated calls can never be faster than the software-issued requests. Based on the hypothesis that the operating system calls issued from a delegate thread show the same latency behavior as calls issued from the measured software thread, both averaged latency values can be used to determine the overhead created by the delegation of a call, thus showing the operating system call penalty for a thread placed in hardware. As shown in Figure 3.13, the overhead is 0.58µs on average, peaking at 0.78µs for inserting a message into a message box. These overhead numbers are within the order of magnitude of the calls themselves, showing

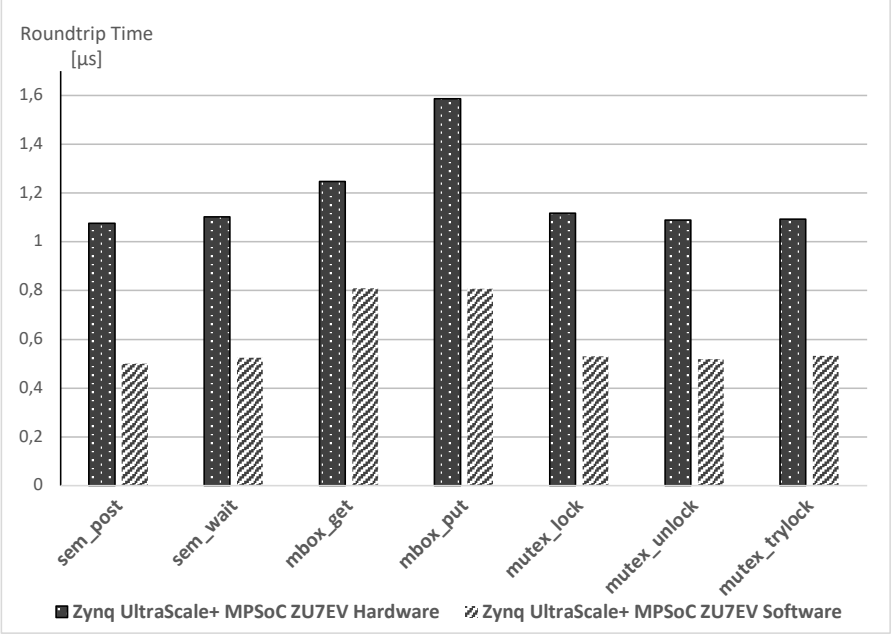

Figure 3.13: OSIF call and message roundtrip times compared between calls issued from within hardware or software threads. First presented in [25].

that operating system calls can effectively be delegated across the hardware-software boundary with reasonable overhead latencies.

3.7.2 *MEMIF Evaluation*

The MEMIF system enables memory access via virtual addresses by translating the requests using an MMU and rewriting the requested address to its physical location before executing the read or write requests to the memory system via the AXI bus.

The transfer speeds for memory requests were evaluated using a hardware thread reading from main memory to a local BRAM memory, subsequently writing the data back into main memory while counting the clock cycles required for both operations. The software component of the evaluation setup consists of a thread allocating and populating memory locations and transferring the virtual address pointer to the hardware thread using a message box. After the hardware thread has read the data and stored it in its local BRAM-located memory, it starts writing back its content to the main memory. This process is repeated for the configured number of 1024 iterations. All read and write accesses are conducted over a number of different data sizes to evaluate the impact of small transactions in contrast to transactions exceeding the page or burst size. Figure 3.15 presents the sequence of operations for evaluating a hardware thread using a single block size. After the last iteration has finished, the software thread verifies the

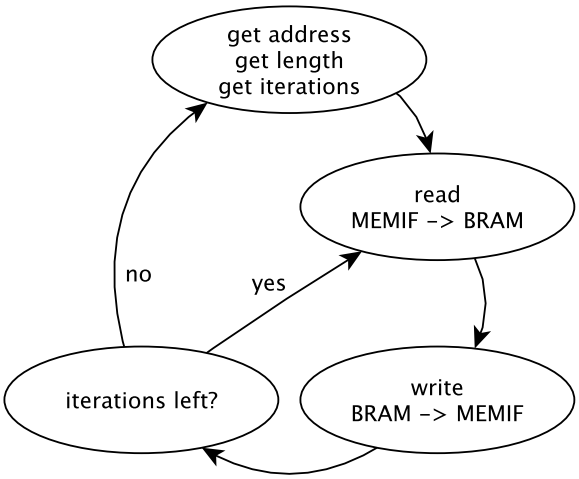

Figure 3.14: MEMIF evaluation hardware thread state machine. First presented in [25].

written content and receives the cycle counts for reading and writing as recorded and averaged by the hardware thread.

Figures 3.16 and 3.17 present the results recorded for reading and writing in a setup comprising one hardware thread running at 200MHz with a MEMIF system set to the same clock speed with a varying amount of data transferred as presented in [25]. The measurements are executed on a 7-series Zynq Z-7020 device running ReconOS, as well as on a Zynq UltraScale+ MPSoC running ReconOS64. During the move from the 32-bit to a 64-bit platform, the default data size was also increased from 32 to 64 bit. This includes the size of message boxes but also impacts the amount of data transferred using a single MEMIF operation. Due to this increase, the naive expectation should be a doubled data throughput on the ReconOS64 platform, which can only be shown for write operations. Especially considering the read operations, despite the faster transfer speeds achieved on the ReconOS64 system, the main limitation is based on the overhead created by the frequent start of new AXI transactions, thus limiting the achieved overhead during this set of measurements to only a slight improve over the previous series-7 systems. The following section will examine changes to the AXI memory access pattern in order to improve the overall MEMIF performance.

3.7.3 *MEMIF Burst Access Optimization*

The previous measurements were executed with a maximum burst size of 16 elements, which was the default for ReconOS. This means that no more than 16 elements of the configured AXI bus width can be transferred within a single transaction. The default bus width is

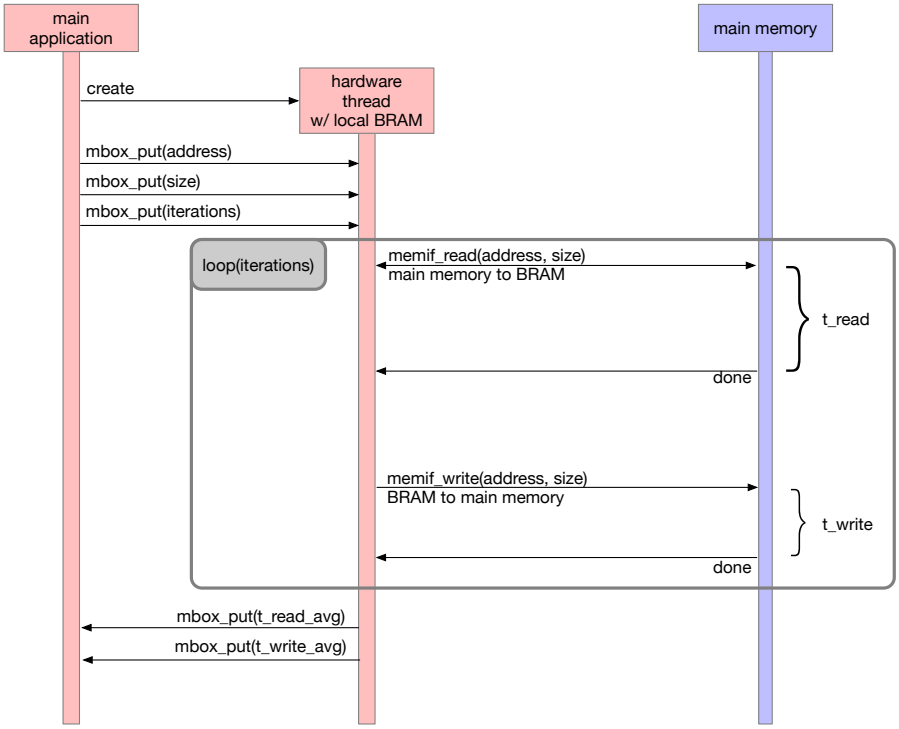

Figure 3.15: MEMIF evaluation sequence diagram for a single hardware thread and a single read and write block size.

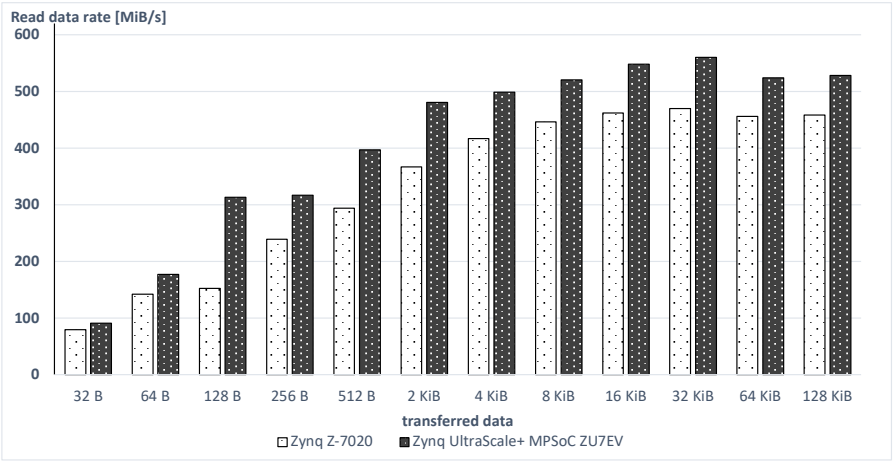

Figure 3.16: MEMIF read data rate comparison between 7-series Zynq and 8-series Zynq UltraScale+ devices. First presented in [25].

32-bit for the Zynq Z-7020 and 64-bit for ReconOS64 on the Zynq UltraScale+ MpSoC ZU7EV devices. The limitation to 16 elements is the maximum possible burst size for AXI3 connections [13]. This section describes how increasing the burst size can increase the memory access throughput for larger consecutive accesses.

To estimate an upper limit of the transfer data rate for read accesses, it is assumed that the memory controller provides one data beat during each clock cycle. Considering the given configuration of the MEMIF

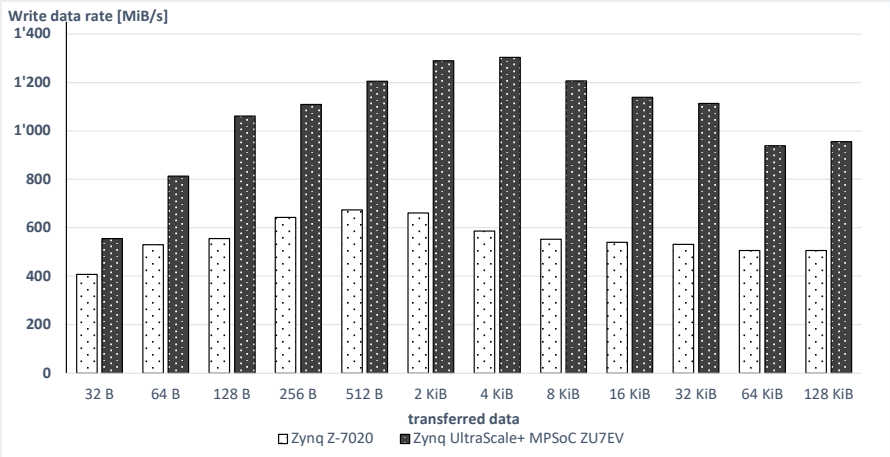

Figure 3.17: Comparison of MEMIF write data rates between 7-series Zynq and 8-series Zynq UltraScale+ devices. First presented in [25].

clock running at 200MHz and the data size to be transferred being 64 bit, the maximum theoretical rate can be calculated as shown in equation 3.1.

$$\max(R_{MEMIF}) = 8B \cdot 200MHz = 1600MB/s \approx 1525MiB/s \quad (3.1)$$

A more realistic assumption for the expected read data rate is lower than the previous estimation due to two limiting factors. First, the maximum possible amount of elements per AXI transaction, considered as *burst length*, is limited at 256 as per the AXI4 standard [13]. For the AXI data width of 64-bit as used by ReconOS64, this limits the maximum amount that can be transferred within a single AXI transaction to be $256 \cdot 8B = 2kiB$.

Figure 3.18 presents ILA traces of a MEMIF read transaction of 1024 64-bit words, resulting in 8kiB overall data. After 256 elements or 2kiB, a new AXI transaction has to be issued, creating an overhead of 30 cycles. Additionally, the address of each memory page needs to be translated by the MMU since contiguously addressed virtual memory locations can be split across different, non-contiguous memory locations. Therefore, for the default Linux page size configuration, an additional MMU translation inserting three consecutive read requests to the page table has to take place every 4kiB, creating an overhead of 75 cycles.

Both overheads can be seen in the waveform. The overall data to be transferred is being split into four individual AXI transactions of 256 elements of 8 Byte each, additionally interrupted by a second MMU page address translation after the first 4kiB. The TLB is empty, so the translation process cannot use any buffered results. The first two AXI transactions read 4kiB with 640 clock cycles passing between the first MMU page table address being sent until the *last* flag is received

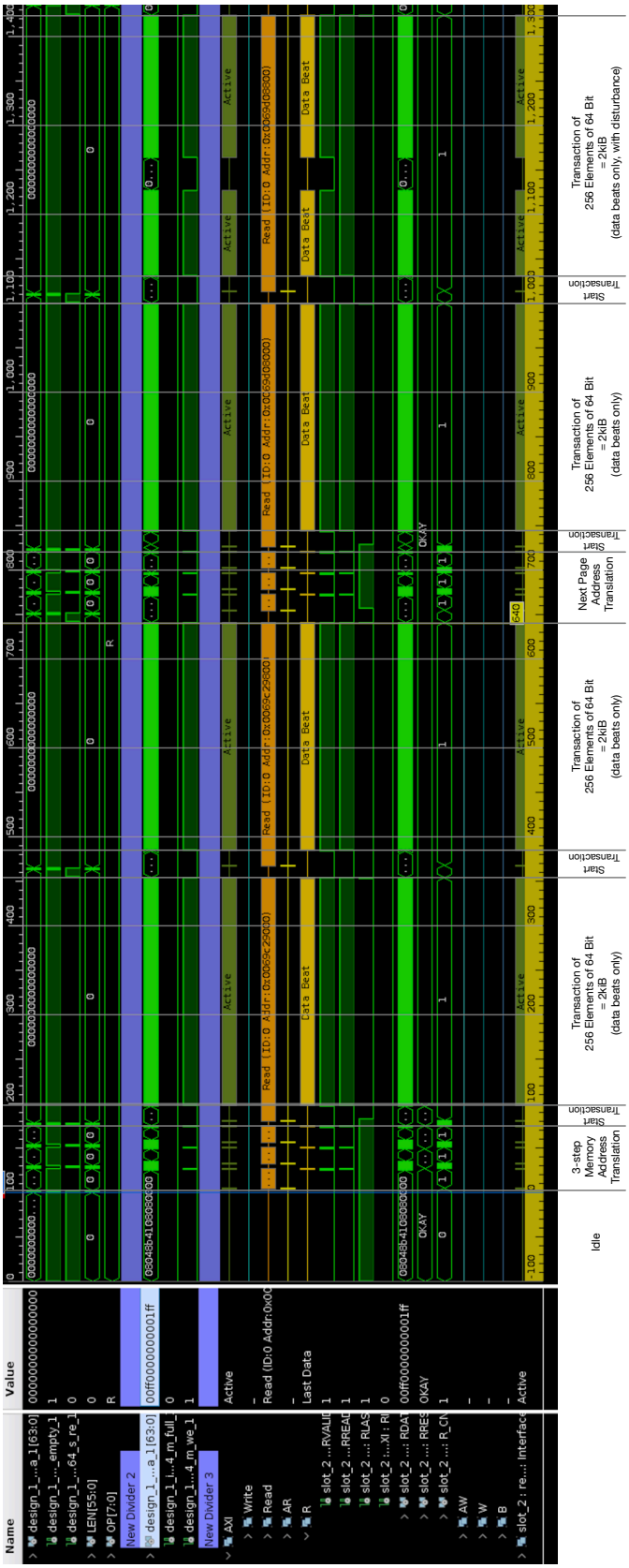

Figure 3.18: Vivado Integrated Logic Analyzer (ILA) waveform of a single MEMIF read instruction transferring 1024 words of 64 bits each. The read access is split into four segments of 256 elements due to the AXI burst limit. After the first two bursts, a new page table walk translates the following page's address.

on the AXI bus, signaling the last data word is present on the data line. During the second page transfer, the transaction is halted for some cycles in addition to the expected overheads, possibly due to the memory interface being unavailable for that period of time. The traces of the *AXI-Read* activity row additionally represent the three short single-byte read requests with their respective addresses and data responses for the three-step page table walk preceding the read request for each page.

Figure 3.19 shows measurement results from 1024 consecutive read accesses with varying data sizes. Lower data size transfer speeds are slower on average due to the added overhead of the transaction setup and memory translation at the beginning of a transfer. The roughly constant maximum read throughput is reached starting at the maximum burst length of 256 elements, so all consecutive increases of the transfer size inevitably result in the splitting of the accesses, thus imposing a fixed minimum overhead amount. The measured maximum speeds for the runs between 4kiB up to 256kiB result in an average data rate of 1219,8MiB/s, which results in an overhead of 20% compared to the optimistic estimation neglecting all overheads as presented at the beginning of this section.

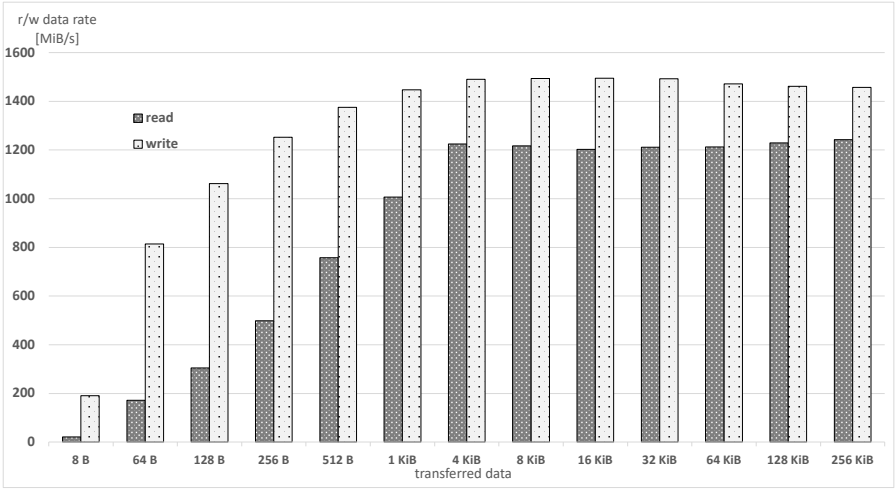

Figure 3.19: MEMIF read and write data rates on a ZU7EV Zynq UltraScale+ MPSoC device at 200MHz using an AXI burst size of 256 elements.

The observed write data speeds are higher and closer to the expected maximum possible speed for this AXI burst configuration. This is mainly due to the fact that there is no additional waiting time as in the reading process. For reading, after the address is specified to the AXI bus, the memory access path needs additional time until the data is returned. In contrast, for writing requests from a hardware thread's perspective, the write command, address, length, and data can be sent immediately after each other with the MEMIF FIFO buffering the data. In addition, the data is written back to the exact memory location from which it was read. Therefore, the initial address translation is already

available in the TLB of the MMU, which affects the first iteration of the test by removing the otherwise required address translation lookup.

3.8 RECONOS64 HARDWARE UTILIZATION

Table 3.2: Resource utilization of the memory access performance benchmark (4 HWT, 200MHz, ZCU104 board). Presented in [25].

Component	CLB LUTs	CLB Reg.	CLB
MEMIF System	1342	1608	433
MEMIF FIFOs	1800	384	399
OSIF, Clock, Control	856	822	306
ReconOS64 Total	3998	2840	1138
% of ZU7EV	1.74%	0.61%	3.95%
ZU7EV Total	230400	460800	28800

Table 3.3: Resource utilization of the memory access performance benchmark (4 HWT, 200MHz, ZCU104 board) with optimization.

Component	CLB LUTs	CLB Reg.	CLB
MEMIF System	1366	1612	470
MEMIF FIFOs	3373	536	600
OSIF, Clock, Control	884	1077	322
ReconOS64 Total	5623	3225	1392
% of ZU7EV	2.44%	0.70%	4.83%
ZU7EV Total	230400	460800	28800

Tables 3.2 and 3.3 show the resource utilization of a ReconOS64 memory test application consisting of four hardware threads on a system configured to run at 200MHz. Table 3.2 was first presented in [25] with additional separations for some hardware components irrelevant to this comparison. As shown in Table 3.3, the optimized MEMIF system, resulting in an overall better read and write throughput, increases the resource requirements of the system, thus reducing the resources available to hardware threads. Nearly all of this impact originates from the increased FIFO sizes to accommodate the additional words in order for them to be transferred via a single AXI transaction. Compared to the number of available logic resources, this

is, however, still a minor increase in logic utilization compared to the performance advantage resulting from the optimization.

A central factor for the overall logic utilization of the MEMIF components is the configuration of the TLB within the ReconOS64 MMU. Throughout the previous measurements, the TLB was constantly set to a size of 16 entries, which means that the buffer stores the latest 16 addresses that were translated using page table walks. Assuming an application that utilizes a lot of consecutive memory, this means that for 4kiB pages, a maximum memory address region of 64kiB can be buffered within the TLB.

Table 3.4: Resource utilization of the memory access performance benchmark (4 HWT, 200MHz, ZCU104 board) with optimization, but increased TLB size (256 translations).

Component	CLB LUTs	CLB Reg.	CLB
MEMIF System	9142	17163	2891
MEMIF FIFOs	3374	536	622
OSIF, Clock, Control	885	1077	311
ReconOS64 Total	13401	18776	3824
% of ZU7EV	5.82%	4.07%	13.28%
ZU7EV Total	230400	460800	28800

Table 3.4 shows the utilization of a MEMIF configuration similar to 3.3 but with a TLB size configured to hold 256 address translations. Despite more than doubling the required logic inventory, the TLB can store the translation for 1MiB of memory, which is relatively small compared to the increase of logic required for this TLB. Concluding from this observation, the key to enabling low-overhead MEMIF structures that buffer a decent amount of memory address space is to utilize the option for larger pages. The ReconOS64 MMU supports 64kiB granularity for page table address translations as described earlier in this section, thus effectively bringing the buffered address translation size to 1MiB for a TLB size of 16 elements while keeping the lower resource utilization similar to Table 3.3.

4

RESOURCE MANAGEMENT FOR PARTIALLY RECONFIGURABLE SYSTEMS

With the increasing capacity of reconfigurable systems, which are able to hold many accelerators, and the option to reprogram individual parts of an SoC, the demand for fine-grained management of its resources and individual sections increases. Adapting to varying workloads by increasing or lowering the number and type of available hardware threads by thread management and partial reconfiguration is enabled by the ReconOS64 system. In addition, the clock management of individual reconfigurable slots allows for an adaptive system reacting to the respective clock requirements of different hardware thread types.

While the previous chapter focused on the internal data access paths and platform-specific functions inside ReconOS64, this section changes the perspective and presents possible management mechanisms relevant to application development.

4.1 RECONFIGURABLE REGIONS

In order to enhance flexibility when using partial reconfiguration as described before, the ReconOS64 build system allows for separating the logic inventory of Hardware Threads (HWTs) into multiple reconfigurable regions or *slots*. A reconfigurable slot is defined as a contiguous portion of logic cells within the FPGA that can hold a single hardware thread. Its outline can be defined by referring to the respective vendor-predefined *clock regions*, separating the logic cells into a two-dimensional grid referenced by X and Y indices. The ZU7EV Multi-Processor System-on-Chip (MPSoC) as used on the ZCU104 board contains 20 individual clock regions [25, Tab.II]. The second option for defining reconfigurable slots is to specify their logic inventory, explicitly stating the two-dimensional borders for regions of slices, DSP, and RAM cells. Internally, clock regions or logic sections are used to define multiple *pblocks* [50] within the Vivado design project.

For the purpose of assigning hardware threads to reconfigurable regions, the concept of slot groups is introduced by ReconOS64. A slot group is an intermediate structure in order to ease the assignment process in both directions. Towards the hardware threads, the slot

group acts as a single point of target, thus targeting a group of reconfigurable slots instead of listing individual pblocks. A ReconOS64 hardware thread can be assigned to precisely one slot group.

Towards the hardware regions, a slot group combines multiple regions into a single structure that can be targeted by a hardware thread's implementation. All reconfigurable slots of a slot group should be of roughly the same size in order to allow the maximum utilization of all comprised FPGA inventory. If the reconfigurable slots vary in size, bigger threads might not be able to be placed into each reconfigurable slot of a slot group, which would prevent the design placement from completing. Similarly, even if a group of reconfigurable slots of varying sizes can hold all assigned hardware threads, all logic cells exceeding the inventory of the smallest reconfigurable slot are wasted and, therefore, decrease the overall maximum possible utilization, degrading the spatial efficiency.

The minimum group size, i.e., the number of regions contained in a group, is one. This allows for targeting individual regions for a specific thread as if no concept of slot groups would be in effect at all. If only a single hardware thread is assigned to a group containing a single region, only one partial bitstream for the thread-region combination is created.

When assigning a hardware thread to a slot group consisting of multiple regions, the scripted Vivado build flow generates individual partial bitstreams for the hardware thread at each reconfigurable slot of the respective group. The generation of individual partial bitstreams is required even for multiple uniform reconfigurable slots since the partial reconfiguration concept currently does not allow for arbitrary placement of partial bitstreams. The relocation of bitstreams is currently an ongoing field of research [43, 56], but since the ReconOS64 design process utilizes the default vendor tools, currently no movement of bitstream contents is possible, thus requiring individual bitstreams for each location.

Figure 4.1 presents an overview of the reconfigurable region concept in ReconOS64. The left side resembles the contents of the project configuration file. The upper part refers to the reconfigurable slot definition, defining two slot groups of two individual slots each, outlined via their respective clock region coordinates. Slot group A holds two reconfigurable slots of 2x1 clock regions each, while the second group, slot group B, contains two reconfigurable slots of 4x1 clock regions each. As recommended before, the slot groups, defined by their clock regions, each contain two reconfigurable slots of similar respective sizes of 2x1 and 4x1 for optimizing the overall hardware utilization.

The lower half of the left side represents the mapping of hardware threads to the slot groups. The uniform external interfaces of each reconfigurable slot allow for multiple different hardware threads to be

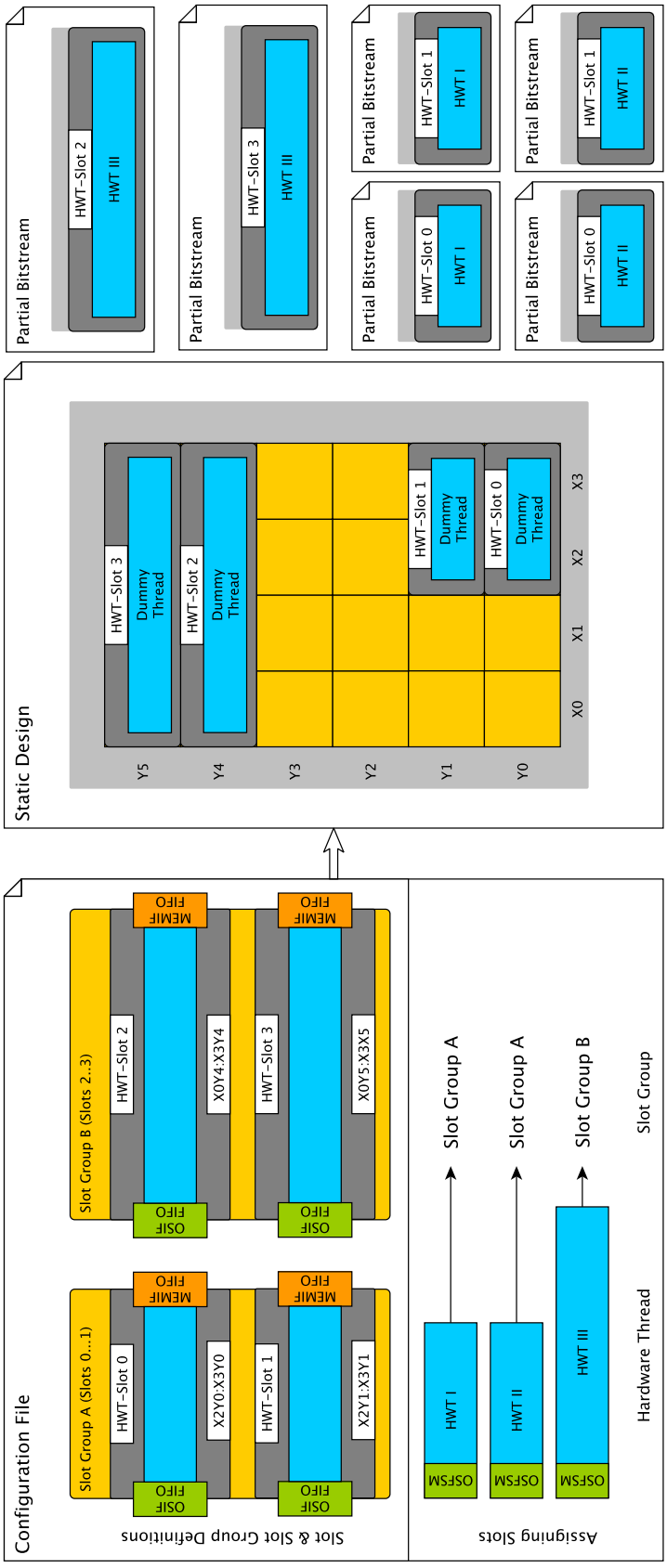

Figure 4.1: ReconOS64 reconfigurable regions concept, showing the configuration options on the left and the produced output on the right. First presented in [25].

assigned to a slot group, so the functionality within a reconfigurable hardware slot can be exchanged at runtime by loading the respective partial bitstream. In this example, the two hardware threads, HWT I and HWT II, are both assigned to slot group A. In total, a slot group of $r = 2$ regions, having $t = 2$ hardware threads assigned, results in a total number of $r \cdot t$ partial bitstreams being created.

The total number of bitstreams for an overall design containing a set $G = g_1, g_2, ..., g_i$ of slot groups, therefore, is $\sum_i t_i \cdot r_i$, where t_i is the number of different hardware threads assigned to slot group i, while r_i is the number of respective hardware regions of slot group i. For a large number of reconfigurable slots, groups, or threads, this underlines the positive impact a possible future availability of relocatable partial bitstreams might bring, reducing the overall number of required bitstreams down to $\sum_i t_i$.

The right side of figure 4.1 represents the output products generated after running the ReconOS64 build flow. The upper section shows the complete bitstream, including the overall static portion, as well as the individual hardware thread interfaces. In order to focus on the reconfigurable slots, static system sections such as the Operating-System Interface (OSIF), Memory Interface (MEMIF), clock and control cores are omitted. All reconfigurable slots defined by the configuration file are filled with a dummy thread during the static design generation step. The dummy hardware threads do not include any specific HDL logic but instead consist of an empty Operating System Finite State Machine (OSFSM) and all ports required for a hardware thread. This concept works for all hardware threads used throughout this thesis; however, it creates specific implications from a designer's perspective since the designated hardware threads are not present during the timing and energy analysis of the overall system. If a more precise energy analysis is required, or if the designated hardware task should be included in the timing analysis, the dummy task has to be altered instead to include the desired hardware description of the later task.

All hardware threads of a clock group share the same clock output of the ReconOS64 clock module, thus ensuring all reconfigurable slots of a slot group offer the same conditions for a thread to be executed while at the same time keeping the overall amount of different clocks limited. This approach still offers the flexibility to adjust the clock speeds for each specific workload assigned to a slot group.

Limiting the clocks to one clock per slot group is required due to the small overall amount of clock management tiles, limiting the number of clock outputs. Therefore, the intended granularity of clock speed adjustments should be taken into consideration when defining the number of slot groups for a hardware design. The clocking system and its details will be described in the following section.

4.2 CLOCKING ON MULTI-ACCELERATOR DESIGNS

Especially in hardware designs incorporating multiple different types of accelerators as HWTs, the maximum operating frequency for each thread implementation might vary. A second scenario where individual operating frequencies are relevant includes multiple hardware threads within a system having real-time constraints that require individual minimum clock speeds to meet the thread's respective deadlines. The ReconOS64 configuration allows for defining individual clocks for every slot group. Internally, the system uses a Mixed-Mode Clock Management (MMCM) tile provided on Zynq UltraScale+ devices. Each MMCM tile allows for generating seven individual clocks with different options for regulating the respective clock speeds [2]. The external input clock can be adjusted by a common pre-divider D and common multiplier M. Each output can be divided by an individual integer divider O or by a fractional divider for some outputs, resulting in an output frequency as shown in Equation 4.13.

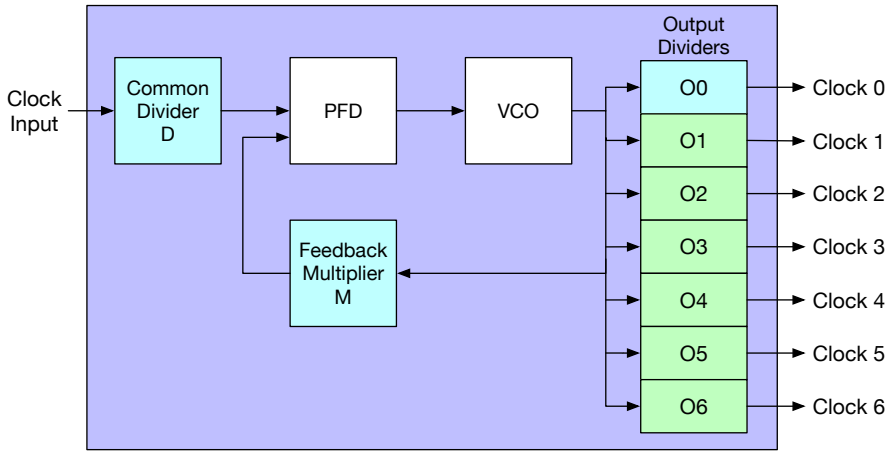

Figure 4.2: MMCM tile clock generation parameters. Adapted from [71].

MMCM tiles were introduced to ReconOS64 due to their ability to generate multiple clock outputs with low-downtime runtime reconfigurable output frequencies, which were not available for the previous clock generation method using PLL tiles. The first output clock of the MMCM tile is statically tied to the ReconOS64 internal components, such as MEMIF and OSIF logic, and will not be changed at runtime. The remaining six clock outputs can be set at individual clock divide rates and will be tied to the desired slot group during the build process.

Figure 4.2 presents the parameters involved in generating the output clock signals. White elements resemble the respective hardware components of the frequency synthesis, which impose certain limitations in their operating condition range, as described in the following section. Blue parameters are fixed by ReconOS64, while green ones can be reconfigured at runtime.

4.3 RUNTIME CLOCK RECONFIGURATION IN RECONOS64

The clock periods of each individual output of the MMCM tile can be set and reconfigured at runtime using the Dynamic Reconfiguration Port (DRP) of a respective tile [71]. The Clock Divide Dynamic Change (CDDC) option available via the DRP allows for each output divider to be set without resetting the overall tile, so the downtime during the reconfiguration only affects the single output to be modified. The MMCM DRP includes addressing and data lines, enabling writing to and reading from a set of 16-bit configuration registers. The ReconOS64 clock module connects to those data access ports and embeds the MMCM, including its DRP, into additional HDL logic to offer a programming interface toward the ReconOS64 runtime.

Internally, the MMCM tile consists of a Voltage-Controlled Oscillator (VCO) that is coupled to the input clock pre-divided by the input divider D and fed back through a feedback multiplier M as shown in Figure 4.2. Thus, D and M define the VCO frequency as shown in Equation 4.6. All outputs are derived from the VCO via a variable phase tap and an individual output divider O (Eq. 4.13).

$$f_{IN} = 100\text{MHz} \tag{4.1}$$

$$f_{PFD_{MMCM},min} = 10\text{MHz} \tag{4.2}$$

$$D_{max} = \left\lfloor \frac{f_{IN}}{f_{PFD_{MMCM},min}} \right\rfloor = \left\lfloor \frac{100\text{MHz}}{10\text{MHz}} \right\rfloor = 10 \tag{4.3}$$

$$f_{PFD_{MMCM},max} = 450\text{MHz} \tag{4.4}$$

$$D_{min} = \left\lceil \frac{f_{IN}}{f_{PFD_{MMCM},max}} \right\rceil = \left\lceil \frac{100\text{MHz}}{450\text{MHz}} \right\rceil = 1 \tag{4.5}$$

$$f_{VCO} = f_{IN} \cdot \frac{M}{D} \tag{4.6}$$

$$f_{VCO_{MMCM},min} = 800\text{MHz} \tag{4.7}$$

$$f_{VCO_{MMCM},max} = 1600\text{MHz} \tag{4.8}$$

$$M_{min} = \left\lceil \frac{f_{VCO_{MMCM},min}}{f_{IN}} \cdot D_{min} \right\rceil = \left\lceil \frac{800\text{MHz}}{100\text{MHz}} \cdot 1 \right\rceil = 8 \tag{4.9}$$

$$M_{max} = \left\lfloor \frac{f_{VCO_{MMCM},max}}{f_{IN}} \cdot D_{max} \right\rfloor = \left\lfloor \frac{1600\text{MHz}}{100\text{MHz}} \cdot 10 \right\rfloor = 160 \tag{4.10}$$

$$f_{out_{MMCM},min} = 6.25\text{MHz} \tag{4.11}$$

$$f_{out_{MMCM},max} = 667\text{MHz} \tag{4.12}$$

$$f_{out_n} = f_{IN} \cdot \frac{M}{D \cdot O_n} \tag{4.13}$$

Equations 4.1 through 4.13 show the VCO and Phase-Frequency Detector (PFD) parameter ranges together with the clock frequency calculation for each clock output n [4].

For ReconOS64, the desired clock configuration aims at maximum runtime flexibility. Since the CDDC reconfiguration can only change

the output divider values O_n, the VCO is configured to run at its maximum possible frequency to allow for a wide range of resulting output frequencies.

The input frequency of $f_{IN} = 100\text{MHz}$, which is fed from the Programmable Logic (PL) clock generator, is sufficient to meet the PFD criteria according to Eq. 4.2, thus limiting the permitted input divider range as shown in Equations 4.3 and 4.5. No common input divider D is used, as it would reduce the frequency resolution at the upper end of the configurable range, so $D = 1$ is set, which leads to a multiplier setting of $M = M_{max}|_{D=1} = \lfloor \frac{f_{VCO_{MMCM},max}}{f_{IN}} \rfloor = 16$ in order to achieve the maximum possible VCO frequency.

Using the CDDC reconfiguration method imposes an additional restriction on the achievable output frequencies. For each clock output, the configuration register holds a pair of two values of six bits each, as shown in Fig. 4.3, respectively defining a counter value for high and low times relative to the base clock after the multiplier. The values define after how many cycles the output switches from high to low (t_{high}) or low to high (t_{low}), respectively. The length of each field limits both counters to a maximum value of $2^6 - 1 = 63$.

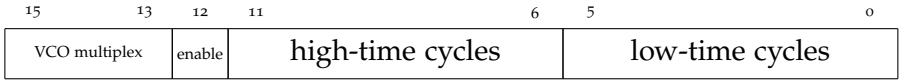

Figure 4.3: Exemplary MMCM register `0x12` for `CLKOUT6` high and low time configuration, as well as enabling the clock out [71].

Since high and low times have to be defined individually and can only hold integer values, the overall cycle length is defined by $t_{high} + t_{low}$ cycles of the VCO at a frequency of $f_{IN} \cdot M$. For a regular clock signal with a 50% duty cycle, both dividers $O_{n,high}$ and $O_{n,low}$ have to be set to match $\frac{O_n}{2}$, which is only possible for integer counter values if using the even values of O_n only.

Table 4.1 lists exemplary divider values and resulting output frequencies $f_{out_n} = \frac{f_{VCO}}{O_n}$ for $f_{IN} = 100\text{MHz}$. Frequencies that would violate any restriction regarding the component's operating ranges or are odd are omitted. The last setting of 6.25MHz equals the minimum allowed output frequency as given in Eq. 4.11. However, this value cannot be reached via CDDC reconfiguration since the required $t_{high} = t_{low}$ divider values exceed the number of available bits.

If the frequency and, therefore, the divider values of any output clock are to be changed, a call to the ReconOS64 runtime from within the application triggers a write to the divider configuration register within the ReconOS64 clock module of the static hardware design part inside the programmable logic. The ReconOS64 runtime receives the desired target frequency, converts it into the floored integer divider $\lfloor \frac{f_{in} \cdot M}{f_{out}} \rfloor = \lfloor \frac{100\text{MHz} \cdot 16}{f_{out}} \rfloor$ and forwards the desired divider to the ar-

Table 4.1: Exemplary divider settings for MMCM tile output dividers suitable for CDDC reconfiguration to reach a 50% duty cycle. $f_{IN} = 100MHz$, $D = 1$. The last value is the theoretical minimum output frequency allowed by MMCM tiles, but it cannot be reached in this configuration due to the cycle count register range limitation.

O	4	6	8	10	12	16	50	256
$t_{high} = t_{low}$	2	3	4	5	6	8	25	128
f_{out} (MHz)	400	266.67	200	160	133.3	100	32	6.25

chitecture code. For even output dividers, $\frac{O}{2}$ is set for both high and low counters, while odd dividers result in asymmetrical high- and low-time counter values, such that the target frequency is still met but with an imbalanced duty cycle. The clock divider counter values are written to the `clock` HDL module, which holds one configuration register of 32 bits per clock configured during project generation. These registers represent the dual 16-bit configuration registers of the `MMCM4_ADV` tile contained within the module.

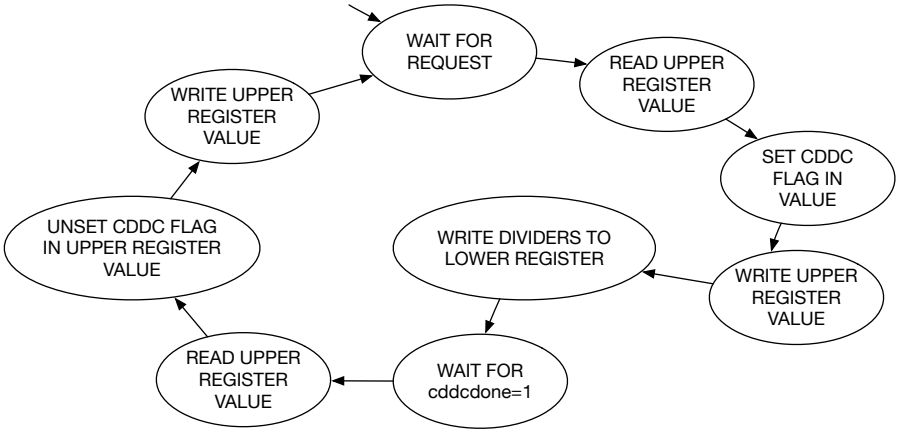

Figure 4.4: State Machine of the clock module for performing the Clock Divider Dynamic Change operation.

Figure 4.4 presents the state machine embedded into the `clock` module for reconfiguration of the output dividers. If a write access is initiated into the register space of the clock module, the selected register offset is used to determine the targeted clock output. A state machine is invoked for the reconfiguration process. It issues a request for a CDDC transaction by setting the CDDC flag of the respective clock's upper register and setting the `CDDCREQ` line, thus effectively stopping the output clock for the rest of the reconfiguration transaction. As a next step, the new t_{high} and t_{low} dividers are written to the lower configuration register, and the CDDC flag is unassigned afterward. After writing the divider data via the data interface and releasing the `CDDCREQ` line, the output will synchronously start again at the next tick, and the `CDDCDONE` line indicates that the process is done

and the output is valid again, which also resets the reconfiguration state machine.

Figure 4.5 shows the logic analyzer recording of a CDDC operation on a ReconOS64 system, focusing on the ports of the MMCM tile. The `mmc_cddcdone` line represents the output of the MMCM tile, signaling the completion of the overall process. The remaining lines are mainly for selecting the component and the hardware handshake for exchanging configuration data. After starting the logic analyzer recording, a ReconOS64 application requests a single clock to be runtime-reconfigured to a new frequency. The request includes a desired clock rate to which the clock should be reconfigured. ReconOS64 calculates the closest frequency achievable given the static parameters $D = 1$ and $M = 16$ as described below. Changing these parameters would require resetting the MMCM tile, so they must remain unchanged and, therefore, not be written using the CDDC process. As an additional rule, the ReconOS64 clocking component always selects the closest divider that does not result in a frequency higher than requested, so the frequency passed to the clock reconfiguration function can be the maximum possible frequency a given hardware thread supports. The selected resulting frequency is returned to the application so it can be used instead of the requested frequency, for example, to calculate times from cycle counts. The divider values for high and low times are then written to the AXI bus address of the clock hardware core, where they trigger the reconfiguration state machine.

As shown with the two blue markers at 140 and 152 cycles from the trigger point, the overall time the clock reconfiguration itself is running with a disabled output clock spans over 12 clock cycles, which is the effective downtime for a hardware thread during the clock reconfiguration. If the data handshake starting with the selection line toggling at 126 is included, the overall CDDC time is 26 cycles before the clock is available again.

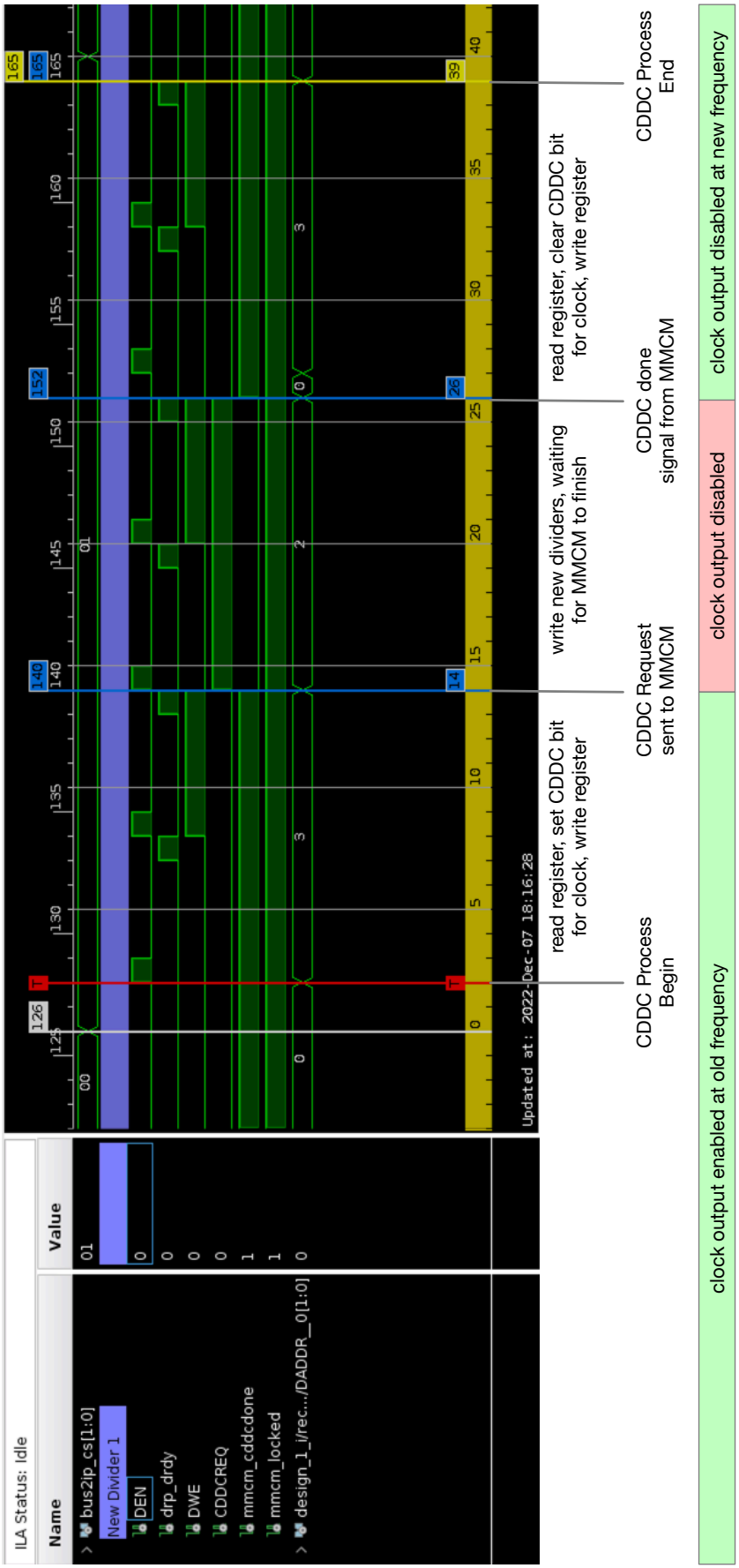

Figure 4.5: Vivado Integrated Logic Analyzer waveform of a MMCM tile CDDC operation.

4.4 RESOURCE MANAGEMENT FOR INDIVIDUAL APPLICATIONS

Considering the described options for configuring the contents of the available hardware application slots, their respective clocking infrastructure, as well as their low-downtime runtime reconfiguration, the ReconOS64 system can be adapted for individual workloads without generating much overhead in terms of downtime.

In an application scenario comprising a set of multiple independent hardware-allocated tasks, the adaption of individual clock speeds helps optimize the overall system performance by minimizing the processing time through each individual task. An exemplary task set that could function as an example for this model is described together with real-time scheduling considerations in the following section.

Figure 4.6: Three-step batch processing mode for hardware accelerators.

A typical mode of operation for a cooperative hardware-software co-design is a batch-based processing approach as presented in Figure 4.6, where individual hardware threads receive a batch of data to process, depending on their respective implemented function. After receiving the data, the processing takes place, followed by a second communication phase for writing processing results back into the shared memory. The extent and ratio between the input and output batch sizes vary depending on the implemented hardware thread; the following section provides multiple examples of varying types. Common to all processing methods is the shared and thus limited resource given by the single main memory access path between all hardware threads. For ReconOS64, this single resource is the MEMIF section, comprising the AXI bus interface, MMU, and the arbiter to connect all individual threads to the single interface. Irrespective of the particular resource sharing algorithm used by the arbiter, a certain limitation of the memory interface occurs since only a single transfer can be carried out at the same time via the AXI memory connection.

In order to reduce the risk of memory accesses being delayed by an active transfer, the transaction times on the shared memory bus should be kept as short as possible, thus allowing for a minimal worst-case execution time of memory transfers in case of the port still being blocked by other memory accesses of same or higher priority. To keep the memory transaction times low, the clocking of the MEMIF subsystem should be run at a high clock speed, limited by the maximum possible frequency for both the AXI interface as well as the MEMIF logic components. The theoretical maximum possible clock frequency for the UltraScale+ PS-PL AXI interface is $F_{AXICLKMAX} = 333\text{MHz}$ [4,

Tab.33]. The frequency of 200MHz was chosen as a baseline for most of the evaluation data presented since it offered a baseline that can also be achieved by Zynq Series-7 devices [70, Tab.19], thus allowing for comparison between both series throughout various evaluations.

A clocking at this rate might not be compatible with the maximum clock speeds of hardware threads, as defined by their individual logic's critical path. The example threads used in section 5.2 reach different maximum possible clock speed estimations, ranging from 122MHz to 390MHz [24, Tab.2]. However, since the mode of operation is separated into individual input, processing, and output steps, which are effectively decoupled by asynchronously readable and writable ReconOS-FIFOs, the memory access can be done at a speed different from the hardware thread's clock speed, thus allowing matching the processing speed closely to the maximum speed of the hardware thread's logic, while keeping the MEMIF speed at a different, and especially compared to extensive thread logic with long critical paths, higher clock speed.

As the maximum processing frequency of multiple hardware threads synthesized from the same HDL code but targeting different reconfigurable hardware slots should always be similar, ReconOS64 allows setting an individual clock per slot group in order to save on the limited number of available clock outputs. If multiple instances generated from the same HDL code at individual locations should be run at different clock speeds, the reconfigurable slots can, however, still be decoupled by duplicating the HDL code and defining individual slot groups assigned to different clock outputs of the MMCM tile.

5

RECONOS PORTABILITY & RECONOS64 USE CASES

Throughout the ReconOS project's evolution, support for multiple platforms was added regarding both software and hardware components. The initial eCos host Real-Time Operating System (RTOS) was replaced by Linux, while the hardware support changed towards heterogeneous platforms using an ARM CPU combined with a Programmable Logic (PL). The most recent ReconOS version supports ARMv7-based processors of Zynq Sytem-on-Chip (SoC) devices as well as *MicroBlaze* soft-core processor within the FPGA component, while ReconOS64 adds support for ARMv8-A Multi-Processor System-on-Chip (MPSoC) devices of the Zynq UltraScale+ series. Depending on the intended application environment and its constraints regarding throughput, energy consumption, parallel processing, or operating system, both ReconOS and ReconOS64 offer flexible options for various scenarios on different platforms. While the first sections of this chapter present ReconOS projects on smaller hardware, the later sections cover ReconOS64-based designs for applications on larger systems.

5.1 PORTABILITY OF RECONOS

The following sections provide an overview of the most recent additions to ReconOS, which were created as additional student projects in parallel to the development period of ReconOS64. They demonstrate the portability options of ReconOS even for smaller SoC and FPGA platforms with low-power processors.

5.1.1 *ARM, MicroBlaze& FreeRTOS on Zynq SoC*

MicroBlaze is a small soft-core microprocessor provided by Xilinx to be embedded in hardware designs using vendor tools. It offers multiple customization options regarding, for example, the memory and data architecture or instruction set extensions. An experimental approach of porting ReconOS concepts towards a MicroBlaze design combined with a FreeRTOS real-time operating system was conducted as a student's thesis project by Mehta [48]. FreeRTOS was selected due to its small hardware requirements regarding both memory and CPU performance. Since FreeRTOS does not implement virtual memory

and, therefore, does not require any address translations, the Memory Interface (MEMIF) section does not require the MMU to be present. Instead, pointer addresses can be directly forwarded to the memory interface to retrieve or store data from within the hardware threads. The Operating-System Interface (OSIF) was adapted to support communication primitives already established in FreeRTOS, such as semaphores or mutexes, as well as implementing data communication via message boxes. Figure 5.1 presents a system overview of the Cortex-A9-based system. In contrast to the default ReconOS architecture, the MEMIF MMU was removed.

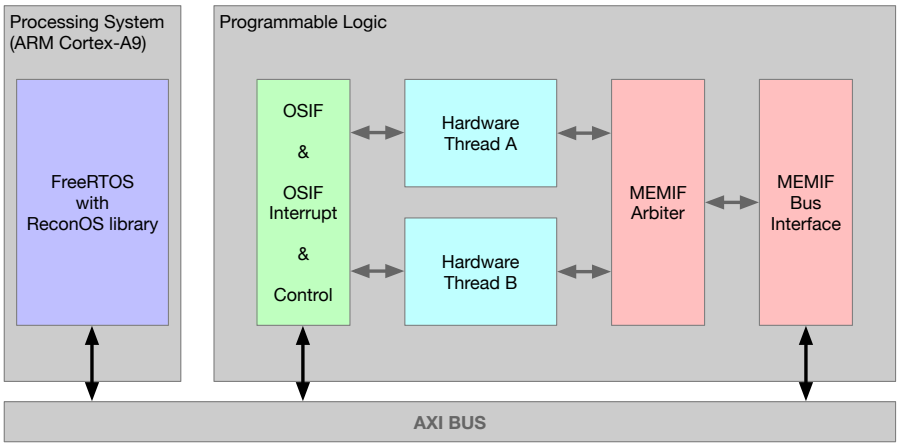

Figure 5.1: System diagram for ReconOS on a 32-bit ARM Cortex-A9 processor running FreeRTOS without virtual memory. Based on [48].

5.1.2 *RISC-V & ZephyrOS on Zynq SoC*

A student thesis project by Rao [53], which was conducted under third-party supervision, ported ReconOS to a RISC-V-based NEORV32 [49] soft-core processor. It uses ZephyrOS [72] as a real-time operating system on a ZCU104 Zynq UltraScale+ MPSoC board. The implementation did not include support for virtual memory, so the MMU was omitted, thus enabling memory access via the physical addresses from within the system. The ReconOS OS calls for mutexes and semaphores are forwarded to the respective ZephyrOS functions, so ReconOS threads can use these calls irrespective of the underlying operating system.

5.1.3 *Real-Time Linux*

A student thesis project by Lienen [36] focused on a real-time scheduling scenario on a Linux-based ReconOS system. A heterogeneous set of tasks, including three robotics sensor and movement controllers together with a video processing pipeline, were implemented to a

7-series Zynq SoC. Multiple scheduling options, including the possibility for partial reconfiguration of individual slots, were analyzed. Bitstream changes were unsuitable for this application due to the high reprogramming overhead conflicting with the thread's cyclic real-time requirements. A priority-based scheduler was introduced together with applying the `PREEMPT_RT` real-time patch [55] to the Linux kernel.

5.1.4 *MicroBlaze & FreeRTOS on FPGA*

During a student thesis project by Tcheussi Ngayap [60], a ReconOS system using FreeRTOS as its host operating system was implemented on a MicroBlaze processor within an Artix-7 FPGA device on a Nexys 4 DDR evaluation board. The system implements a neural network using ZyNet [63] for classifying handwritten digits of the MNIST dataset [35]. Figure 5.2 presents the system diagram of the implemented design. Since only a single hardware thread and no virtual memory were used, the arbiter and MMU were removed. The implemented neural network processed the 28x28 pixel input images as a 784 elements vector, feeding the data to two hidden layers before reaching an output layer consisting of ten output classes. The hardware thread finds the maximum output probability value and returns the respective class for one of the ten possible recognizable digits. A comparison between a direct hardware implementation and a ReconOS hardware thread was conducted, comparing processing latency and hardware utilization. It was demonstrated that a neural network could be implemented as a ReconOS thread along with a soft-core processor inside a small FPGA system that does not contain a dedicated processing system.

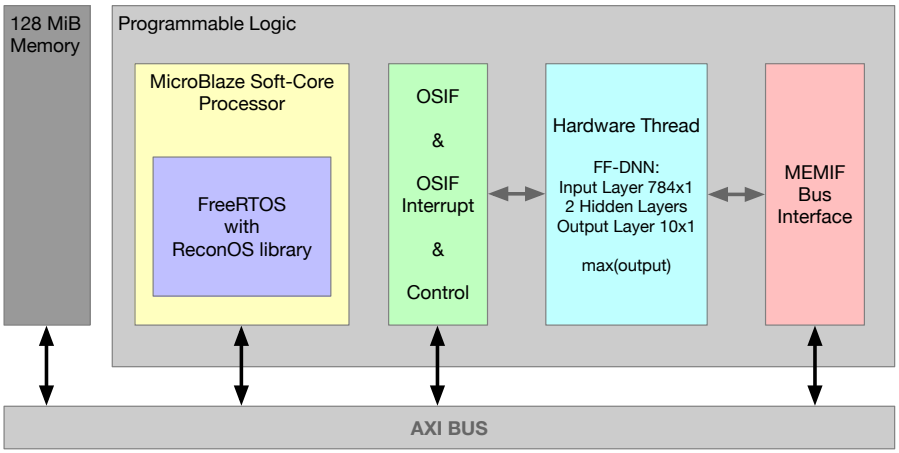

Figure 5.2: System diagram for ReconOS on a MicroBlaze soft-core processor inside an Artix-7 FPGA. Based on [60].

5.2 REAL-TIME APPLICATIONS

The integration of real-time applications to ReconOS64 can be achieved in multiple ways based on different operating systems. For application environments requiring a real-time operating system like *FreeRTOS*, ReconOS was integrated into these systems. Another option is to utilize a default Linux operating system, with the possibility of an added real-time kernel patch. Several implementations of demonstrator systems were created and evaluated for both ReconOS and ReconOS64 as presented in section 5.1.

In [24], a ReconOS64 environment implementing a set of five different periodic real-time tasks was presented to evaluate the schedulability of the overall system. The execution of each task is divided into three consecutive phases: *memory in*, *execution*, and *memory out*, resembling a task model and its schedulability analysis presented in [42]. Each task τ_i is assigned a priority P_i. A scheduler controls the task set, issuing clearance for input and output memory accesses via the shared memory interface. While the input operation of all tasks is organized according to their respective task priority, the output operation is based on a first in, first out (FIFO) queue.

The demonstration tasks included two image processing tasks (sobel filtering and grayscale conversion), a SHA1 hashing task, a robotics kinematic task, and a bubble sorting task. This set of tasks τ_1 to τ_5 was selected to reflect different workload types. The image processing tasks feature a symmetrical input and output data size of 900kiB, while the SHA1 hash demonstrates a task with a 1MiB input data size that is hashed into 32B of output data. The kinematic task represents tasks with a smaller but symmetrical input and output size of 8B each, and the sorting task represents a task with a medium-sized symmetrical data size of 16kiB for input and output.

The memory phases occupy the shared ReconOS64 MEMIF, while the execution phase happens in isolation within the hardware thread as the data is buffered inside of the Block-RAM (BRAM) or UltraRAM (URAM) local to each hardware thread. This matches the processing model as described in section 4.4. In order to apply the proposed schedulability test, it is necessary to determine the individual duration of each phase per task.

For the demonstration task set, the individual execution times for all tasks were recorded by attaching AXI Timers [67], providing two capture ports each to debug ports of each hardware thread. Figure 5.3 presents an exemplary hardware task τ_1 with asymmetrical input and output data sizes in a ReconOS64 system. First, the address pointers for the allocated input and output memory regions are transferred via OSIF messages (①), followed by the hardware thread loading the input data via the MEMIF (②) and starting the execution step. This generates

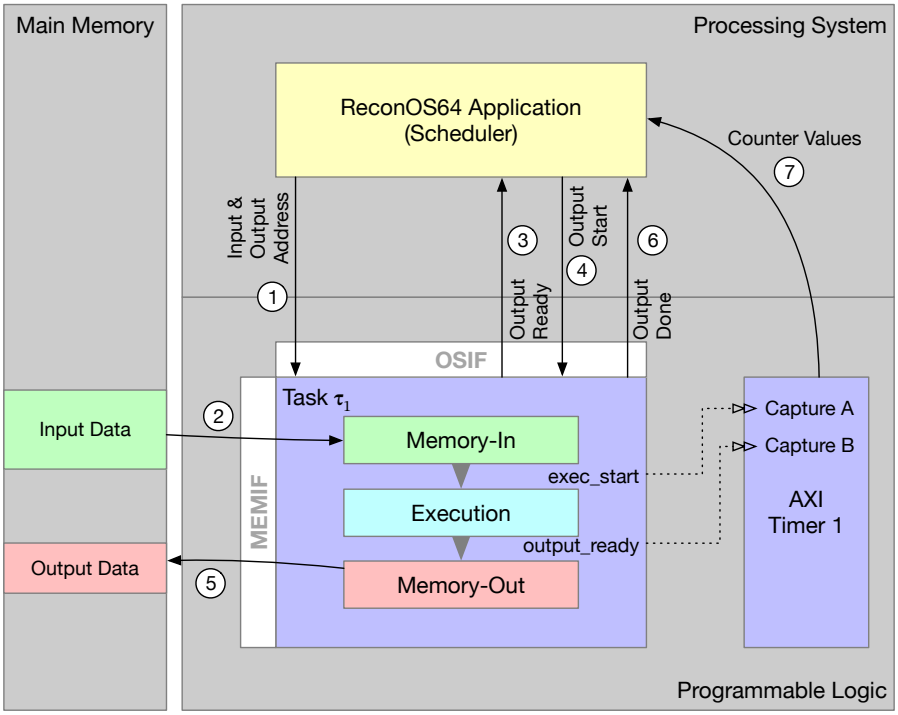

Figure 5.3: ReconOS64 execution time measurement for three-phased tasks [24].

a capture signal for the AXI timer, so it records the current timer value into the first capture register. Once the processing is finished, the AXI timer's second input is triggered to capture into the second register. The scheduler in the software application is notified via a semaphore through the OSIF (③) that new output data is available. After the scheduler initiates the return of the output data (④), it is written to the main memory (⑤), and the output step is signaled complete afterward (⑥). Finally, the AXI counter is read to retrieve the tick values for the beginning and end of the execution phase to calculate its duration in clock cycles.

To provide consistent timing measurements for the memory access, the Translation Lookaside Buffer (TLB) of the Memory Management Unit (MMU) was pre-populated with the translation of each memory address used by executing hardware-issued accesses to all pages of the input and output address range before starting the measurement. This ensures no virtual address translation needs to be carried out during the measurement window. This assumption is justified since a large enough TLB always ensures virtual memory access without disturbances. In addition, RTOSs such as FreeRTOS do not offer virtual memory, so the MEMIF path on such platforms lacks the MMU, resulting in a memory access timing where no address translations need to be considered.

The overall priority-based schedulability of a given task set was shown to be given if the task deadlines allowed sufficient time for the

memory-in operation of the respective task and each higher-prioritized task, its execution time, and the cumulative memory-out phase of all tasks.

5.2.1 *Considerations for Hardware Scheduling*

The previous demonstration environment included a set of tasks that could fit the overall logic inventory of the target system. Therefore, a single static PL design and bitstream was sufficient to map all tasks to the system. If, however, a set of tasks exceeds the logic inventory capacity of a system, partial bitstream reprogramming of individual hardware tasks at runtime becomes relevant.

While partial reconfiguration, as offered by ReconOS64, would suit the overall system concept, it would impose further restrictions. The previous example considered memory access as the single shared resource impacting the real-time schedulability of the overall task set. If partial reconfiguration for loading and unloading individual tasks were to be added to the system, two additional impact factors would arise. First, the deadline considerations must include the reconfiguration time itself, i.e., loading the bitstream via either the Processor Configuration Access Port (PCAP) or Internal Configuration Access Port (ICAP) port. Here, enhancing the reconfiguration speed using custom ICAP implementations as presented in 3.6.2 would result in reduced blocking times. As a second factor, the configuration access port itself has to be considered a shared medium, thus taking into account the worst-case blocking times if any task of higher priority already enqueued a partial reconfiguration, thus delaying the reprogramming of a task beyond the reconfiguration time itself. These blocking times could also be reduced by utilizing fast configuration ports.

5.3 RECONROS

ReconROS [37], first presented in [38], is a ReconOS- and ReconOS64-based fork for robotic applications, integrating functionalities of the ROS 2 Robot Operating System [57], targeting both 7-series Zynq and 8-series Zynq UltraScale+ devices.

It allows individual ROS nodes to be mapped to either hardware or software, similar to the placement of threads in ReconOS64. For communication, it allows bridging the ROS subscription and topic model to hardware-located nodes. In addition, ReconROS integrates the ZyCAP [64] driver for ICAP-based configuration access on 7-series devices.

Figure 5.4 presents the process of a hardware-mapped node subscribing to a topic. If the node requests a `subscription_take` (①), the call is forwarded via the OSIF delegation process of ReconOS to the

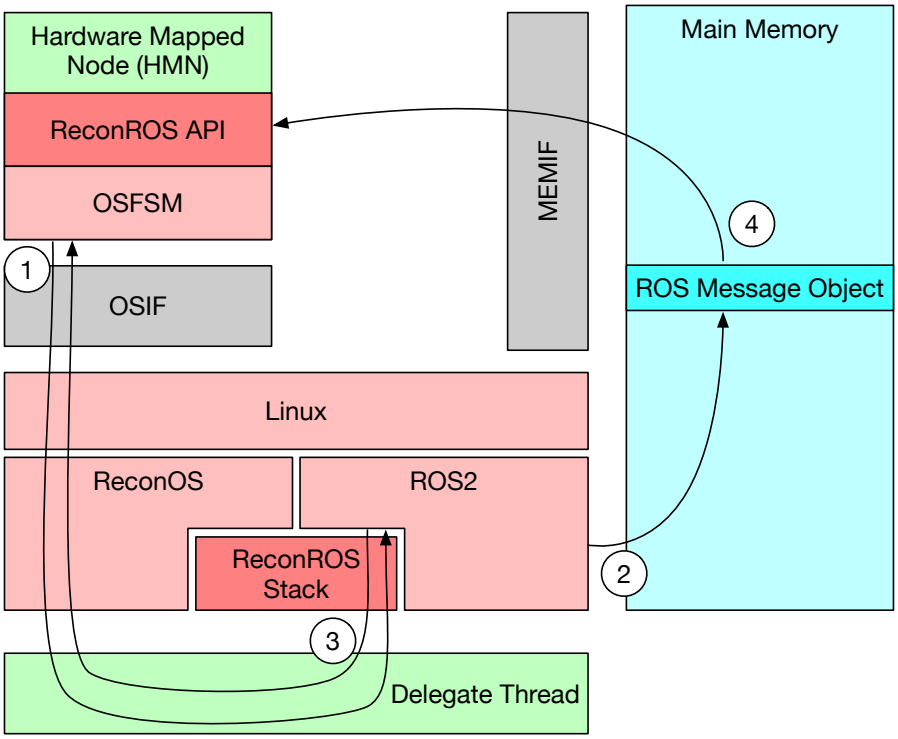

Figure 5.4: ReconROS subscription process for a hardware-mapped topic. Adapted from [37].

ROS2 system. The requested message object is placed into the main memory (②), returning its address, which will be forwarded to the hardware node (③). After receiving the pointer of the message object, the hardware-mapped node then uses the MEMIF system to access the message object by issuing a read request to the virtual address location (④).

By including communication and system paradigms from ROS2, ReconROS enables the use of heterogeneous platforms for robotic applications. Similar to ReconOS and ReconOS64, it aims to provide techniques familiar from software development to a unified platform for both hardware and software targets, reducing the gap between development processes specific to each system.

6

CONCLUSION AND FURTHER WORK

The development of ReconOS64 brings the ReconOS concept of inter-thread communication methods across the hardware-software boundary onto modern Multi-Processor System-on-Chip (MPSoC) devices and extends its functionality both from a designer's as well as a system architecture perspective. The presented concept of reconfigurable slots provides a method of grouping similar threads for partial reconfiguration while at the same time providing flexibility for different types, sizes, and amounts of Hardware Threads (HWTs). The architectural changes support the platform switch by embedding flexible virtual memory options for multiple types of addressing and translation modes used throughout various host operating system configurations. The addition of individual clocking options with the option for runtime reconfiguration per slot group allows for a system design adapting its thread frequencies based on the individual needs of various hardware thread implementations. This allows for a flexible system while bringing individual slot operating frequencies close to their respective maximum operating speed.

The presented work provides the open-source code foundations for targeting those platforms, focusing on the resource management of multi-threaded hardware-software co-designs. With increasing numbers of available FPGA systems as well as the growing popularity of FPGAs in both embedded as well as datacenter applications, systems such as ReconOS64 can help manage FPGA logic resources as a shared resource by allowing runtime reconfiguration of individual hardware slots and offering adaptability with respect to clocking settings to match varying types of hardware threads.

6.1 POTENTIAL FOR FUTURE WORK

The presented state of ReconOS64 comprises potential for follow-up research and projects, of which some ideas are outlined here:

- The potential for partial reconfiguration as presented in 3.6 can be exploited by implementing a runtime management chain for bitstream reconfiguration into ReconOS64. Fast interfaces such

as the Internal Configuration Access Port (ICAP) combined with custom management logic as shown in 3.6.2 can be integrated. This requires the definition of additional static regions surrounding the ICAP tile itself but would allow for fast reconfiguration of individual hardware threads.

- An overall integration of fast, runtime-deterministic partial reconfiguration to scheduling considerations could allow for extended schedulability analysis of a given set of exchangeable hardware tasks, thus finding a method to prove the real-time capabilities of a reconfigurable heterogeneous system.

- Integrating partial bitstream relocation for systems comprising multiple uniform slots in a slot group would reduce the number of individual partial bitstreams to be created as described in Chapter 4.

- The Memory Interface (MEMIF) access path can be extended by utilizing the maximum available bit width of 128-bit transfers. This would enhance the throughput but has not been introduced to ReconOS64 since it would violate the default data size of 64-bit, thus making it impossible to access individual 64-bit elements without additional methods for adding padding to accesses and ensuring alignment to 16 bytes.

- The ReconOS64 memory subsystem utilizes the Accelerator Coherency Port (ACP) for providing Advanced eXtensible Interface (AXI) bus access. Enabling a secondary method of transfers via the High-Performance (HP) AXI ports, potentially including a parallel utilization of multiple HP0...HP3 ports would allow for higher transfer speeds [44] but would require additional caution to ensure coherency between accesses originating from Processing System (PS) and Programmable Logic (PL).

- The clocking infrastructure currently utilizes a single Mixed-Mode Clock Management (MMCM) tile, providing up to six configurable outputs for individual slot groups. The ZU7EV MPSoC found, for example, on the ZCU104 development board, provides up to eight MMCM tiles. Utilizing multiple tiles with algorithms to distribute the requested clock frequency ranges of individual thread groups could lead to a more fine-grained frequency resolution using optimized Voltage-Controlled Oscillator (VCO) frequencies. Currently, the ReconOS64 defaults to the maximum VCO frequency in order to enable many different clock steps at the cost of reduced granularity for selectable frequencies.

- In order to achieve high utilization for MPSoC systems with high FPGA logic cell count, ReconOS64 could be extended to offer multi-tenant application scenarios. Especially due to the

newer ARMv8 architecture extensions providing hypervisor capabilities, a system housing multiple guest operating systems sharing a single FPGA could be achieved. An alternative approach without relying on multiple virtual machines would be to extend ReconOS64 to support multiple applications at the same time. Both approaches require the MMU to switch between different virtual memory translation tables. The OSIF implementation would differ for a VM-based approach, while for a multi-application architecture, it would remain connected to a single operating system interface.

AUTHOR'S PUBLICATIONS

[1] Alexander Boschmann, Lennart Clausing, Felix Jentzsch, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "Flexible Industrial Analytics on Reconfigurable Systems-On-Chip." In: *On-The-Fly Computing – Individualized IT-services in dynamic markets*. Ed. by Claus-Jochen Haake, Friedhelm Meyer auf der Heide, Marco Platzner, Henning Wachsmuth, and Heike Wehrheim. Vol. 412. Verlagsschriftenreihe des Heinz Nixdorf Instituts. Heinz Nixdorf Institut, Universität Paderborn, 2023, pp. 225–236. DOI: 10.5281/zenodo.8068713.

[2] Lennart Clausing. "ReconOS64: High-Performance Embedded Computing for Industrial Analytics on a Reconfigurable System-on-Chip." In: *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. HEART '21. Online, Germany: Association for Computing Machinery, 2021. ISBN: 9781450385497. DOI: 10.1145/3468044.3468056. URL: https://doi.org/10.1145/3468044.3468056.

[3] Lennart Clausing, Zakarya Guettatfi, Paul Kaufmann, Christian Lienen, and Marco Platzner. "On Guaranteeing Schedulability of Periodic Real-Time Hardware Tasks Under ReconOS64." In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 19th International Symposium, ARC 2023, Cottbus, Germany, September 27-29, 2023, Proceedings*. Cottbus, Germany: Springer-Verlag, 2023, pp. 245–259. ISBN: 978-3-031-42920-0. DOI: 10.1007/978-3-031-42921-7_17. URL: https://doi.org/10.1007/978-3-031-42921-7_17.

[4] Lennart Clausing and Marco Platzner. "ReconOS64: A Hardware Operating System for Modern Platform FPGAs with 64-Bit Support." In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 120–127. DOI: 10.1109/IPDPSW55747.2022.00029.

BIBLIOGRAPHY

[1] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. "ReconOS – An Operating System Approach for Reconfigurable Computing." In: *IEEE Micro* 34.1 (Jan. 2014), pp. 60–71. ISSN: 0272-1732. DOI: 10.1109/MM.2013.110. URL: http://ieeexplore.ieee.org/document/6636314/.

[2] AMD Xilinx. *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)*. v1.10. 2022. URL: https://www.amd.com/content/dam/xilinx/support/documents/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

[3] AMD Xilinx. *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*. v2.4. Dec. 2023. URL: https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm.

[4] AMD Xilinx. *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics (DS925)*. v1.23. 2023. URL: https://www.amd.com/content/dam/xilinx/support/documents/data_sheets/ds925-zynq-ultrascale-plus.pdf.

[5] *AMD Zynq UltraScale+ MPSoC Product Website*. https://www.amd.com/de/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html. Accessed 2025-01-28.

[6] AMD, Inc. *Zynq UltraScale+ MPSoC Software Developer Guide (UG1137)*. v2024.1. 2024. URL: https://docs.xilinx.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev.

[7] AMD, Inc. *Xilinx FPGA Resource Manager*. https://xilinx.github.io/XRM/. Accessed 2025-01-03.

[8] AMD, Inc. *Xilinx XRT: ZOCL Driver Interfaces*. https://xilinx.github.io/XRT/master/html/zocl_ioctl.main.html. Accessed 2025-01-03.

[9] AMD, Inc. *Xilinx®Runtime (XRT) Architecture*. https://xilinx.github.io/XRT/2024.2/html/index.html. Accessed 2024-12-29.

[10] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. "Achieving Programming Model Abstractions for Reconfigurable Computing." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.1 (2008), pp. 34–44. DOI: 10.1109/TVLSI.2007.912106.

[11] *Architecture of ReconOS*. http://www.reconos.de/documentation/architecture/. Accessed 2025-01-03.

[12] *Architecture of ReconOS*. http://www.reconos.de/documentation/api/. Accessed 2025-01-03.

[13] *ARM Developer: An introduction to AMBA AXI*. https://developer.arm.com/documentation/102202/0300. Accessed 2024-12-11.

[14] ARM Ltd. *ARM Cortex-A Series Programmer's Guide (ARM DEN0013D)*. ID012214. ARM Ltd. 2014.

[15] ARM Ltd. *ARM Cortex-A Series Programmer's Guide for ARMv8-A (ARM DEN0024A)*. ID050815. ARM Ltd. 2015.

[16] ARM Ltd. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition (ARM DDI 0406C.d)*. ID040418. ARM Ltd. 2018.

[17] ARM Ltd. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile (ARM DDI 0487D.b)*. ID042519. ARM Ltd. 2019.

[18] ARM Ltd. *Armv8-A Address Translation (100940)*. Version 1.1. 2019.

[19] ARM Ltd. *Understanding the Armv8.x extensions (ARM062-948681440-2884)*. Version 1.0. 2019.

[20] ARM Ltd. *Learn the architecture - AArch64 Exception Levels (102412)*. Version 1.3. ARM Ltd. 2022. URL: https://developer.arm.com/documentation/102412/0103.

[21] Gordon Brebner. "A virtual hardware operating system for the Xilinx XC6200." In: *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*. Ed. by Reiner W. Hartenstein and Manfred Glesner. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 327–336. ISBN: 978-3-540-70670-0.

[22] Alex R. Bucknall and Suhaib A. Fahmy. "ZyPR: End-to-end Build Tool and Runtime Manager for Partial Reconfiguration of FPGA SoCs at the Edge." In: 16.3 (June 2023). ISSN: 1936-7406. DOI: 10.1145/3585521. URL: https://doi.org/10.1145/3585521.

[23] Lennart Clausing. "ReconOS64: High-Performance Embedded Computing for Industrial Analytics on a Reconfigurable System-on-Chip." In: *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. HEART '21. Online, Germany: Association for Computing Machinery, 2021. ISBN: 9781450385497. DOI: 10.1145/3468044.3468056. URL: https://doi.org/10.1145/3468044.3468056.

[24] Lennart Clausing, Zakarya Guettatfi, Paul Kaufmann, Christian Lienen, and Marco Platzner. "On Guaranteeing Schedulability of Periodic Real-Time Hardware Tasks Under ReconOS64." In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 19th International Symposium, ARC 2023, Cottbus, Germany,*

September 27–29, 2023, Proceedings. Cottbus, Germany: Springer-Verlag, 2023, pp. 245–259. ISBN: 978-3-031-42920-0. DOI: 10.1007/978-3-031-42921-7_17. URL: https://doi.org/10.1007/978-3-031-42921-7_17.

[25] Lennart Clausing and Marco Platzner. "ReconOS64: A Hardware Operating System for Modern Platform FPGAs with 64-Bit Support." In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 120–127. DOI: 10.1109/IPDPSW55747.2022.00029.

[26] Marcel Eckert, Dominik Meyer, Jan Haase, and Bernd Klauer. "Operating System Concepts for Reconfigurable Computing: Review and Survey." In: *International Journal of Reconfigurable Computing* 2016.1 (2016), p. 2478907. DOI: https://doi.org/10.1155/2016/2478907. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2016/2478907. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2016/2478907.

[27] *eCos website*. http://ecos.sourceware.org. Accessed 2024-09-25.

[28] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. "The LEAP FPGA operating system." In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927488.

[29] *FPGA Manager documentation*. https://www.kernel.org/doc/html/v4.19/driver-api/fpga/fpga-mgr.html. Accessed 2024-10-16.

[30] *Install Ubuntu on AMD website*. https://ubuntu.com/download/amd. Accessed 2024-10-16.

[31] Aws Ismail and Lesley Shannon. "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators." In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. 2011, pp. 170–177. DOI: 10.1109/FCCM.2011.48.

[32] Xabier Iturbe, Khaled Benkrid, Ahmet T. Erdogan, Tughrul Arslan, Mikel Azkarate, Imanol Martinez, and Antonio Perez. "R3TOS: A reliable reconfigurable real-time operating system." In: *2010 NASA/ESA Conference on Adaptive Hardware and Systems*. 2010, pp. 99–104. DOI: 10.1109/AHS.2010.5546274.

[33] Xabier Iturbe, Khaled Benkrid, Chuan Hong, Ali Ebrahim, Raul Torrego, Imanol Martinez, Tughrul Arslan, and Jon Perez. "R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs." In: *IEEE Transactions on Computers* 62.8 (2013), pp. 1542–1556. DOI: 10.1109/TC.2013.79.

[34] Alexander Klassen. *Fast Partial Reconfiguration for ReconOS64 on Xilinx MPSoC Devices*. Paderborn University, 2023.

[35] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[36] Christian Lienen. "Implementing a Real-time System on a Platform FPGA operated with ReconOS." MA thesis. 2019.

[37] Christian Lienen, Marco Platzner, Erdal Kayacan, Roman Dumitrescu, Stefan Sauer, and Tobias Kenter. *Enabling reconfigurable hardware acceleration for ROS-based robotics applications*. Paderborn, 2024. DOI: 10.17619/UNIPB/1-1960. URL: https://dx.doi.org/10.17619/UNIPB/1-1960.

[38] Christian Lienen, Marco Platzner, and Bernhard Rinner. "ReconROS: Flexible Hardware Acceleration for ROS2 Applications." In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 268–276. DOI: 10.1109/ICFPT51103.2020.00046.

[39] *Linux Kernel Devicetree Overlay documentation*. https://docs.kernel.org/devicetree/overlay-notes.html. Accessed 2025-01-31.

[40] *Linux Kernel Page Tables documentation*. https://docs.kernel.org/mm/page_tables.html. Accessed 2025-01-31.

[41] Enno Lubbers and Marco Platzner. "ReconOS: An RTOS Supporting Hard-and Software Threads." In: *2007 International Conference on Field Programmable Logic and Applications*. IEEE, 2007, pp. 441–446. DOI: 10.1109/FPL.2007.4380686.

[42] Cláudio Maia, Geoffrey Nelissen, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. "Schedulability analysis for global fixed-priority scheduling of the 3-phase task model." In: *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2017, pp. 1–10. DOI: 10.1109/RTCSA.2017.8046313.

[43] Kristiyan Manev, Joseph Powell, Kaspar Matas, and Dirk Koch. "byteman: A Bitstream Manipulation Framework." In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022, pp. 1–9. DOI: 10.1109/ICFPT56656.2022.9974549.

[44] Kristiyan Manev, Anuj Vaishnav, and Dirk Koch. "Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems." In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 179–187. DOI: 10.1109/ICFPT47387.2019.00029.

[45] Catalin Marinas. *Linux on AArch64 ARM 64-bit Architecture*. LinuxCon North America, 2012. URL: https://events.static.linuxfound.org/images/stories/pdf/lcna_co2012_marinas.pdf.

[46] Catalin Marinas. *Memory Layout on AArch64 Linux*. https://www.kernel.org/doc/Documentation/arm64/memory.txt. Accessed 2024-12-19. Linux Kernel Organization, Inc.

[47] M.C. McFarland, A.C. Parker, and R. Camposano. "The high-level synthesis of digital systems." In: *Proceedings of the IEEE* 78.2 (1990), pp. 301–318. DOI: 10.1109/5.52214.

[48] Jinay D Mehta. "Multithreaded Software/Hardware Programming with ReconOS/freeRTOS on a Reconfigurable System-on-Chip." MA thesis. 2019.

[49] *NEORV32 website*. https://stnolting.github.io/neorv32/. Accessed 2025-01-29.

[50] *PBLOCK property, Vivado documentation*. https://docs.amd.com/r/en-US/ug912-vivado-properties/PBLOCK. Accessed 2024-10-24.

[51] Khoa Dang Pham, Kyriakos Paraskevas, Anuj Vaishnav, Andrew Attwood, Malte Vesper, and Dirk Koch. "ZUCL 2.0: Virtualised Memory and Communication for ZYNQ UltraScale+ FPGAs." In: *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*. 2019, pp. 1–9.

[52] Khoa Dang Pham, Anuj Vaishnav, Malte Vesper, and Dirk Koch. "ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications." In: *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. 2018, pp. 1–9.

[53] Aniruddh P Rao. "Multithreaded Software/Hardware Programming with ReconOS/Zephyr on a RISC-V-based System-on-Chip." MA thesis. 2023.

[54] *ReconOS Project History Website*. http://www.reconos.de/about/history/. Accessed 2025-01-05.

[55] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. "The Real-Time Linux Kernel: A Survey on PREEMPT_RT." In: *ACM Comput. Surv.* 52.1 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3297714. URL: https://doi.org/10.1145/3297714.

[56] Jens Rettkowski, Konstantin Friesen, and Diana Göhringer. "RePaBit: Automated generation of relocatable partial bitstreams for Xilinx Zynq FPGAs." In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016, pp. 1–8. DOI: 10.1109/ReConFig.2016.7857186.

[57] *ROS - Robot Operating System Project Website*. https://ros.org/. Accessed 2025-01-05.

[58] Chris Shore. *Porting to 64-bit Arm - White Paper*. v1.0. ARM Ltd. July 2014. URL: https://developer.arm.com/documentation/102902/0100.

[59] Hayden Kwok-Hay So, Artem Tkachenko, and Robert Brodersen. "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH." In: *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '06. Seoul, Korea: Association for Computing Machinery, 2006, pp. 259–264. ISBN: 1595933700. DOI: 10.1145/1176254.1176316. URL: https://doi.org/10.1145/1176254.1176316.

[60] Vanessa Ingrid Tcheussi Ngayap. "FreeRTOS on a MicroBlaze Soft-Core Processor with Hardware Accelerators." MA thesis. 2022.

[61] Anuj Vaishnav, Khoa Dang Pham, Kristiyan Manev, and Dirk Koch. "The FOS (FPGA Operating System) Demo." In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 429–429. DOI: 10.1109/FPL.2019.00081.

[62] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. "FOS: A Modular FPGA Operating System for Dynamic Workloads." In: *ACM Trans. Reconfigurable Technol. Syst.* 13.4 (Sept. 2020). ISSN: 1936-7406. DOI: 10.1145/3405794. URL: https://doi.org/10.1145/3405794.

[63] Kizheppatt Vipin. "ZyNet: Automating Deep Neural Network Implementation on Low-Cost Reconfigurable Edge Computing Platforms." In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 323–326. DOI: 10.1109/ICFPT47387.2019.00058.

[64] Kizheppatt Vipin and Suhaib A. Fahmy. "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq." In: *IEEE Embedded Systems Letters* 6.3 (2014), pp. 41–44. DOI: 10.1109/LES.2014.2314390.

[65] Herbert Walder and Marco Platzner. "A Runtime Environment for Reconfigurable Hardware Operating Systems." In: *Field Programmable Logic and Application*. Ed. by Jürgen Becker, Marco Platzner, and Serge Vernalde. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 831–835. ISBN: 978-3-540-30117-2. DOI: doi.org/10.1007/978-3-540-30117-2_84.

[66] Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12 (2013), pp. 2179–2192. DOI: 10.1109/TVLSI.2012.2231101.

[67] Xilinx, Inc. *AXI Timer v2.0 LogiCORE IP Product Guide (PG079)*. v2.0. Oct. 2016. URL: https://docs.amd.com/v/u/en-US/pg079-axi-timer.

[68] Xilinx, Inc. *AXI HBICAP v1.0 LogiCORE IP Product Guide (PG349)*. v1.0. Oct. 2019.

[69] Xilinx, Inc. *PetaLinux Tools Documentation - Reference Guide (UG1144)*. v2020.1. 2020. URL: https://docs.amd.com/v/u/2020.1-English/ug1144-petalinux-tools-reference-guide.

[70] Xilinx, Inc. *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, Z-7020): DC and AC Switching Characteristics (DS187)*. v1.21. 2020. URL: https://docs.amd.com/v/u/en-US/ds187-XC7Z010-XC7Z020-Data-Sheet.

[71] Xilinx, Inc. *UltraScale Architecture Clocking Resources (UG572)*. v1.10.1. 2021.

[72] *ZephyrOS website*. https://www.zephyrproject.org/. Accessed 2025-01-29.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX: https://bitbucket.org/amiede/classicthesis/