Paderborn University — Faculty EIM-I
Computer Engineering Group

Implementing a Real-time System on a Platform FPGA operated with ReconOS

Master's Thesis
Computer Engineering Group
Prof. Dr. M. Platzner

Department of Computer Science
Faculty of Electrical Engineering,
Computer Science and Mathematics
Paderborn University

submitted by
Christian Lienen

September 2019

Supervisors:	Prof. Dr. M. Platzner
	Lennart Clausing, M.Sc.
Examiners:	Prof. Dr. M. Platzner
	Prof. Dr. S. Hellebrand

Christian Lienen
Matriculation number: 7062779
Biekehöhe 5
33129 Delbrück

Abstract

This thesis presents the development of an extended demonstrator based on an existing Ball-on-Plate Stewart-platform. The demonstrator is a real-world example of a Stewart platform, which allows the movement of a surface in six degrees of freedom. Due to the presented extensions on the setup, three of these platforms are available. The resulting demonstrator also enables the processing and output of an HDMI input signal as an HDMI output signal. The goal of these enhancements is to create a heterogeneous set of different tasks for real-time studies. A Xilinx Zynq-7000 Platform FPGA is used for the calculation of the control and video processing, which offers programmable logic in addition to two ARM Cortex-A9 processor cores. For the development of the FPGA design and the software ReconOS is used, whose real-time behavior is improved by the extension of priority-based scheduling. For the design of the real-time system, execution times of the set of tasks are determined by measurements. Other measurements are performed on the demonstrator that show the temporal behavior of different communication types and memory accesses even when used in parallel. Different degrees of parallel processing are implemented and examined regarding the run time.

Zusammenfassung

In der vorliegenden Masterarbeit wird die Entwicklung eines Demonstrators vorgestellt, der auf der Basis eines vorhandenen Ball-on-Plate Demonstrators entwickelt wird. Dieser Demonstrator basiert auf einer Stewart-Plattform, die die Bewegung einer Oberfläche in sechs Freiheitsgraden ermöglicht. Durch die vorgestellte Erweiterung stehen insgesamt drei dieser Plattformen zur Verfügung. Der erweiterte Demonstrator ermöglicht zudem die Bearbeitung und Ausgabe eines HDMI-Eingangssignals als HDMI-Ausgangssignal. Das Ziel dieser Erweiterungen ist die Schaffung einer heterogenen Menge an verschiedenen Tasks für Echtzeituntersuchungen. Für die Berechnung der Regelung und der Videobearbeitung wird ein Xilinx Zynq-7000 Platform FPGA eingesetzt, das neben zweier ARM Cortex-A9 Prozessorkernen eine programmierbare Logik bietet. Für die Entwicklung des FPGA Designs und der Software wird ReconOS eingesetzt, dessen Echtzeitverhalten durch die Erweiterung um prioritätenbasiertes Scheduling verbessert wird. Für die Auslegung der Regelung werden Bearbeitungszeiten durch Messungen ermittelt. An dem Demonstrator werden Versuche durchgeführt, die das zeitliche Verhalten von verschiedenen Kommunikationstypen und Speicherzugriffen auch bei paralleler Nutzung darstellen. Verschiedene Grade von paralleler Abarbeitung werden implementiert und auf die Laufzeit hin untersucht.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The general technical progress touches an ever-increasing part of our life. This development requires energy-efficient embedded devices with a higher need of performance and longer battery lifetimes due to increasing data volumes. Additionally, progresses in machine learning accelerate this trend. For embedded systems that are often powered by batteries or accumulators, the efficient use of available energy is particularly important. To meet these properties, pure software based solutions executed on microcontrollers are often not sufficient. Therefore, combinations of software and hardware execution are more and more used, where subtasks of the problem are executed faster and more efficiently in hardware.

In order to implement parts of the functionality in hardware, various approaches are possible. Besides to the fix implementation in silicon as an Application Specific Integrated Circuit (ASIC), functions can also be implemented using a Field Programmable Gate Array (FPGA). The development of an ASIC is associated with high fixed costs and does not offer the ability to change the design after the production.

In contrast to ASICs, FPGAs offer the advantage that the function can still be changed after production and delivery to the costumer. FPGAs are a class of integrated circuits which mainly consists of three basic components. The first group are standard logic blocks based on look-up tables for combinatorial logic, flip-flops for registers and multiplexer for data selection and the realization of logic functions. These standard blocks are connected by a programmable interconnect. The third group of basic elements are input-output blocks for the connection to the environment in the electrical circuit.

In addition to the described components, modern FPGAs have further units such as transceivers for serial interfaces, block memory (BRAM) and dedicated units for arithmetic operations, e.g. for digital signal processing (DSP-units). As a result, FPGA have also been used for more advanced applications like as hardware accelerators for some years now [18]. Initially, FPGAs were only used as a glue-logic. For the decomposition of functionalities in software and hardware parts there are different approaches. One of these approaches is the usage of ReconOS, which is described in the chapter 2.2.

While the functionality of the FPGA design is typically done in hardware description languages like VHDL or Verilog, High-level Synthesis (HLS) approaches are increasingly used to enable shorter development times and reusability of existing algorithm which are developed for the usage in software.

Since FPGAs are more and more used as hardware accelerators, they are offered as system on chip hybrids that contain fixed CPU cores in addition to reconfigurable logic. With the Cyclone V SoC FPGAs architecture, for example, Intel offers FPGAs with an integrated ARM Cortex-A9 multi-core processor. The abbreviation SoC stands for System-on-Chip, which is typically an integration of processors, memory and other peripherals in one

integrated circuit. A similar architecture is also offered by Xilinx with the Zynq-7000 series.

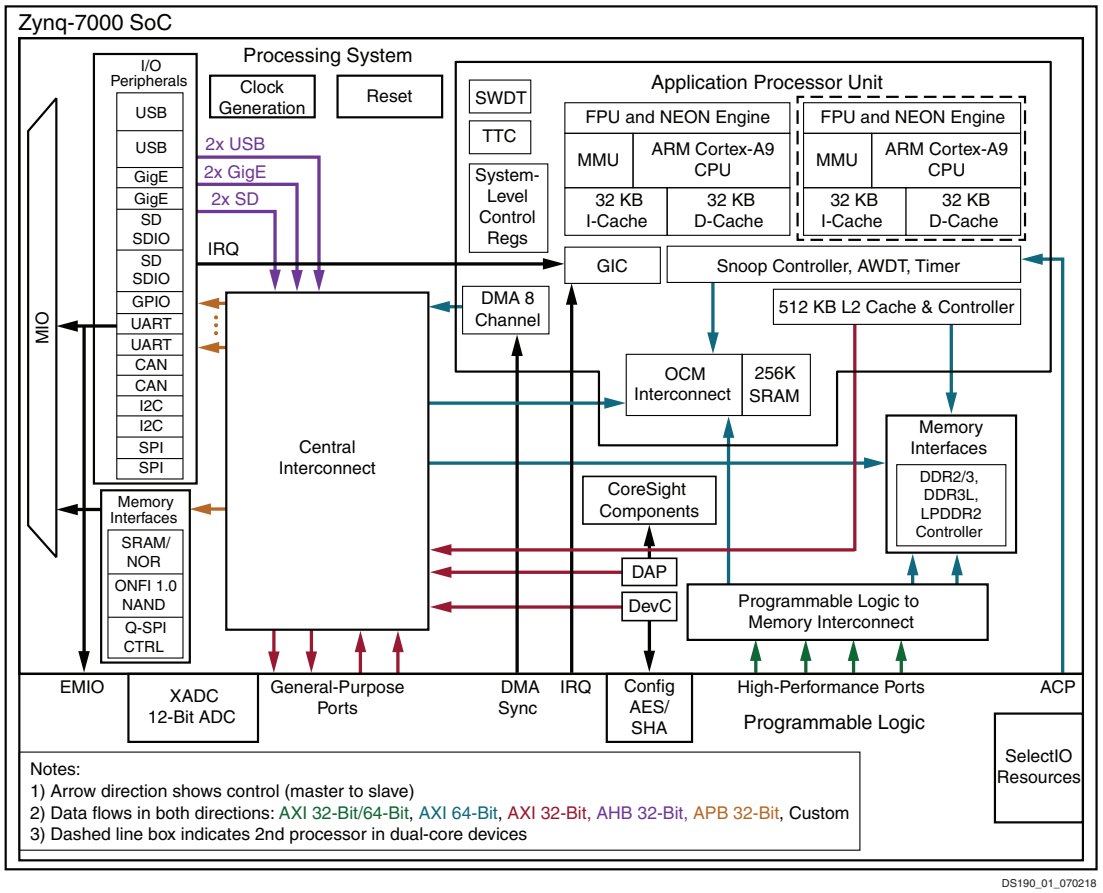

Figure 1.1: Zynq-7000 Architectural Overview, taken from [2]

An overview of the architecture of the Zynq-7000 platform is shown in figure 1.1. The overall architecture is divided into the processing system (PS) and the programmable logic (PL). The processing system contains two ARM Cortex-A9 cores with hardware floating point units. The memory access is accelerated by the two-level cache architecture. Through the Central Interconnect, the processors can access a set of MIO (Multi-Input-Output) signals like USB, GPIO and other standard interfaces.

The access from and to the programmable logic is possible by a set of interfaces, which provide different properties. For example, the ACP-interface (accelerator coherency port) supplies cache coherent access to the memory. Other ports like the high-performance ports allow faster access to the memory without coherency. The AXI-interface (Advanced eXtensible Interface) is another standard port for the connection of peripheral-modules. This interface is generalized in the figure 1.1 by general-purpose ports. Interrupt signals allow the interruption of the processor by events occurred in the programmable logic. The listed features make SoC FPGAs a flexible platform for hardware-software co designs.

The configuration of the programmable logic is done via the DevC (Device Configuration Interface) interface which implements the PCAP (Processor Configuration Access Port)

protocol for the configuration. This interface is abstracted by the Linux kernel through the `/dev/xdevcfg` device driver which can be used to read and write configuration files, so called bitstreams. This also includes the dynamic partial reconfiguration, where parts of the programmable logic are reconfigured during run time. The remaining parts of the programmable logic continue to work uninfluenced during this time.

Additionally, many embedded systems are needed to meet deadlines, which are set by the user or by the environment. For the reliable fulfillment of deadlines, the system must provide a deterministic system behavior. These systems are categorized as real-time systems. A classification of these systems is done either by the consequences after a missed deadline or by the way the task is called up. The detailed description of real-time system is presented in the background chapter 2.

1.1 Problem Definition

According to the project description for this thesis, the overall aim is the construction of an real-time demonstrator by using the ReconOS operating system. This real-world example should be based on an existing Ball-on-Plate demonstrator. The existing demonstrator is described in the background chapter 2.4.

For further investigations, the demonstrator should provide a set of threads which consists of periodic and non-periodic threads. These threads should also differ according to the hardness of the real-time conditions. On the other hand, most of the tasks should be implemented both in software and in hardware.

Besides the construction of the demonstrator, the following questions should be answered. Most of these questions are demonstrator specific. However, approaches for the providing of upper bounds for the execution times or the determination of overheads by system calls can be answered generally.

Q1 *How are tasks interfaced with sensors and actuators?*

Q2 *How to provide upper bounds for hardware and software task execution times?*

Q3 *How to model and determine the overheads posed by operating system calls and the hardware/software communication in ReconOS?*

Q4 *How to deal with resources shared between tasks, in particular buses and memories?*

Q5 *What degree of parallelism can and should be used, and what are the resulting scheduling and placement problems?*

Q6 *Is there a case for partial reconfiguration?*

1.2 Aim of the Thesis

The overall goal of this thesis is the creation of a real-world example for a demonstrator based on ReconOS. This demonstrator should provide a setup for investigations on ReconOS, especially regarding the real-time behavior.

These extensions will allow the demonstrator to provide a set of tasks with different characteristics regarding real-time requirements and scheduling behavior. This makes it possible to answer the questions from the problem definition.

In summary, this work is intended to provide insights that can be used in later developments of real-time systems based on a hardware software design with ReconOS to assess the system behavior. Additionally, the developed demonstrator can be used for further investigations on ReconOS and real-time systems.

1.3 Structure of the Thesis

The structure of this thesis is as described in the following. After the introduction chapter 1, the needed background for the understanding of this thesis is presented. This includes the ReconOS operating system and programming model, general real-time systems and the current state of the existing Ball-on-Plate demonstrator, which is extended.

After that, the extensions are presented in chapter 3. This includes the mechanical and electrical extensions but also changes on the control program and the video processing units. Due to these extensions, the demonstrator can be used as an experimental setup for following chapters.

Investigations regarding the real-time behavior are described in chapter 4. This contains the enhancement of ReconOS regarding priority-based scheduling but also measurements of the execution and communication times. The effect of parallel usage of the ReconOS infrastructure is measured later in this chapter. Additionally, four possible schedules for the execution of the threads are presented.

The following evaluation chapter 5 analyzes the implementations and the performance of the demonstrator. For this purpose, the improved real-time capabilities are being investigated. Additionally, the run times of the different implementations are shown. The thesis finishes with the last chapter 6, which conclude the thesis and give an outlook for future work.

2 Background

This chapter introduces the basic background, which is helpful for further reading. First, multitasking on FPGAs is introduced, as this is made possible using partial reconfiguration. After that, the ReconOS operating system is described, which is used for the implementation of the demonstrator hardware and software design. An introduction into the topic of real-time systems follows since their characteristics are considered in this work.

A demonstrator is being developed for these investigations in combination with ReconOS. This demonstrator is based on an already existing demonstrator, which has been described in a previous thesis [34]. The essential properties and the function are summarized in the last part of this chapter.

2.1 Multitasking on FPGAs

In the recent years, some procedures have been presented that enable multitasking on FPGAs. The overall goal of multitasking on FPGAs is to use at least parts of the FPGAs for different tasks concurrently. For this, dynamic (partially) reconfiguration is used which allows to reconfigure parts of the FPGA during run time. This technique enables to split large designs into smaller parts and execute them sequentially. During the reconfiguration, the untouched parts of the FPGA can be executed without interruption.

As like multitasking on a CPU, a distinction is made here between preemptive and cooperative multitasking. In cooperative multitasking, the task to be interrupted must actively return the processing unit. The scheduler then decides to which thread the processing unit is assigned based on a defined scheduling policy. If the currently active thread has a state or context that is necessary for later reentrance, it must be saved by the thread itself. This can be done for example in the global memory of the processing system or in additional local memory, which is added for context saving. The context contains the value of registers and local memory in the FPGA. Whereas in cooperative multitasking the thread generally only must save the necessary part of the context, a distinction is generally not possible in preemptive multitasking.

A more advanced multitasking procedure represents the preemptive multitasking on a FPGA. In this case, the context of the processing must be able to be read from outside. Therefore, different techniques are described. The approach of Readback, Scanpath and Multi-Context FPGAs are presented in the following.

- **Readback** In the Readback approach, the same configuration interface which is already used for the writing of the configuration data in the programmable logic, is used to read the so called bitstream back in the processing system memory. This

approach is used for example by Simmler et al. [36] or Happe et al. [21]. The readback includes not only the written bitstream but also the state of the registers and the local memory in the programmable logic.

- **Scanpath** Another possibility for the saving and restoring of the FPGA context is the usage of Scan-path registers. For example, this approach is used by Jovanovic et al. in [25]. Jovanovic extends the classical n-bit register, which is a combination of n-flip-flops to preemptive flip-flops. The difference comparing to the classical register structure is the support of an additional mode of operation. This additional mode connects the n-flip flops of the register to a shift register, which can be used to shift the current value of register clockwise out of the register. The restoration of the saved state can be done in the same way. The advantage of this approach is that no further support by the FPGA is needed since the logic can be built with standard logic elements. On the other hand, this approach requires additional logic. However, the proposed paper does not describe how to deal with other components like on-chip memory.

- **Multi-Context FPGAs** Another approach for the context switch in the FPGA are multi-context FPGAs, which can contain different sets of configuration data. An example architecture is proposed by Chong et al. in [13]. The biggest advantage of this method is the fast context change, because the different configurations are already written into the FPGA at the beginning. The disadvantage lies in the permanent availability of all configurations and the resulting usage of resources. in addition, storing elements such as registers and memories must be configured for each context in parallel, which also results in an inefficient use of resources. The availability of multi-context FPGA architectures is also very limited.

2.2 ReconOS

Due to the demand for more powerful and energy-efficient solutions for embedded systems, functional software parts must be moved from pure software implementations to hardware/software co-designs. The results of this decomposition are functional software and hardware parts, which have to communicate and synchronize with each other and often with a operating system like Linux. In pure software systems, this communication is widely standardized by the operating system, e.g. by the POSIX (Portable Operating System Interface) standard. As opposed to that, hardware functions often use application specific implementations without any standardization. This fact leads to longer development times and more error sources during implementation.

In order to counter this issue, hardware functional units are partially modeled as hardware-threads. However, they must be made accessible to the operating system, for which various approaches have been developed in recent years [16].

The aim of these approaches is to reduce the productivity gap between the rising density of logic and the slower increasing productivity of the development process. Approaches like HThreads[5], R3TOS[24], FUSE[23] use an adopted Pthreads model for the abstraction of the application dealing with the operating system [16].

The term Hthreads stands for the hybrid thread programming model for heterogeneous processing units consisting of CPU and FPGA. Hthreads introduces a novel high-level programming approach and includes also operating system and middleware abstraction in hardware. The scheduling and dispatching of the system is done in hardware, which leads to a minimal latency and a maximal performance on the CPU, since operating system functions does not have to executed on it.

Other approaches like SPREAD[40] and RTSM[12] use the abstraction with delegated threads, which represent the hardware application for the operating system.

SPREAD is specially developed for streaming applications, for example for hardware encryption between two software threads [40]. Hardware threads can be chained by streaming interfaces or to the multi-port memory controller, which allows access to the global memory. The control of the hardware threads is done via a control path interface, which is connected to the system bus of the CPU. SPREAD allows dynamic partially reconfiguration, which enables the changing of the streaming architecture during run time.

Another of these delegated thread-based approaches is ReconOS, which was proposed in 2007 [30] and is currently available in version 4.0. ReconOS combines a programming model, a hardware environment and host operating system support services. It also provides standardized communication and synchronizing mechanisms either for hardware-to-hardware communication but also for hardware-to-software communication.

ReconOS Programming Model

ReconOS supports either pure software but also hardware threads, which are used as an abstraction for a specific hardware functionality in the programmable logic. It is based on the idea of delegated threads, in which a delegated thread represents a single hardware thread in the operating system. Every interaction between a hardware thread and the operating system is done via the delegated thread assigned to the hardware thread. This enables the hardware thread to interact system-wide like a software thread through this standard interface.

This allows ReconOS hardware threads the usage of standard primitives like mutexes and semaphores for the communication and synchronization with other threads. Because the delegated thread acts as a standard POSIX software thread, principally all other standard system calls can be executed by the hardware thread. The standard system calls, which are already implemented, are [4]:

- Mutex (Try)-Lock, Unlock
- Semaphore Post, Semaphore Wait
- Conditional Variable Wait, Signal, Broadcast

Additionally to the implemented standard system calls, there are some ReconOS specific calls, which e.g. allow the hardware thread to get the address of initial data for processing or to signal the exit of the thread. By the fact, that all ReconOS threads (hardware and software) run within a shared memory space, standard systems calls are used to im-

plement a mailbox communication interface, which allows message-based communication between all threads. For that, the operations (try)-get and (try)-put are implemented.

Each delegated thread has a table of resources that can be used to assign an object such as a mailbox to a specific identifier. This identifier is propagated by an interface instead of the real object. The identifier is therefore not system-wide compliant, but only in the context of the specific hardware thread. This is one reason why the used bitstream must be compatible with the current running ReconOS software application. Specially, this must be considered if dynamically partial reconfiguration is used. The identifier of the different threads must be consistent.

ReconOS Global Hardware Design

ReconOS allows both soft-CPU cores and hard-CPU cores for its operation. For example, the current framework 4.0 supports the Xilinx Zynq platform with ARM Cortex-A9 CPU but also the Xilinx Microblaze soft-CPU core architecture.

At application build time, the framework automatically generates a global hardware design around the processor, which provides necessary functionalities for the hardware threads. The global hardware design of ReconOS provides every hardware thread an interface for operating system functions (OSIF, Operating System Interface) and for shared memory access (MEMIF, Memory Interface). Additionally, other needed components like reset controllers for the hardware threads and a configurable clock supply are also inserted into the hardware architecture.

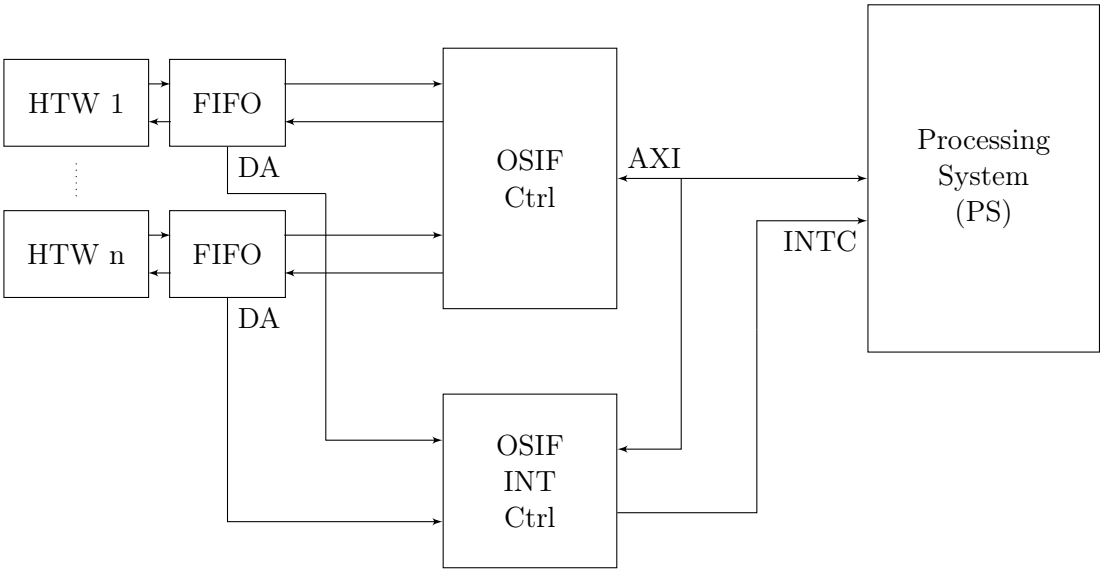

Figure 2.1: ReconOS OSIF Infrastructure

The OSIF interface allows hardware threads to use the standard operating system functions for communication and synchronization mechanisms. Figure 2.1 shows the interconnection between n-hardware threads and the processing system. Each of the hardware threads is bidirectionally connected to the OSIF AXI-controller device via a FIFO (First In - First Out) buffer. The "data available" (DA) output of the FIFO is connected to

the OSIF interrupt controller, which is connected to the interrupt input of the processing system. Both the OSIF controller and the OSIF interrupt controller provide an AXI-slave interface to get accessed by processing system.

If one or more hardware threads want to perform a communication through the OSIF interface to the processing system, it sends the system call command and the necessary data (e.g. a mailbox identifier) to its FIFO. The FIFO signals the OSIF interrupt controller that a new request has been received and forwards the data to the OSIF controller. The interrupt output of the OSIF interrupt controller is then set and an interrupt in the processing system is triggered.

The process within the processing system is shown in the figure 2.2. As soon as the interrupt is triggered in the processing system, the kernel driver reads the OSIF interrupt controller and detects which thread triggered the interrupt. The kernel driver then unblocks the corresponding delegated thread and provides the received data to the delegated thread. This thread executes the function which is mapped to the command in behalf of the hardware thread. This execution may results in a block or in a direct return. The result of the execution is then transmitted through the OSIF to the hardware thread.

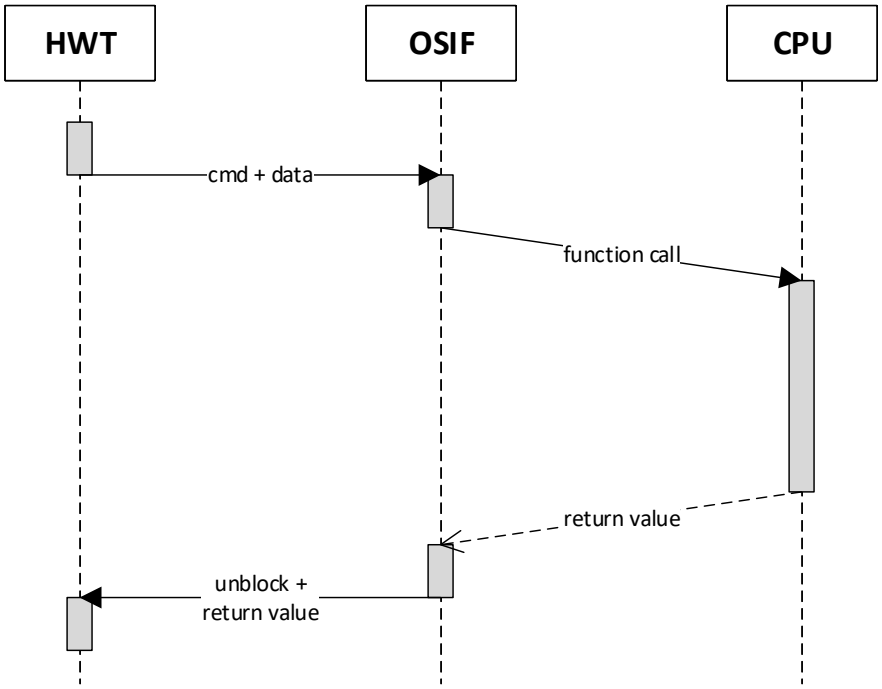

Figure 2.2: Sequence Diagram of the Hardware-Thread System Call, taken from [31]

Additionally, hardware threads can access the virtual address space of the running ReconOS application by the MEMIF interface. The structure of the Memory Interface is shown in figure 2.3. In the figure, the presence of n hardware threads is assumed again. For space reasons, the FIFOs between the hardware threads and the arbiter have been omitted in the figure, as they are not necessary for understanding the function.

Since concurrent memory access from different hardware threads is possible, the MEMIF arbiter must schedule the available memory bandwidth to the accessing hardware threads.

The scheduling policy is Round-Robin, which allocates the same communication time to all memory accessing threads.

Since both software and hardware threads should be able to access the same virtual address space, the requested addresses must be translated into physical addresses, which are needed for the access in the main memory. Due to that, an MMU (Memory Management Unit) is included into the interface.

The memory control unit (MEMIF CTRL) acts as master for the ACP-interface. Through this interface, the memory control unit has a cache coherent access to the main memory of the processing system but also to the rest of the physical address space. Therefore, not only the main memory is accessible but also other physical address spaces like the AXI peripheral interface of the processing system.

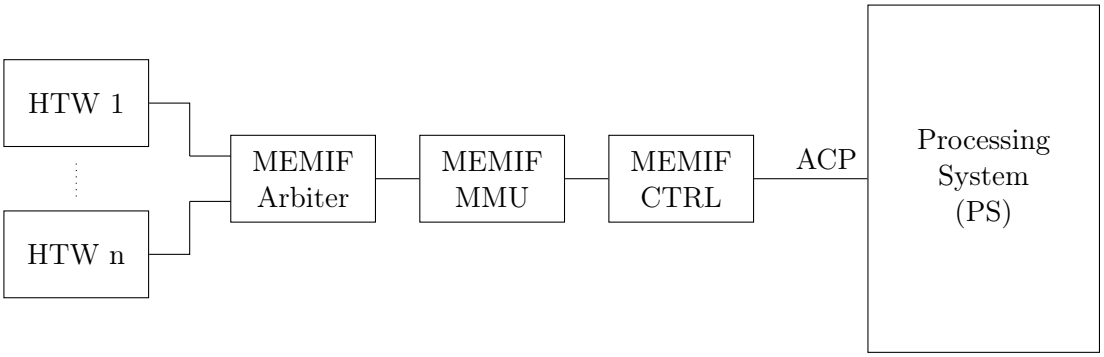

Figure 2.3: ReconOS Memory Interface Structure

2.3 Real-time Systems

There are numerous definitions for real-time systems in literature. Some of these definitions aim at the presence of deadlines by which the processing should be completed [29]. The consequences of missing such a deadline lead to a classification of real-time behavior. If the deadline is missed, this will only lead to a loss of quality, for example when decoding a video in a playback device, this is widely referred to as a soft real-time system.

On the other hand, if missing this deadline can lead to catastrophic consequences, it is a hard-real-time system. This is the case, for example, with the control of an aircraft where a missing of the deadline could lead to a plane crash. A more precise definition of the two categories is given by Reghenzani et al. in [33]. Provided a unknown distributed random execution time X for a real-time system and a given deadline D, hard real-time systems have to fulfill the condition from equation 2.1. The probability that the system meets the required deadline must be one.

$$P(X \leq D) = 1 \tag{2.1}$$

Soft real-time systems have to fulfill this condition with an accepted probability p, which is mainly a trade-off between costs and value (equation 2.2).

$$P(X \leq D) \geq p \tag{2.2}$$

Besides the definition about the consequences of a missed deadline, real-time systems can also be classified by the task triggering mechanism. In the field of reactive systems, there exists mainly two different program paradigms: event-triggered systems and time-triggered systems. Specially time-triggered systems are used during this thesis for the control program. The possibility to reconfigure parts of the FPGA design through an external event provides event-triggered behavior for the demonstrator.

Time-triggered systems only support interrupts by a central timer, which starts a new execution session of the corresponding threads. In figure 2.4, a simple scheduling for a timer triggered system is shown. The vertical arrow symbolizes the start of a new cycle by the periodic timer. After the start of a new cycle, thread τ_1, τ_2 and τ_3 are executed sequentially. After the computation is done, the processor runs in an idle process or powers down and waits for the next timer interrupt. If available, non-periodic non-real-time tasks can also be executed during this time, whose reaction time is not critical for the entire system.

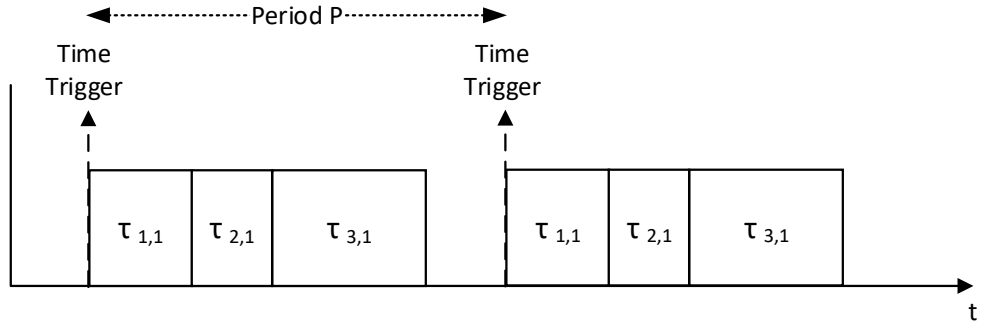

Figure 2.4: Simple Periodic Scheduling Example

The temporal control structure of a time-triggered system is designed off-line. Therefore, a task descriptor list (TDL) is generated, which includes the cyclic schedule of all threads [28]. The dispatcher is called by the timer interrupt and starts the threads regarding the TDL.

The criterion of scheduling feasibility is trivial in this case. The sum of worst-case execution time must be less or equal than the period time P to fulfill the requirements of hard real-time systems (equation 2.3). However, the determination of the worst-case execution times is the challenging part of the designing of such systems.

$$\sum_k WCET(\tau_k) \leq P \tag{2.3}$$

Due to the good analyzability of time-triggered real time systems, they are often used in safety-critical applications. Periodic timer triggered systems are also often used for signal processing applications, where periodic sample times are important for the quality of the data. However, time-triggered systems generate a considerable overhead because of the polling of the sensor data.

On the other hand, more advanced timer triggered systems also allow task specific periods instead of equal periods for all threads. In that case, checking whether a scheduling is feasible or not is more challenging than for simple periodic tasks.

The second group of program paradigms is the class of event-triggered systems, which schedule is based on events. These events are triggered by sensors or other tasks in the system. Therefore, tasks are only executed if an event occurs. Schedulers that support event-triggered threads require a dynamic scheduling strategy since the arrival of the tasks is not known during design time [27].

Real-time Operating Systems

Many real-time systems work without an operating system. The application is programmed directly for a specific microcontroller on bare metal. However, this type of implementation is more suitable for smaller systems, since the abstraction of the hardware by the operating system can be omitted. On the other hand, the handling and implementation of multiple threads with different periods and event-based execution without an underling operating system is a challenging task.

In general, there are different approaches to achieve real-time behavior in an operating system. On the one hand, there are operating systems that have been designed as real-time systems from the beginning. eCos or FreeRTOS are among these operating systems.

eCos (embedded configurable operating system) was designed for flexible configuration and adaption to different embedded systems. It contains a hardware abstraction layer (HAL), the eCOS kernel, an IO system and the standard C-library [38]. The kernel was designed for minimum interrupt latency, low task switching latency, small memory footprint and deterministic behavior.

Another real-time operating system is FreeRTOS, which was developed by Richard Barry [20]. It provides a scheduler which supports both cooperative and preemptive scheduling behavior, dynamic and static memory allocation and objects for message exchange and synchronization.

On the other hand, there are various approaches to general purpose operating systems like Windows and Linux to adopt real-time operating behavior. For example, some commercial vendors rely on the approach of a timer triggered system for Windows, where real-time tasks are cyclically triggered by the kernel timer. The remaining computing time, which is not used for real-time tasks, is made available to the windows scheduler [1].

For Linux, there are different approaches to ensure real-time behavior [35]. Two of these approaches are the Xenomai framework and RTAI extension, which both use the Adeos nanokernel. The nanokernel works between the hardware and the Linux kernel and acts as an arbiter for the hardware functionalities [8]. The applications, which should have real-time properties, are executed in parallel to the Linux kernel. Hardware interrupts are forwarded from the Nanokernel to the target application by a pipeline mechanism. The approach of RTAI is similar to that of Xenomai but offers the possibility to intercept interrupt directly without the nanokernel. The main disadvantage of these approaches is that the application must be adapted to the system architecture. For example, the

application must call special functions for synchronizing which are not compatible to the POSIX standard.

This disadvantage is at the same time a motivation to harden the real-time behavior of the standard Linux kernel. General purpose operating system like Linux are designed for maximum average throughput, not for deterministic behavior [33].

The default Linux kernel supports different levels of preemptive behavior, which can be selected at compile time:

- `PREEMPT_NONE`: Kernel functions may run without any interruption through user-space applications for maximum throughput.
- `PREEMPT_VOLUNTARY`: Explicit preemptive points allow user-space applications the interruption of kernel functions.
- `PREEMPT`: Except for spinlocks and some critical sections, preemption is always allowed.

For realizing a more deterministic behavior, several improvements have been made over the years. These changes are already supported by the `PREEMPT` compiler option. Some of changes are presented in the following. A more detailed listing is proposed in [33].

- **High-Resolution Timers** In 2006, the high-resolution timer subsystem (HR-timers) is introduced in the kernel sources by Gleixner and Niehaus [19]. Compared to previous timer precision, HR-timers can archive resolutions in order of nanoseconds instead of milliseconds. The subsystem stores timer events in red-black-tree structure, which enables fast access to the next expiring event. The `nanosleep()` function used in this thesis is based on this timer subsystem.
- **Priority Inheritance** Priority Inheritance is a mechanism for the elimination of priority inversion situations. Priority inversion can occur in situations where a thread with a low priority uses a resource that is also requested by a thread with higher priority. Since the high-priority thread has to wait in this case, a medium-priority thread could displace the low-priority thread. This would result in the medium-priority thread being executed preferentially compared to the high priority thread. The Priority Inheritance mechanism assigns the high priority to the low priority thread as long as it occupies the resource. On the other hand, this behavior can lead to unbounded latency.
- **Scheduler** The state-of-the-art Linux Scheduler, the CFS (Completely Fair Scheduler) was introduced in 2007. The scheduler supports the default scheduling policy `SCHED_OTHER` and two important real-time scheduling policies `SCHED_RR` and `SCHED_FIFO`. In 2010, the real-time scheduling policy `SCHED_DEADLINE` was added to the mainline kernel, which is based on the earliest deadline first (EDF) scheduling algorithm. The scheduler allows the assignment of priorities in the range of (1...99) for real-time tasks. The priority of normal threads is controlled by the nice-value, which is in the interval $(-20...19)$

Applying the real-time patch *PREEMPT_RT* expands the configuration options by the option `PREEMPT_FULL`. The patch leads to the removing of most of the spinlocks in kernel threads, which are not non-preemptive [33].

Finally, the question arises whether the patched changes on the kernel lead to more reliable real-time system regarding the meeting of deadlines. For this, an unknown distributed runtime X is compared to a latency-improved runtime X'. The implication, that a lower expected value $\mu' = \mathbb{E}[X']$ compared to the previous expected value $\mu = \mathbb{E}[X]$, does not hold (equation 2.4)[33].

$$\mu' < \mu \nRightarrow P(X' \leq D) \geq P(X \leq D) \tag{2.4}$$

Even for a smaller resulting variance of the distribution, the implication does not hold (equation 2.5)[33].

$$\mu' < \mu, \sigma < \sigma' \nRightarrow P(X' \leq D) \geq P(X \leq D) \tag{2.5}$$

Brown and Martin compare the real-time behavior of the standard Linux kernel, the patched Linux kernel and the Xenomai co-kernel approach in [11]. In this paper, a periodic test case in which an application has to trigger an output signal periodically and a test case in which the application has to wait a random interval and toggle an output signal afterwards.

The results of this test are that the co-kernel-based approach (Xenomai) still outperforms a single kernel approach. Due to the results, Linux is classified as an *95% hard real-time* System, which accepts deadline misses with a probability of equal or less than 5 percent [11].

The result of these experiments is that while the change allows sufficient real-time properties, it is less suitable for use as a hard-real-time system for safety-critical applications. However, for the purposes of this work the real-time properties are sufficient.

2.4 Ball on Plate Demonstrators

Over the last few years, several demonstrators have been developed in which a ball can be balanced on a surface (so called Ball-on-Plate). Bdoor et al. [9] describe a mechanical system with two degrees of freedom, which can balance a ball on its surface. The position of the ball is recognized trough vision detection and the two stepper motors are connected to the platform.

On the other hand, there are some more advanced Ball on Plate implementations, which use the Stewart-Platform [39] for platform moving. Most of the systems use a vision feedback for the control loop. In this case, a camera is mounted over the platform and the resulting video signal is used to detect the position of the ball on the platform. This approach is used by Copot et al. [15], which use a fractional PID controller for the position control. Yaovaja [41] use a fuzzy control-based approach for the position control. Arshad et al. [6] use a classical PID-controller for this purpose. Bang and Lee [7] use a touch-screen-based approach for the feedback of the balls position. The same approach is used by Ruething [34].

The Ball-on-Plate demonstrator presented in the master's thesis by Ruething is used as basis for further extensions and is therefore summarized in the following. The term Ball-on-Plate refers to the ability to balance a metal ball on its surface.

The demonstrator is a real-world implementation of a Stewart-platform, which was presented by D. Stewart in 1965 [39]. Due to the Stewart Platform, the surface of the demonstrator can move the platform with six degrees of freedom. This are linear translations in x, y and z direction but also three rotations (pitch, roll and yaw). This makes the platform ideal for applications such as flight simulators or telescopes. The demonstrator is shown in figure 2.5.

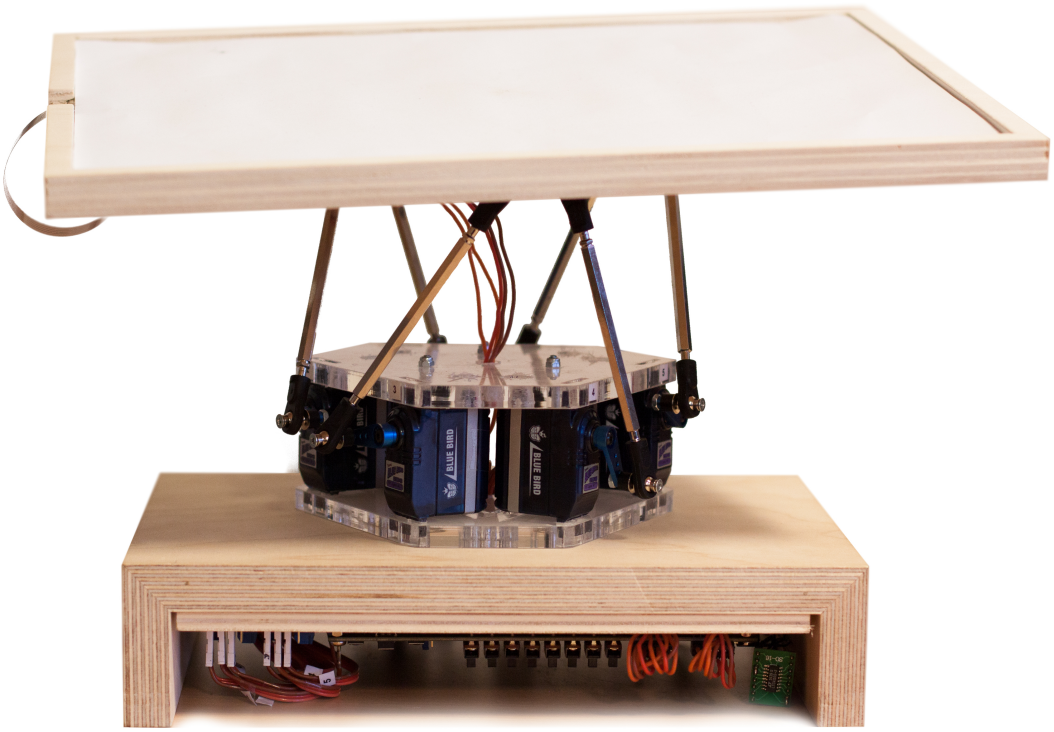

Figure 2.5: Ball-on-Plate Demonstrator, taken from [34]

Since linear drives are either more expensive (electric linear motors) or impracticable due to external auxiliary units (e.g. for hydraulic or pneumatic actors), the movements are realized with servo motors in this implementation. The rotation of the servo motor in combination with its leg and the rod substitutes the needed linear translation. The corresponding rod is mounted on the end of the leg of the motor and the other side on the platform. However, this replacement results in a more complex calculation of the motor angle compared to a linear position.

In order to balance a ball, the position of the ball on the surface must be registered. This demonstrator uses a resistive touchscreen on the surface of the platform. The touch screen is connected to the touch-screen-controller, which can be read out through a serial interface. This touch-screen-controller contains the analog-digital-conversion of the electrical signals from the touchscreen and converts them into a digital signal. The controller is also able to generate interrupts when the ball is placed on the surface.

The central control logic of the demonstrator is the Zedboard, a FPGA development board containing the Xilinx Zynq-7020 FPGA with two ARM Cotex-A9 hard CPU cores. This board executes ReconOS together with the Linux operating system and is mounted under the demonstrator.

The software and FPGA design implementation combine hardware and software-threads in the ReconOS framework. The architecture contains four different threads (Touch, Control, Inverse, Servo), of which control and inverse are implemented in both software and hardware. Touch and Servo are hardware-only implementations, because they must communicate through external interfaces like SPI (Serial Peripheral Interface) or provide the motor angle through pulse-width-modulation. An overview of the hardware software architecture is shown in figure 2.6.

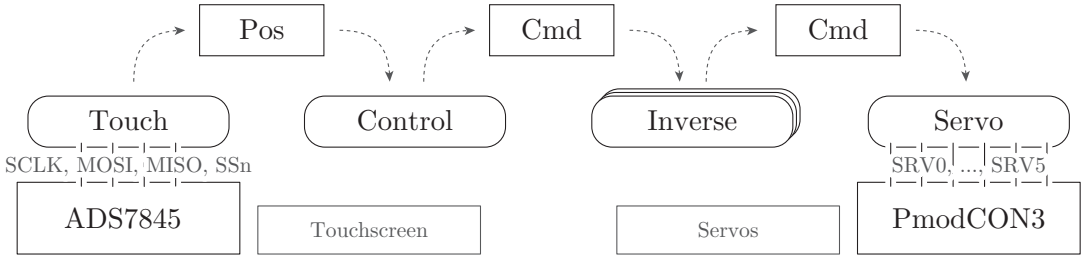

Figure 2.6: Control Software Architecture, taken from [34]

The touch thread starts a new control cycle with the reading of the actual position of the ball on the plate. This information is scaled and send to the Control Thread via the *Pos*-mailbox.

The control thread implements the control algorithm and put the result six times in the *Cmd*-mailbox, one for each servo motor. The control algorithm implements an adapted PID-controller, which is not further considered. A new controller design is shown in chapter 3.

The inverse thread, which can be instantiated six times for parallel execution or run six times sequentially, transforms the value in the *Cmd*-mailbox, into the required servo position and sends the data to the servo hardware thread, which gets the data and sets the motor position.

The original control program also contains a self-awareness mechanism, which adapts the cycle time to the dynamic off the ball. The regarding self-properties are power consumption, ball position and the output of the PID-controller and processing performance. These features are removed for further considerations, since self-awareness is not used in this thesis.

3 Demonstrator

This chapter describes the construction of the demonstrator setup. As mentioned, the existing Ball on Plate demonstrator is used and extended during this thesis. The extension leads to more workload for the control unit of the overall demonstrator and enables further real-time investigations. The most visible extension is the construction of two more Stewart-platforms with own touch-screen surfaces and six servo motors per platform.

Beside the mechanical extensions of the demonstrator, the existing FPGA development board (Zedboard) is used to run the control algorithms for all three platforms. Additionally, the demonstrator is extended by a video processing chain, which allows to process and output incoming video data. For this processing, a two filter kernels are implemented. For providing a event-triggered thread, the remote reconfiguration server accepts requests for filter changing by requests through the network interface. For sequential computing on the FPGA, partial reconfiguration is enabled by the mechanism described in sub chapter 3.3.

3.1 Mechanical and Electrical Extensions

The mechanical construction of the Ball-on-Plate platform is already described in the background chapter 2.4. For the construction of two further platforms, this mechanical construction was overtaken by the original platform except for a few smaller changes, such as the usage of other materials. The old and new dimensions of the Stewart-platform are shown in figure 3.1. The dimensions and the positions of the mountings e.g. of the legs are needed for the inverse kinematics, which are described during this chapter.

The electrical connection between the platforms is done via plug-able interfaces. This includes the connections for the servomotor control signals and the communication interface for the touch controllers. Therefore, the original platform can be used in the legacy mode without the extension. On the other hand, every platform has its own 5 Volt power supply for the servo motors, which is needed because of the significant power consumption of the servo motors.

Besides that, the demonstrator is enabled to process video data between two HDMI interfaces. Fortunately, the FPGA development board is already providing an HDMI output interface, which can be used for this purpose. The interface is controlled by the ADV7511 HDMI transmitter integrated circuit (IC) from Analog Devices, which leads to less resource consumption into the FPGA design and less critical output timing requirements due to lower output frequencies because of the parallel interface instead of a serial interface.

For providing a HDMI input interface, the Zedboard is extended by an FMC-HDMI input extension card. The FMC-Interface (FPGA Mezzanine Card) is a standardized

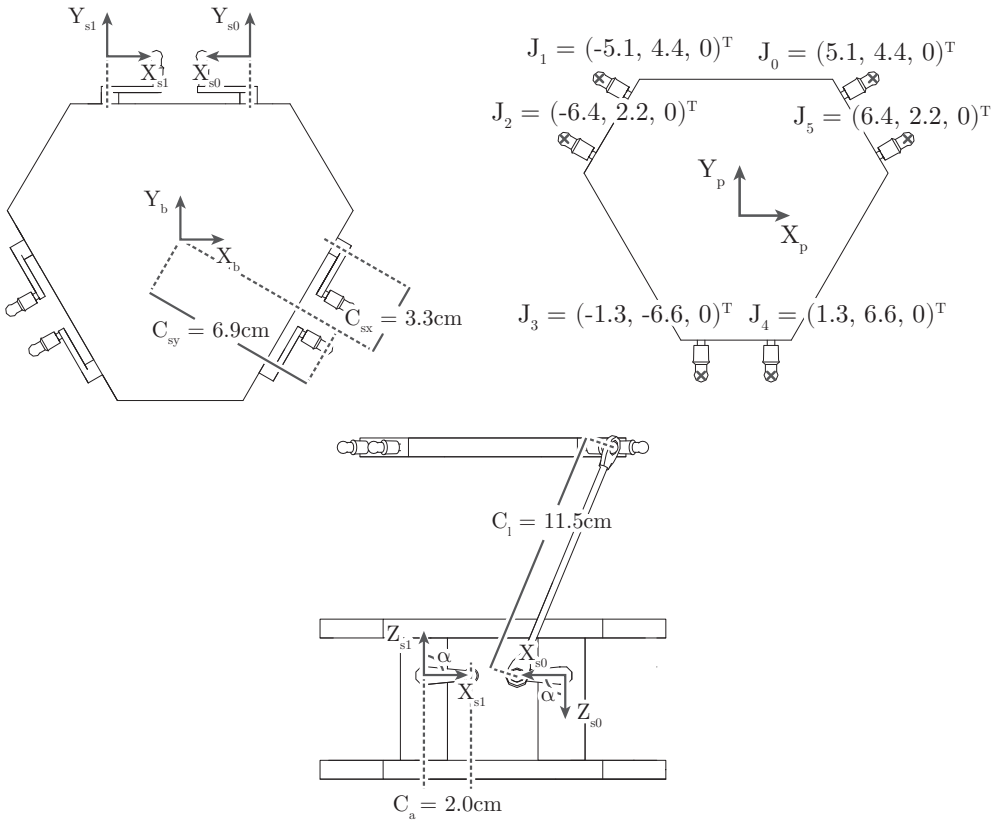

Figure 3.1: Notations for the Stewart-Platform, taken from [34]

extension interface for FPGA development boards or other devices with reconfigurable behavior. The HMDI-input-interface card provides two HDMI inputs. The first input is driven by the HMDI receiver chip ADV7611 from Analog Devices, similar to the output interface of the Zedboard. The second input interface provides only signal conditioning. All other parts of the HDMI specification have to be implemented into the FPGA, which results in a considerable task. Due to that, the first HDMI input port is used for the implementation. Analog Devices provides FPGA IP-cores and software kernel drivers for both HDMI transmitter and receiver circuits.

3.1.1 Hardware Modeling and Controller Design

For the controller design later this chapter, the dynamic system behavior of the demonstrator must be modeled. The dynamic representation of the behavior is done in a state-space model, which can be used for further considerations. The state-space model is a set of linear first-order differential equations, which are a common representation of dynamic systems in the field of control theory. The equations 3.1 and 3.2 show the general form of a state-space model.

$$\dot{\vec{x}} = \boldsymbol{A} \cdot \vec{x} + \boldsymbol{B} \cdot \vec{u} \tag{3.1}$$

$$\vec{\boldsymbol{y}} = \boldsymbol{C} \cdot \vec{\boldsymbol{x}} + \boldsymbol{D} \cdot \vec{\boldsymbol{u}} \tag{3.2}$$

The first step of the hardware modeling is the determination of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ and $\boldsymbol{D}$, which is presented in the following. $\boldsymbol{A}$ is the state-space matrix, $\boldsymbol{B}$ is the input matrix, $\boldsymbol{C}$ is the output matrix and $\boldsymbol{D}$ the feed through matrix.

The motion of the ball on the plate can be described by the Euler-Lagrange equation, where L is the difference between the kinetic Energy T and the potential Energy V [14]. $\boldsymbol{q}$ are the generalized coordinates of the observed system, in this case the ball coordinates x and y.

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\boldsymbol{q}}} - \frac{\partial L}{\partial \boldsymbol{q}} = 0 \tag{3.3}$$

From this equation follow the two equations of translation, which finally describe the system behavior. Through the approximation of the sine function with $\sin(\phi) \approx \phi$ for small angles ϕ, the non-linear system results in a linear differential equation system (equation 3.4). In the equation, m_b is the mass of ball, r_b the radius of the ball and g the gravitational acceleration.

$$\begin{aligned}(m_b + \frac{J}{r_b^2})\ddot{x}_1 + m_b \cdot g \cdot \sin(\alpha) \approx (m_b + \frac{J}{r_b^2})\ddot{x}_1 + m_b \cdot g \cdot \alpha = 0 \\ (m_b + \frac{J}{r_b^2})\ddot{x}_2 + m_b \cdot g \cdot \sin(\beta) \approx (m_b + \frac{J}{r_b^2})\ddot{x}_2 + m_b \cdot g \cdot \beta = 0\end{aligned} \tag{3.4}$$

After the substitution of J through the moment of intertia of a sphere $J = \frac{5}{2}mr^2$, the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \boldsymbol{B} = -\frac{5}{7} \cdot g \cdot \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.5}$$

The state vector $\vec{\boldsymbol{x}}$ contains the position and the velocity in both directions on the platform (equation 3.6).

$$\vec{\boldsymbol{x}} = \begin{pmatrix} x \\ v_x \\ y \\ v_y \end{pmatrix} \tag{3.6}$$

The output matrix $\boldsymbol{C}$ determines the values of the system state, which can be measured from outside of the system. Since the demonstrator enables the measurement of the position vector and not the velocity, the resulting output matrix is:

$$\boldsymbol{C^T} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \tag{3.7}$$

The last matrix of the model is $\boldsymbol{D}$, which is set to $\boldsymbol{D} = 0^{2\times 2}$, since the system has no direct feed through. The state-space model for the demonstrator is used later during this chapter for the design of the Kalman filter and for finding stable parameters for the controller.

Inverse Kinematics

The Stewart-platform is controlled by the six angles $\vec{\boldsymbol{\gamma}} = (\gamma_0, \gamma_1, ..., \gamma_5)$ of the servo motors. These angles are set by the control program. By the kinematic of the platform $\boldsymbol{J}$, the angle pair α and β are set for a given vector $\vec{\boldsymbol{\gamma}}$ (equation 3.8).

$$(\alpha, \beta) = \boldsymbol{J}(\vec{\boldsymbol{\gamma}}) \tag{3.8}$$

Unfortunately, the angle pair (α, β) cannot be modified directly from the control program. This leads to the nonlinear system model in equation 3.9, since the control program can only influence the vector $\vec{\boldsymbol{\gamma}}$.

$$\dot{\vec{\boldsymbol{x}}} = \boldsymbol{A} \cdot \vec{\boldsymbol{x}} + \boldsymbol{B} \cdot \boldsymbol{J}(\vec{\boldsymbol{\gamma}}) \tag{3.9}$$

The controller, which is developed during this work, outputs the desired angle pair (α, β). Therefore, the inverse kinematic $\boldsymbol{J^{-1}}(\alpha, \beta)$ is needed, that must meet the condition from equation 3.10.

$$(\alpha, \beta) \stackrel{!}{=} \boldsymbol{J}(\boldsymbol{J^{-1}}(\alpha, \beta)) \tag{3.10}$$

The inverse kinematic $\boldsymbol{J^{-1}}(\alpha, \beta)$ accepts the desired angle pair (α, β) and outputs the needed angles for the Stewart-platform. Due to the usage of the inverse kinematic, the non-linear system model is transformed into a linear system model again (equation 3.11).

$$\dot{\vec{\boldsymbol{x}}} = \boldsymbol{A} \cdot \vec{\boldsymbol{x}} + \boldsymbol{B} \cdot \boldsymbol{J}(\boldsymbol{J}^{-1}((\alpha, \beta)^T) = \boldsymbol{A} \cdot \vec{\boldsymbol{x}} + \boldsymbol{B} \cdot (\alpha, \beta)^T \tag{3.11}$$

Another aspect of the mathematical description of the system behavior is the coordinate transformation depending on the angles α and β. The angle α describes the rotation around the y-axis and the angle β the rotation around the y-axis. The rotation is shown in figure 3.2. Coordinates from the coordinate system rotated with α and β can be transferred to the base coordinate system of the demonstrator by the sequential multiplication of two rotation matrices $\boldsymbol{R}_x(\alpha)$, $\boldsymbol{R}_y(\beta)$ or just by the multiplication with $\boldsymbol{R}_{x,y}(\alpha, \beta)$, since the resulting rotation around the x-axis and y-axis is the product of both rotation matrices (equation 3.14) [10].

$$\boldsymbol{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix} \tag{3.12}$$

$$\boldsymbol{R}_y(\beta) = \begin{pmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{pmatrix} \tag{3.13}$$

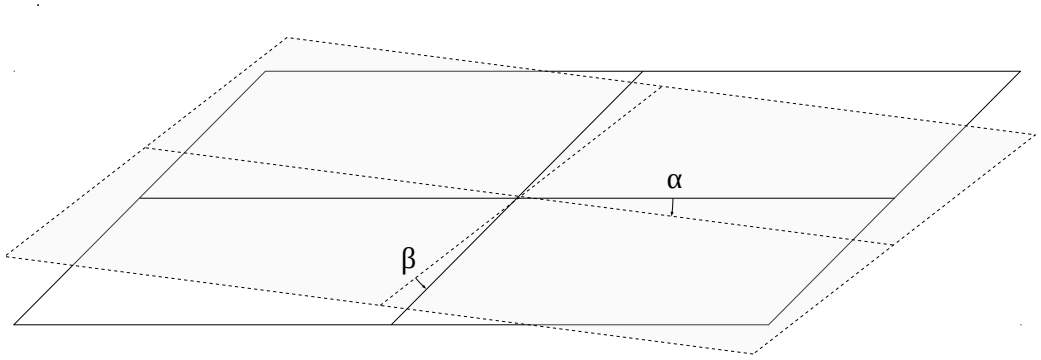

Figure 3.2: Rotation around x- and y-Axis

$$\boldsymbol{R}_{x,y}(\alpha,\beta) = \boldsymbol{R}_x(\alpha) \cdot \boldsymbol{R}_y(\beta) \tag{3.14}$$

The rotation matrix from equation 3.14 is used in the calculations for the inverse kinematics to transform the positions of the rods in the base coordinate system of the demonstrator.

Controller Design

The described system modeling is now used to design a position control for the ball. The demonstrator's existing controller worked with non-equidistant sampling times. In order to enable a predictable processing of the control, a constant sampling interval is implemented in this work. Thus, a controller design according to known methods can be realized afterwards.

The system behavior including inverse kinematics is generally a double-integral behavior, which is typically controlled by a PD-controller. Since the PD-controller has a strong differential behavior, it reacts sensitively to noise and interference. Therefore, a Kalman filter [26] is implemented in the controller, which provides an optimal position estimation and reduces or even prevents noise and disturbances. State observers like the Kalman filter can only be realized if the system to be observed fulfills the condition for observability. This condition is checked with an appropriate software and should be considered as given in the following.

The difference between the reference position and the estimated position is then fed to the PD-controller. There are extended methods for controller design, for example the design by pole assignment [3]. Since the focus of this work is not on advanced controller design, simpler iterative methods are used. The structure is shown in figure 3.3.

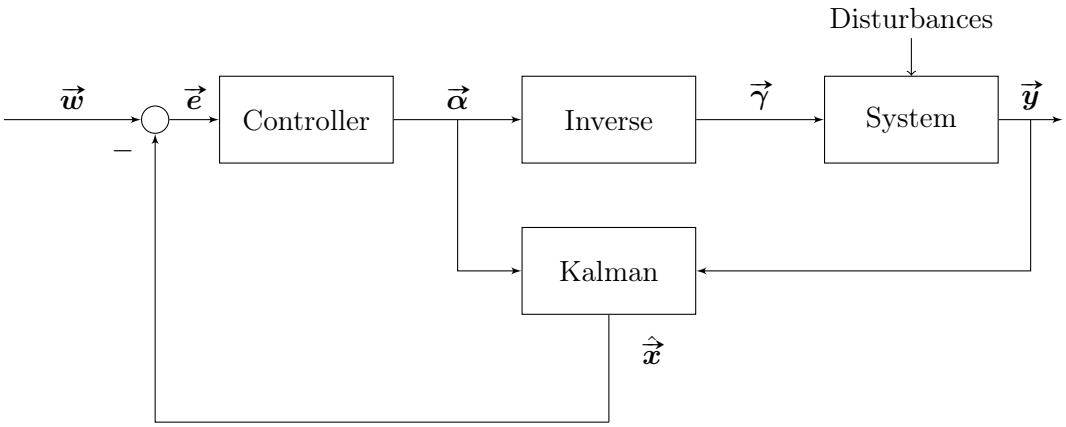

Figure 3.3: Control Loop Model of the Demonstrator

The Kalman filter calculates an optimal estimate for the position of the ball. For this, two different steps are processed. However, first of all the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ have to be discretized to calculate the filter. The zero-order hold estimation is based on the approximation $\dot{x} \approx T_A^{-1}(x_{k+1} - x_k)$ and leads to the following matrices $\boldsymbol{A}_d$ and $\boldsymbol{B}_d$.

$$\boldsymbol{A}_d = \boldsymbol{I} + \boldsymbol{A} \cdot T_A \qquad \boldsymbol{B}_d = \boldsymbol{B} \cdot T_A \tag{3.15}$$

The resulting discrete model of the system is then given by equation 3.16. For the matrix $\boldsymbol{C}$, there are no changes due to the discretization.

$$\begin{aligned} \vec{\boldsymbol{x}}_{k+1} &= \boldsymbol{A}_d \cdot \vec{\boldsymbol{x}}_k + \boldsymbol{B}_d \cdot \vec{\boldsymbol{u}}_k \\ \vec{\boldsymbol{y}}_k &= \boldsymbol{C} \cdot \vec{\boldsymbol{x}}_k \end{aligned} \tag{3.16}$$

In the first processing step of the Kalman filter, the state of the ball is predicted with the input vector (α, β) (3.17). Furthermore, the error covariance matrix $\boldsymbol{P}_{k-1}$ of the last processing cycle is updated with the discrete state-space matrix and the covariance matrix of the process noise $\boldsymbol{Q}$ (3.18).

$$\vec{\boldsymbol{x}}_{k|k-1} = \boldsymbol{A}_d \cdot \vec{\boldsymbol{x}}_{k-1} + \boldsymbol{B}_d \cdot (\alpha, \beta)_{k-1} \tag{3.17}$$

$$\boldsymbol{P}_{k|k-1} = \boldsymbol{A}_d \boldsymbol{P}_{k-1} \boldsymbol{A}_d^T + \boldsymbol{Q} \tag{3.18}$$

In the second step of every processing cycle, the measured position of the ball is used to correct the simulated filter position. The first part of that is to calculate the Kalman gain (3.19), which is a measure for the effect of the measurement on the simulated state. The variance of the measurement $\boldsymbol{R}$ influences the measurement inverse proportionality, which supports the previous statement.

$$\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T (\boldsymbol{H}_k \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T + \boldsymbol{R})^{-1} \tag{3.19}$$

The Kalman gain is then used to correct the system state $\vec{\boldsymbol{x}}_k$ and the error covariance matrix $\boldsymbol{P}_k$ (equation 3.20 and 3.21).

$$\vec{\boldsymbol{x}}_k = \vec{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k(\vec{\boldsymbol{z}}_k - \boldsymbol{H}_k\boldsymbol{x}_{k|k-1}) \tag{3.20}$$

$$\boldsymbol{P}_k = (\boldsymbol{I} - \boldsymbol{K}_k\boldsymbol{H}_k)\boldsymbol{P}_{k|k-1} \tag{3.21}$$

The presented Kalman filter needs for the calculation of the gain matrix $\boldsymbol{K}_k$ an inverting of the matrix expression $(\boldsymbol{H}_k\boldsymbol{P}_{k|k-1}\boldsymbol{H}_k^T + \boldsymbol{R})$. Algorithms for the calculation of inverse matrices generate a considerable computational effort, which is why a related approach is chosen in this thesis.

The mentioned approach of sequential Kalman filters is described in [37]. The sequential Kalman filter splits the correction part with the r-dimensional vector $\vec{\boldsymbol{z}}_k$ measured values into r single correction steps with one single measurement value $z_{k,j}$ each. This is possible if and only if the covariance matrix variance of the measurement $\boldsymbol{R}$ is a diagonal matrix. On other words, all measurements have to be independent from each other.

In the case of the Ball-on-Plate demonstrator, the measurement of the position components x and y is assumed as statistical independent and equal which leads to the following matrix structure for $\boldsymbol{R}$.

$$\boldsymbol{R} = \sigma^2 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \tag{3.22}$$

The constant factor σ^2 is determined to 10 by iterative testing. The covariance matrix of the process $\boldsymbol{Q}$ is also assumed as statistical independent. The factor q is set to 16.

$$\boldsymbol{Q} = q \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.23}$$

Since the measurement errors are assumed as statistically independent, the correction process of the Kalman filter is done sequentially [37]. Therefore, the following calculations have to be done for $i = 1..r$. r is the dimension of the output vector $\vec{\boldsymbol{y}}$ $(r = 2)$.

$$\boldsymbol{K}_{ik} = \frac{\boldsymbol{P}_{i-1,k}\boldsymbol{H}_{ik}^T}{\boldsymbol{H}_{ik}\boldsymbol{P}_{i-1,k}^+\boldsymbol{H}_{ik}^T + R_{ik}} \tag{3.24}$$

$$\vec{\boldsymbol{x}}_{ik}^+ = \vec{\boldsymbol{x}}_{i-1,k}^+ + \boldsymbol{K}_{ik}(y_{ik} - \boldsymbol{H}_{ik}\vec{\boldsymbol{x}}_{i-1,k}^+) \tag{3.25}$$

$$\boldsymbol{P}_{ik}^+ = (\boldsymbol{I} - \boldsymbol{K}_{ik}\boldsymbol{H}_{ik})\boldsymbol{P}_{i-1,k}^T \tag{3.26}$$

After r iteration the resulting a priori estimation of the state and the error covariance matrix is the result of the last iteration.

$$\vec{\boldsymbol{x}}_k^+ = \vec{\boldsymbol{x}}_{rk}^+, \boldsymbol{P}_k^+ = \boldsymbol{P}_{rk}^+ \tag{3.27}$$

The simulated behavior of the Kalman filter is shown in figure 3.4. For the figures, the step-response of the system model is predicted and corrected with the Kalman filter. The unstable behavior of the controlled system is also visible. For the initialization of the Kalman filter, the matrices are set to $\boldsymbol{P}_0 = \boldsymbol{I}$ and $\vec{\boldsymbol{x}}_0 = 0$.

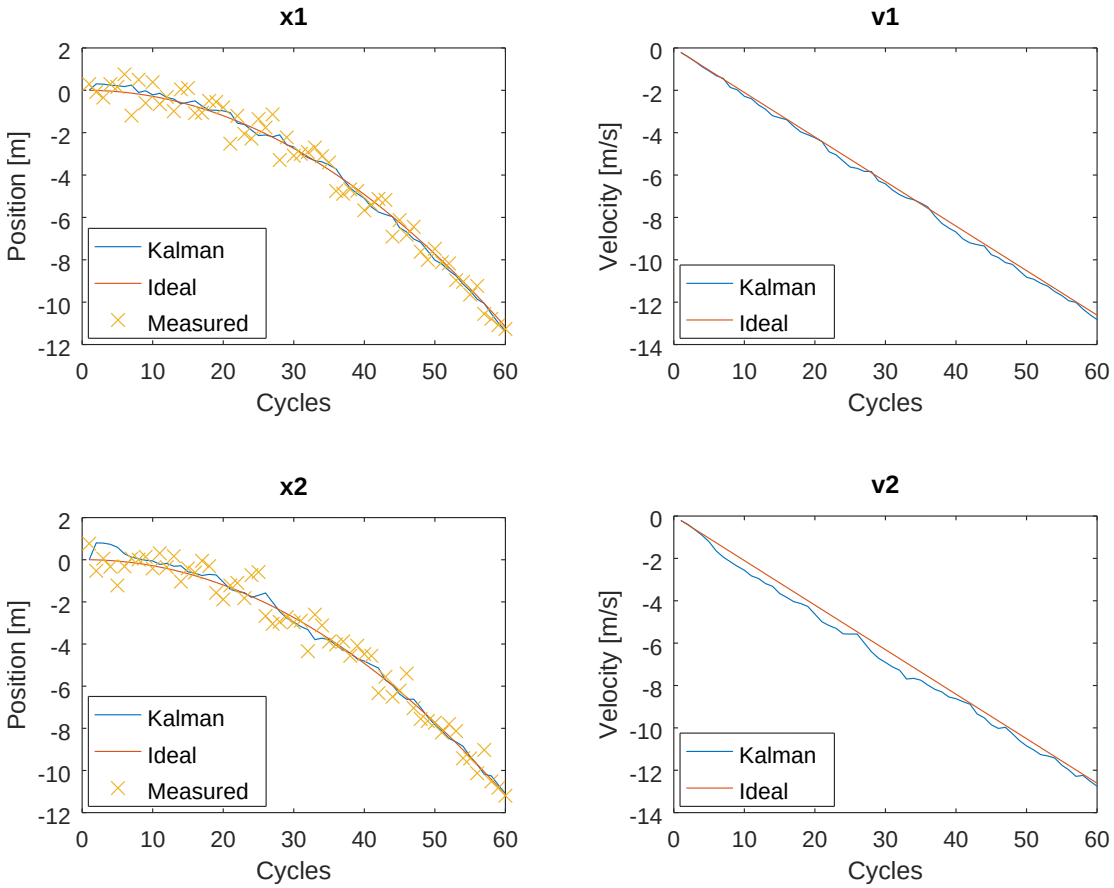

Figure 3.4: Simulated Kalman Step Reponse

After position estimating, the state-vector $\vec{\boldsymbol{x}}$ is used for the calculation of the control variables α and β. The controller is a PD-controller which reacts proportionally both to the control deviation and to the derivative of it. The transfer function of the controller is given in the equation 3.28 and equation 3.29 [32]. The constant T_A is the sample period of the discrete control system.

$$G_R = b_0 + b_1 \cdot z^{-1} \tag{3.28}$$

$$b_0 = K_R(1 + \frac{T_V}{T_A}), b_1 = -K_R \cdot \frac{T_V}{T_A} \tag{3.29}$$

The challenging part of the control design is to determine the constants K_R and T_V, such that the controller provides at least stability and sufficient stationary accuracy [17]. Other properties like damping of the control loop and dynamic behavior are second order constraints.

A further limitation in the controller design are limits with regard to the control variable $\vec{\boldsymbol{u}}$. The Ball-on-Plate platform from this work is limited to about 11 degrees for each alpha and beta because of mechanical construction of the platform. Larger angles can only be achieved in certain combinations with the other angle and are neglected for the controller design.

Since basic methods for the parameter determination of controllers are not well applicable for PD-controllers, the parameters are determined by trial and error. Therefore, the parameters are set to $K_R = -0.08$ and $T_V = 0.008/K_R$. The resulting step-response for both directions is shown in the figure 3.5.

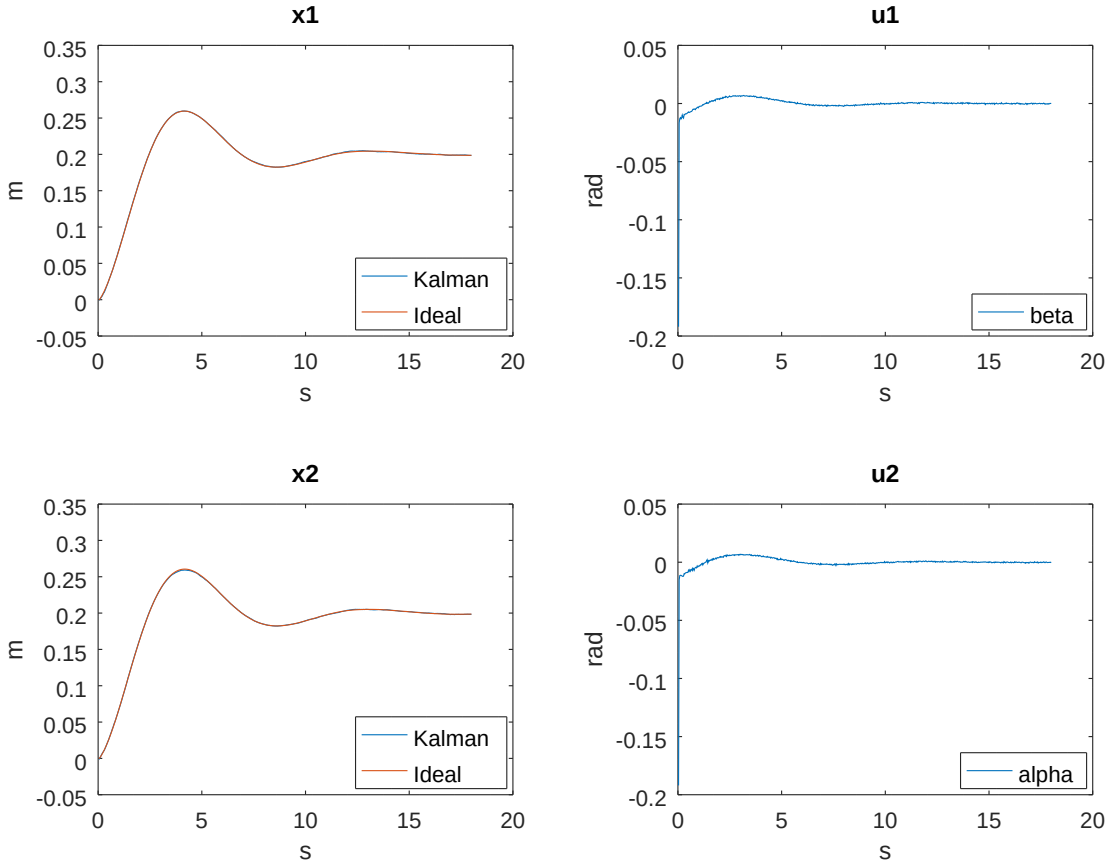

Figure 3.5: Simulated Step Response to (0.2,0.2)

3.2 Hardware and Software Implementation

In the following, the investigations from the previous part are used to create a hardware software co-design. In the first step, the hardware interfaces of the two threads servo and touch are separated from the ReconOS thread. Instead, the interfaces are implemented as AXI-modules. The advantage of this design decision is that the overall architecture gains in flexibility due to memory-mapped access to the interfaces. Therefore, also pure software-based control program implementations are possible. The control and inverse thread contain the decomposition of the controller, Kalman filter and the inverse kinematic of the control loop.

AXI Touch Controller

The touch controller makes the position data of the ball available on the respective demonstrator. The block symbol of the interface is shown in figure 3.6. The interface is divided into the input pins on the left side and the output pins on the right side. The registers are available through the AXI-interface, which also provide the clock for the module. The SPI-interface is realized by the MISO (Master In - Slave Out), SCLK (SPI Clock), MOSI (Master Out - Slave In) and the SS (Slave Select) signal. The interrupt signal (IRQ) triggers the module if a contact on the screen is registered. For debugging and real-time investigation purposes, the read interrupt pin (TC_READ_INT) is true for one clock cycle if the AXI-master reads from the module.

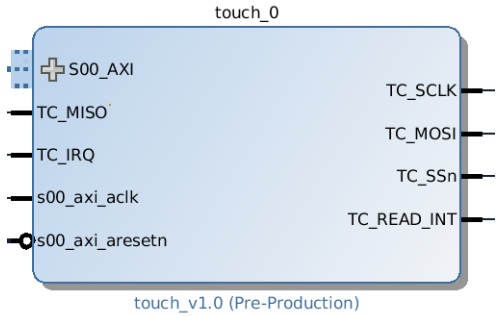

Figure 3.6: AXI-Touch Controller Block

The register map of the module is shown in table 3.1. Each position register contains a cycle counter, which increments after every update of the register. This information allows the control program to check whether the data is new or old.

Table 3.1: AXI Touch Controller Register Description

Byte Offset	Bit							
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
0x00	Cycle Counter					x Position		
0x04	Cycle Counter					y Position		

AXI Servo Controller

Analogous to the AXI-touch controller, the servo controller is also available via the AXI-interface and can be memory-mapped into the physical address space. The module provides six outputs with pulse-width modulation (PWM). The ratio between the duration of power on and power off times is proportional to the requested angle for the motor. The internal logic of each servo motor calculates the angle by this ratio. The AXI-master can set this angle by setting the register of the motor. The value in the register is the actual angle times ten. This results in a resolution of $0.1 \cdot degree/inc$. The register map is shown in table 3.2. Analogous to the AXI-touch controller, the servo controller module provides a signal which is true for one cycle if the sixth register is written by the AXI-master (SERVO_WRITE_INT). The default values of the registers after reset are 900, which is equal to an angle of 90 degree.

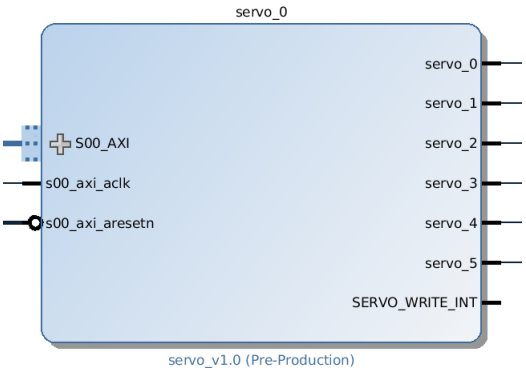

Figure 3.7: AXI-Servo Controller Block

Table 3.2: AXI-Servo Controller Register Description

Byte Offset	Bit							
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
0x00	Servo 0 angle							
0x04	Servo 1 angle							
0x08	Servo 2 angle							
0x0C	Servo 3 angle							
0x10	Servo 4 angle							
0x14	Servo 5 angle							

The address areas of both modules are integrated into the virtual memory area of the ReconOS application via the Linux system function `mmap()`. This allows access by the software threads as well as via the MEMIF-interface by the hardware threads.

However, during the implementation and test of the modules there were problems with regarding cache coherence, since write accesses by hardware threads were only executed sporadically. Due to that, caching is disabled on the ACP-port of the Zynq platform. The configuration is done with the `AWCACHE[3:0]` and `ARCACHE[3:0]` signals of the port. Both vectors are set to zero which solved the problem.

3.2.1 Control Program

The control program of the three Stewart-platforms consists of four threads per demonstrator: touch, control, inverse and servo thread. The touch thread reads the input data from the AXI-touch controller, scales the data and puts it into the mailbox for the control thread.

The mapping of the control loop to the control thread and inverse thread is shown in figure 3.8. The control thread uses the measured position of the ball and updates the Kalman filter with this information. This includes also the preceding prediction of the position. The difference of the reference position $\vec{\boldsymbol{w}}$ and the Kalman estimated position $\hat{\vec{\boldsymbol{x}}} = (\hat{x}, \hat{y})$ of the ball $\hat{\vec{\boldsymbol{e}}}$ is then used for the PD-controller, which calculates the desired angle pair $\vec{\boldsymbol{\alpha}} = (\alpha, \beta)$ for the actual control loop cycle.

The inverse thread uses the output value of the control thread for the calculation of the desired angle γ_i for every servo motor. This is done by the inverse kinematic algorithm, which was overtaken by the existing demonstrator in parts. The servo thread takes this information, scales it and sets the desired pulse-pause ratio at the AXI-servo controller through the AXI-interface.

Since this control loop is implemented as time-triggered system, the system requires a periodic time base for the start of a new cycle. The reason for choosing the time-triggered program paradigm for implementing is to enable equidistant sampling times for the controllers. Therefore, a cycle timer thread is implemented. This thread broadcasts a conditional variable, which starts the calculation of the control program.

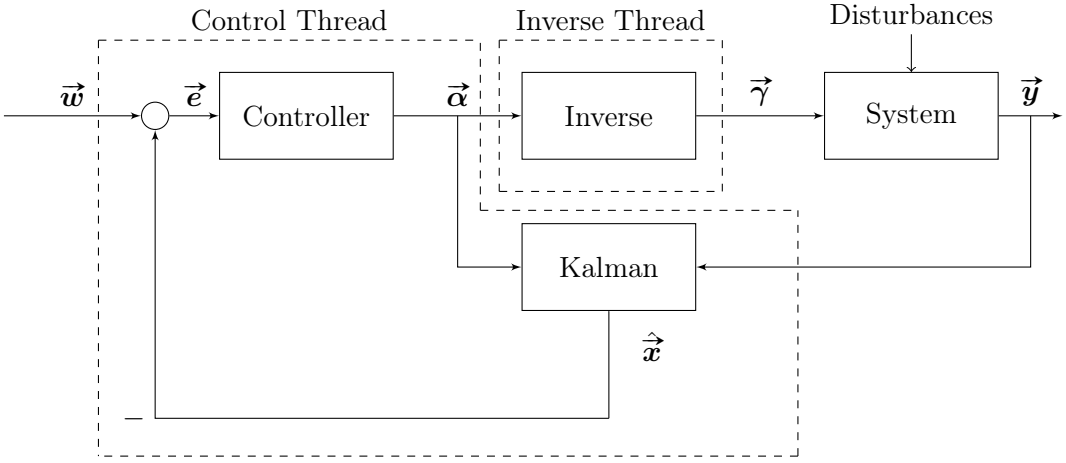

Figure 3.8: Mapping of Control-Loop Functions to Threads

The structure of the control loop is the same for all three platforms. For this reason, the implementation of the functionality is instantiated three times. Each instance must be provided with the information about the demonstrator concerned as well as the memory areas for the input and output functions. For this a structure is created, whose address is made available to the threads as ReconOS initialization data. The structure is shown in listing 3.1.

Listing 3.1: Ball-on-Plate Initialization Data Info Structure

```
struct recobop_info {
        // Offset +00
        volatile uint32_t* pTouch;
        // Offset +04
        volatile uint32_t* pServo;
        // Offset +08
        volatile uint32_t  demo_nr;
        // Offset +12
        volatile uint32_t* timerregister;
        // Offset +16
        volatile uint32_t* stackaddr_control;
        // Offset +20
        volatile uint32_t  rc_flag_control;
        // Offset +24
        volatile uint32_t  threadid_control;
        // Offset +28
        volatile uint32_t* stackaddr_inverse;
        // Offset +32
        volatile uint32_t  rc_flag_inverse;
        // Offset +36
        volatile uint32_t  threadid_inverse;
        // Offset +40
        volatile uint32_t* stackaddr_touch;
        // Offset +44
        volatile uint32_t  rc_flag_touch;
        // Offset +48
        volatile uint32_t  threadid_touch;
        // Offset +52
        volatile uint32_t* stackaddr_servo;
        // Offset +56
        volatile uint32_t  rc_flag_servo;
        // Offset +60
        volatile uint32_t  threadid_servo;

        volatile int thread_count;
        volatile struct reconos_thread *thread_p[4];
};
```

The first both elements of the structure contain pointers to the memory mapped AXI-modules for the regarding servo and touch controller. The demonstrator number is used for the hardware threads to get the corresponding mailboxes for data receiving and transmission, since the mailbox identifier of the regarding thread can be calculate by the mailbox of demonstrator 0 plus the demonstrator number.

The timer register entry points to the global timer of the ARM Cortex-A9 processor and is used for running time measurement purposes. The next 12 entries are used for dynamic reconfiguration, which is described later in this thesis. In general, the `rc_flag_*` variables signal the corresponding hardware thread, that the scheduler wants to reconfigure the running slot. The `stackaddr_*` variable contains the address for the stack of the hardware thread, which can be used to save the state of the thread after a request for reconfiguration. In the last both entries, the pointer to the ReconOS threads of the cor-

responding platform are saved. The `threadid_*` entry assigns a unique id to the current instance of the thread.

At the start time of the hardware ReconOS thread, it requests the initialization data from the delegated thread first. After receiving the initialization data, it uses the memory interface for accessing the needed information. This flexibility will be later used for faster context switching compared to partially dynamic reconfiguration.

The calculations of the servo and the touch thread could certainly have been integrated into the control or inverse thread. However, a larger amount of threads should be deliberately developed for later investigations.

Cycle Timer Thread

The cycle timer thread provides the time base for the control program. In principle, this timer can either be instantiated for individual platforms or provide a common time base for all platforms. The essential function of the timer is shown in the listing 3.2.

The function `nanosleep(const struct timespec *req, struct timespec *rem)` blocks for the requested time defined in `*req` and returns 0 afterwards. The advantage of this function compared to the usual `usleep()` function is the use of the high-resolution timer for time measurement. This results in a lower deviation and higher accuracy of the period time.

On the other hand, the software implementation of the cycle timer in contrast to the hardware implementation with a trigger signal offers the advantage that this timer can also be used as a time base for pure software implementations. The additional time required in the function by handling mutual exclusion and signaling the condition variable can be neglected for cycle times of several tens of milliseconds.

Listing 3.2: Cycle Timer Thread

```
void * cycle_timer_thread(void* arg)
{
    t_cycle_timer * cycle_timer;
    cycle_timer = (t_cycle_timer*)arg;

    struct timespec tim;
    tim.tv_sec = (cycle_timer->period * 1000000) / 1000000000;
    tim.tv_nsec = (cycle_timer->period * 1000000) % 1000000000;

    while(1)
    {
        pthread_mutex_lock(cycle_timer->mutex);
        pthread_cond_broadcast(cycle_timer->cond);
        pthread_mutex_unlock(cycle_timer->mutex);
        nanosleep(&tim , NULL);
    }

}
```

All software threads that want to use the respective instance of the cycle timer as time base can call the function `cycle_timer_wait(t_cycle_timer * cycle_timer)` at the beginning of processing (shown in listing 3.3). ReconOS hardware threads can use the functions for conditional wait and mutual exclusion, which are provided by the ReconOS framework.

Listing 3.3: Cycle Timer Wait Function Implementation

```
void cycle_timer_wait(t_cycle_timer * cycle_timer)
{
        pthread_mutex_lock(cycle_timer->mutex);
        pthread_cond_wait(cycle_timer->cond, cycle_timer->mutex);
        pthread_mutex_unlock(cycle_timer->mutex);
}
```

Touch Thread

In general, the touch thread must fulfill two different tasks. First, the thread has to start a new control loop cycle for the overall platform. For this, it blocks for the condition variable of the cycle timer. When a new cycle begins, the cycle timer releases the waiting threads and blocks for the rest of the cycle. The touch thread unblocks and starts a new memory access to the regarding AXI-touch module. For more advanced systems, it would also possible to wait multiple cycle times by counting. This is one possibility to enable different cycle times for the three platforms with one cycle timer.

On the other hand, the touch thread has to read the input data from the touch screen AXI-interface controller. After reading, the data is scaled and put into the mailbox, which is read by the control thread. The scaling of the position is done by the equation 3.30.

$$\vec{y}' = 0.1166 \frac{[\text{mm}]}{[\text{Inc}]} \cdot \vec{y} \tag{3.30}$$

Control Thread

The control thread gets the scaled position data from the touch thread through mailbox communication. The thread blocks for the corresponding mailbox and unblocks after the put command by the touch thread. The position information and the last output value of the control thread are used to calculate the sequential Kalman filter, which provides a state estimation for the ball on the plate.

The position is used to calculate the PD-controller for both directions, x and y. The results of the controllers are the angles α and β, which are put into the mailbox six times concatenated with their corresponding servo identifier 1...6. Due to that, six independent jobs are created for the inverse thread.

Inverse Thread

The inverse thread is responsible for the calculation of the inverse kinematic and the final calculation of the required motor angles. For this, the positions of mounting of the motor legs is transformed into the base coordinate systems. These positions are permanently stored in the source code and are selected based on the received identifier.

Since the platform is rotated by the angles (α, β), the new resulting position of the mounting is given by equation 3.31.

$$\vec{\boldsymbol{x}}'_m = \boldsymbol{R}_{x,y}(\alpha, \beta) \cdot \vec{\boldsymbol{x}}_m \tag{3.31}$$

The following operations are taken from the previous work [34] and are described here for the sake of completeness only. The mounting position of the rod is then transformed into a coordinate system based on the regarding servo motor shaft. The distance between the coordinate origin and the calculated position is the geometrical sum of the motor leg and the rod, which is mounted on the platform compared to the implementation with floating-point datatypes.

The last step of the calculation is the numerical solution of the motor angle depending on the vector distance to the origin, since an analytical solution would require compute intensive operations like arctan. The resulting angle is concatenated with the corresponding servo identifier and queued into the mailbox for the servo thread.

For the software thread, the functionality is implemented using floating point operations. In the hardware thread implementation, fixed point data types are used for a more resource efficient implementation.

Servo Thread

The servo thread is the counterpart to the touch thread. It gets the angle of the inverse thread and writes the scaled angle into the corresponding register in the AXI-servo module. The address of the register is calculated by the base-address provided by the initialization data and the leg identifier. This operation is done six times per control loop cycle. The scaling is done according to equation 3.32.

$$\vec{\gamma}' = 10 \cdot \vec{\gamma} \tag{3.32}$$

3.2.2 Video Processing

Additionally to described demonstrator and control loop extensions, a video processing chain is added to the design. Thus, the setup can read video input data from the HDMI input interface and write output video data to the HDMI output interface. The video processing hardware architecture is shown in the block diagram in figure 3.9.

The HDMI input interface is connected to the HDMI Receiver IC, which converts the serial HDMI data into a 16-bit parallel pixel signal. The pixel bus is coded in the YCrCb 4:2:2 format. The HDMI Receiver IP-Core converts the data in the ARGB-format with

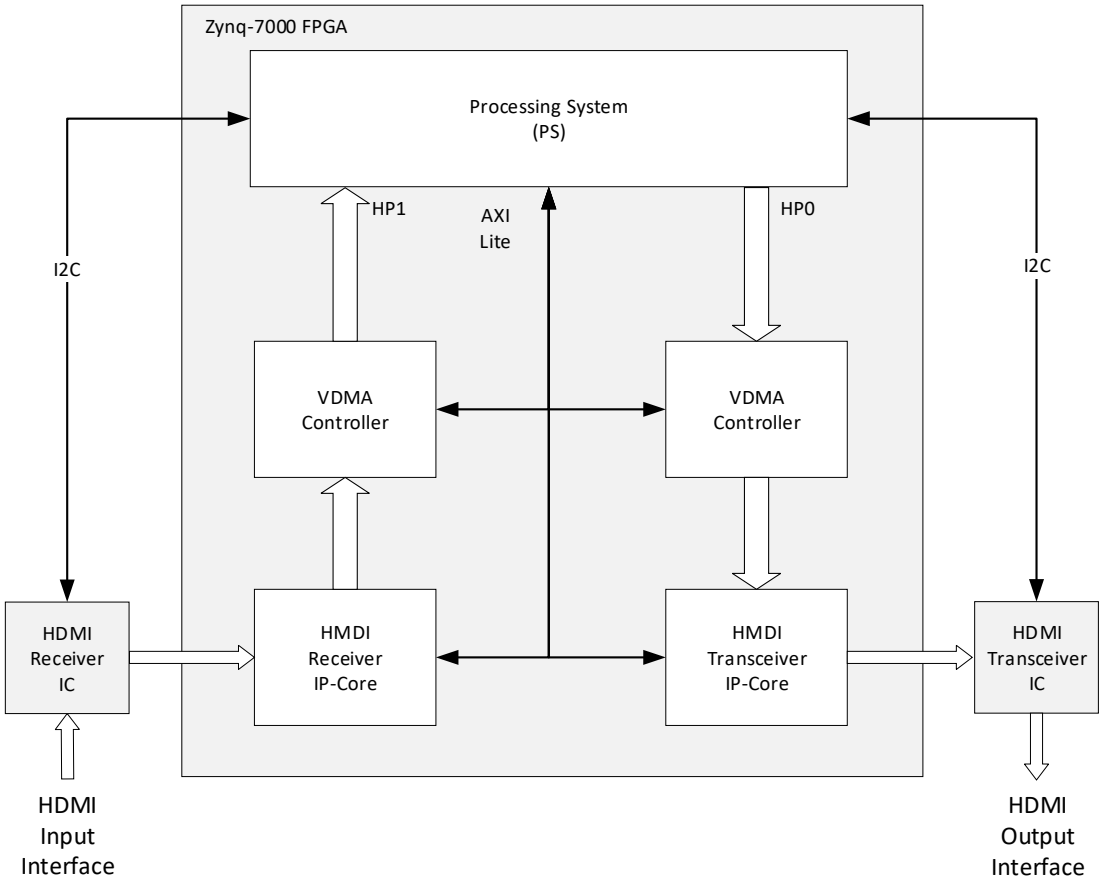

Figure 3.9: Video Processing Architecture

8 bits per channel. From the output of this IP-core, the data is moved to the processing system by a video direct memory access controller block (VDMA).

Analogously to the input side, the content of the frame buffer into the processing system is moved by a second VDMA controller to the HDMI transmitter IP-core, which sends the data through a parallel interface to the HDMI transmitter integrated circuit. Both external integrated circuits get their configuration through an I2C-interface, which is controlled by the kernel driver in the processing system.

For the connection of the VDMA controllers with the processing system, the AXI High Performance Ports (HPx) of the Zynq are selected. These ports allow transfer rates up to 1200 MBit/s into the main memory [22].

For a standardized access, both interfaces are controlled by the Linux kernel with the help of device drivers provided by the chip vendor. The video driver architecture is shown in figure 3.10. The standardized access via the framebuffer device `\dev\fb0` for the output device and the Video for Linux (V4L) `\dev\video0` interface.

For the support by the Linux kernel, the devices are included into the Linux device tree, which provides the hardware description for the kernel at startup. This information is

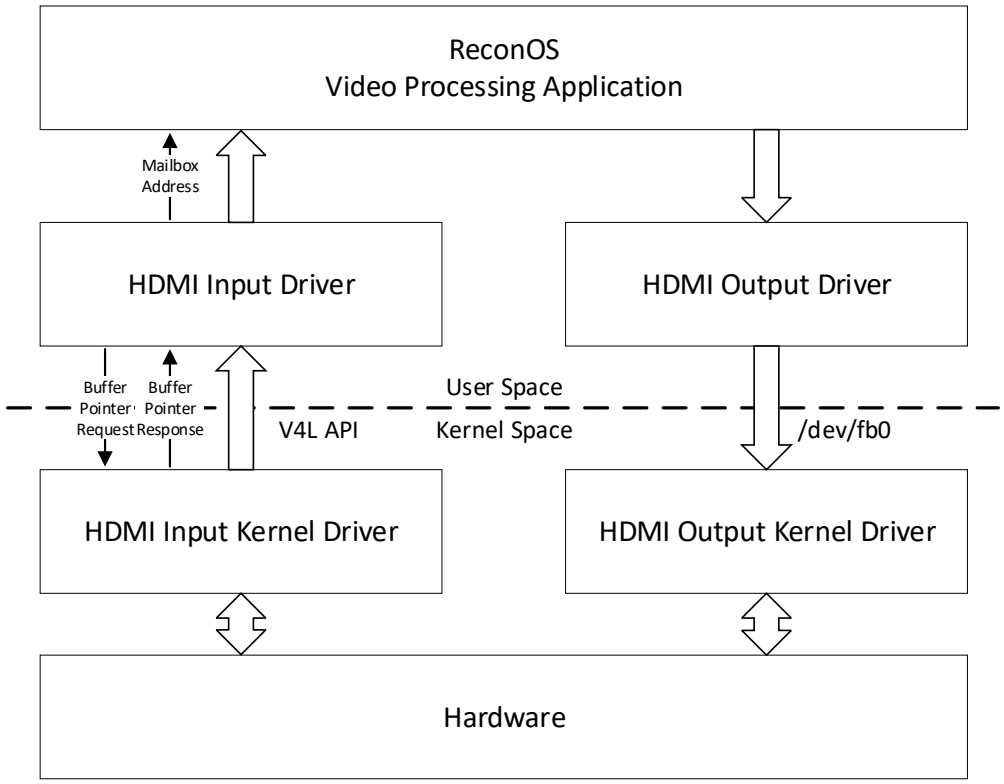

Figure 3.10: Video Processing Driver Architecture

used to load the needed drivers and provides them memory addresses and input and output assignments.

On the application side, there are two drivers in the user space. The HDMI input driver configures the video input device for the access from the ReconOS application. For allowing access from both software and hardware threads, a separate thread requests frames from the HDMI input kernel driver. This thread is shown in listing 3.4.

Listing 3.4: HMDI Input Buffer Thread

```
void * hmdi_input_buffer_thread(void* arg)
{
        t_hdmi_input * hdmi_input = (t_hdmi_input*)arg;
        uint32_t * buffer;

        while(1)
        {
                buffer = hdmi_input_request_new_buffer();
                MBOX_PUT(hdmi_input->mb,(uint32_t)buffer);

    }

}
```

The thread requests a new buffer from the driver and puts the start address of this buffer into the mailbox. This mailbox has a length of one, which means that the thread blocks if there is another frame to process already in the mailbox. Due to that, the driver thread adapts the processing rate of the consumer (e.g. Sobel filter thread) for the request of new frames.

The userspace HDMI output driver does not need such a mechanism for processing. However, it only provides an initialization function for the regarding framebuffer, which should be written. Information about both input and output driver a included in the `t_video_info` structure, which is provided for the processing thread through the ReconOS initialization data 3.5. Therefore, the thread is able to get information about the location of the output buffer but also about mailbox for the input pointer and the width and height of the input and output frame buffer. The third member `volatile struct reconos_thread *thread_p` is the pointer to the processing thread itself.

Listing 3.5: HMDI Video Info Structure

```
typedef struct {
        t_hdmi_input    hdmi_input;
        t_hdmi_output   hdmi_output;
        volatile struct reconos_thread *thread_p;
} t_video_info;
```

Both buffers have the same data structure regarding the image. The image is ordered row-wise with 32 bits per pixel. The colors in the double word are ordered (A, R, G, B), where A is the alpha channel byte and the most significant byte.

For video processing, two example functions are implemented in hardware and software. These threads are implemented for the processing based on the video processing chain in the ReconOS application.

RGB2Gray Thread

The RGB2Gray thread calculates the intensity of the image and outputs a gray image. This very trivial image processing method calculates the intensity of the regarding pixel with the average mode, which does not consider the different intensities of the colors. The value of every output color channel is the average of the colors from the input pixel. Since the gray values in the RGB-space are the values on the line defined by equation 3.33, the equation 3.34 determines the regarding gray value for every pixel. λ is the intensity of the gray value in the range of 0..255 for 8 bit channels.

$$Gray(\lambda) = \lambda \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{3.33}$$

$$\boldsymbol{G_i} = \frac{1}{3}(\boldsymbol{I}_1 + \boldsymbol{I}_2 + \boldsymbol{I}_3), i = 1..3 \tag{3.34}$$

In the software implementation, the input image is processed pixel by pixel with a for-loop and the corresponding operation is performed for each pixel. The result of the averaging is written into all three-color channels of the output pixel.

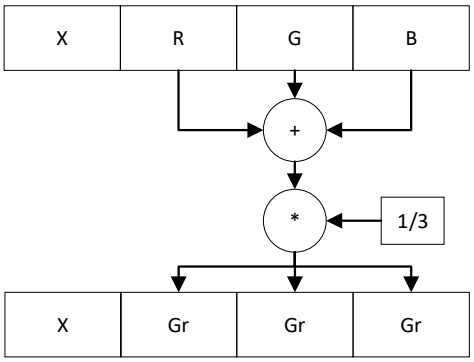

Figure 3.11: Dataflow RGB2Gray Design

For the hardware implementation, the input pixels are read from the main memory in rows. Compared to reading the pixels individually, this method reduces communication overheads while reading. After reading, the data flow graph from figure 3.11 is passed through pixel by pixel. Afterwards, the calculated row of the output image is written back to the corresponding position in the frame buffer.

Sobel Thread

The Sobel operator is used in image processing and realizes a first order derivative of an image for both directions. Due to that, the image filter highlights differences in the input images. Constant areas on the image are removed by the filter.

The filter operation consists of two convolution operations with two rotated filter masks. The result of both operations is geometrical added. The masks are shown in 3.35. The convolution operation of the image with the filter mask $\boldsymbol{S}_x$ highlights the differences in the x-direction, and the convolution operation of the image with the filter mask $\boldsymbol{S}_y$ highlights the differences in the y-direction.

The overall filter operation is show in equation 3.36. Since the input image is an RGB image in the general case, the filter operation has to be done for every color channel independently. The square root operation and the absolute is done elementwise on the resulting matrix.

$$\boldsymbol{S}_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, \boldsymbol{S}_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{3.35}$$

Since square root operations are computational and resource expensive, the approximation of adding absolute values is often used. This approximation is also used in this implementation.

$$\boldsymbol{G}_i = \sqrt{(\boldsymbol{S}_x * \boldsymbol{I}_i)^2 + (\boldsymbol{S}_y * \boldsymbol{I}_i)^2} \approx |(\boldsymbol{S}_x * \boldsymbol{I}_i)| + |(\boldsymbol{S}_y * \boldsymbol{I}_i)|, i = 1..3 \tag{3.36}$$

Like the other threads, the Sobel filter operation is implemented both as a software and as a hardware ReconOS thread. Like the RGB2Gray processing, both implementations get the input buffer address from the HMDI input driver and start the computation afterwards. After that, the processing on the input buffer starts. The dataflow of the Sobel operation is shown in figure 3.12.

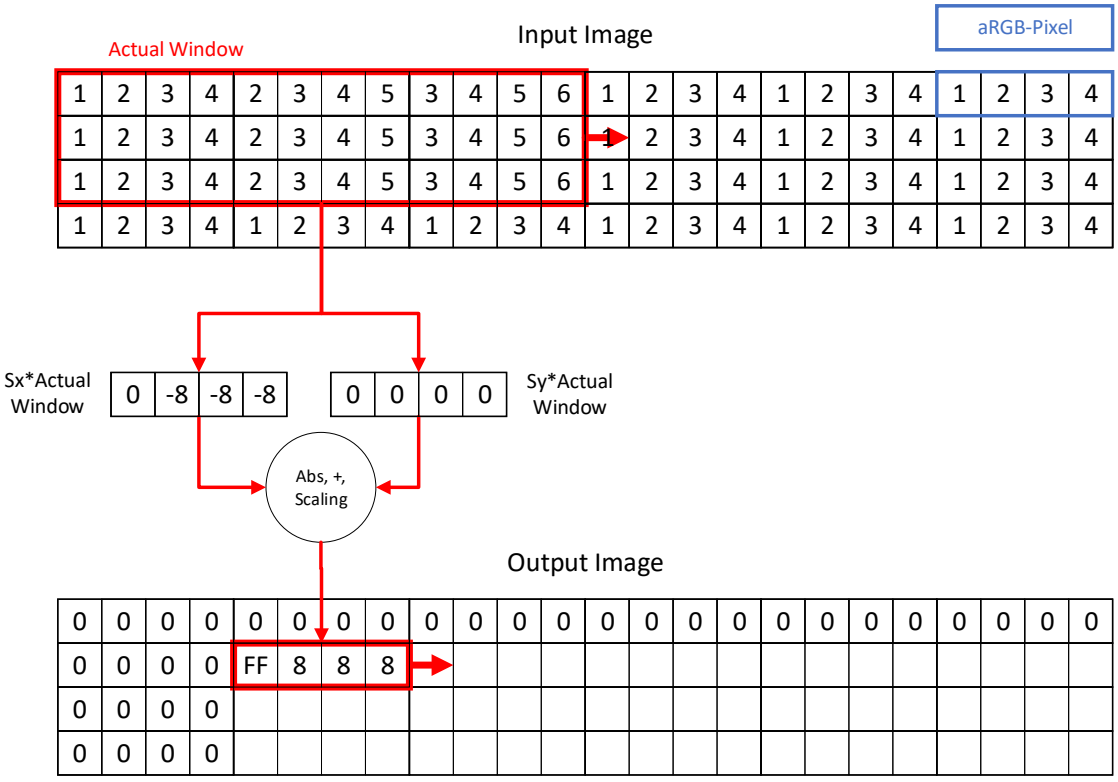

Figure 3.12: Sobel operation for ARGB Images

The processing of a new image starts with the prefetching of the first four rows of the image in the local BRAM. The dimensions of the filter mask of 3×3 leads to a requirement of at least three rows for prefetching. On the other hand, the computation of the actual positions of the pixels in the local BRAM has to do many times. Because of this, the buffer size is extended from three rows to four, because the resulting modulo operations ($address = x \mod 4$) are more efficient in hardware compared to $address = x \mod 3$ operations. A calculation of the address modulo four only requires an exclusive consideration of the lowest two bits of the result.

After that, the window slides over the first three rows. After reaching the end of the row, the next row of the image is loaded from the main memory in the local block RAM and the computation starts again at the beginning of the next row. Before starting, the results of the last row have to be written back to the main memory.

The execution of the current image finishes after processing of the second-last row. The borders of the image are written to zero, since the filter operation of this area is not defined.

On the other hand, the results of the computation may exceed the value range of one byte. To avoid this, the result of the computation is multiplied by the factor of 2^{-3} since the maximum possible value of an operation step is $2^3 \cdot 2^8$. To make the filter also compatible with ARGB color spaces, the fourth byte in the output pixel is set to 255. This sets the alpha channel to not transparent. The operation of the three-color channel is done in parallel, which allows the filter to compute one pixel per loop iteration.

Since the hardware design is done in Vivado HLS, the software implementation is mainly overtaken from the hardware implementation, except for the handling of the initial data.

3.2.3 Remote Reconfiguration Server

Since the demonstrator contains several time-triggered threads, it should also be enabled to execute at least one event-triggered thread. For this purpose, a remote reconfiguration server (RRS) is implemented.

Listing 3.6: Remote Reconfiguration Request Request Loop

```
while(!(reconf_server->shutdown))
{
        request = udp_get_new_request(reconf_server->sockfd);

        switch(request)
        {
                case RECONF_REQUEST_RGB2GRAY:
                        *(reconf_server->rc_flag) = 1UL;
                        reconos_thread_suspend_block(reconf_server->rt);
                        *(reconf_server->rc_flag) = 0UL;
                        reconf_server_reconfigure(reconf_server->
                            bitstreams[0], 0, 1);
                        reconos_thread_resume(reconf_server->rt,((int*)
                            act_hwslot)[0]);
                        break;

                case RECONF_REQUEST_SOBEL:
                        *(reconf_server->rc_flag) = 1UL;
                        reconos_thread_suspend_block(reconf_server->rt);
                        *(reconf_server->rc_flag) = 0UL;
                        reconfigure(reconf_server->bitstreams[1], 0, 1);
                        reconos_thread_resume(reconf_server->rt,((int*)
                            act_hwslot)[0]);
                        break;

                default:
                        printf("[RECONF SERVER] invalid request! \n ");
                        break;
                }
}
```

In general, the remote reconfiguration server binds a socket for datagram-based communication through UDP (User Datagram Protocol) and waits in for a new request. The loop for the request execution is shown in listing 3.6. When a new request arrives the reconfiguration server, the partial reconfiguration of the video thread is started.

Before the reconfiguration, the actual running video thread has to be suspended from the hardware slot. The pending reconfiguration is communicated to the thread via the `rc_flag`, which the actual running thread polls once in each processing cycle. When the thread recognizes the request, it sends a confirmation via the OSIF-interface to the reconfiguration server. After that, the thread blocks and waits for reconfiguration. This ensures that the thread is not in active communication via the OSIF-interface during reconfiguration. This prevents an invalid system state of the delegated thread that would result from the reconfiguration during communication.

After the reconfiguration, the thread is resumed, and the video processing is running again. For faster reconfiguration, both filter kernel bitstreams are cached in the main memory at the beginning. This prevents non-deterministic behavior through file system accesses. The timing of the reconfiguration process is shown in figure 3.13.

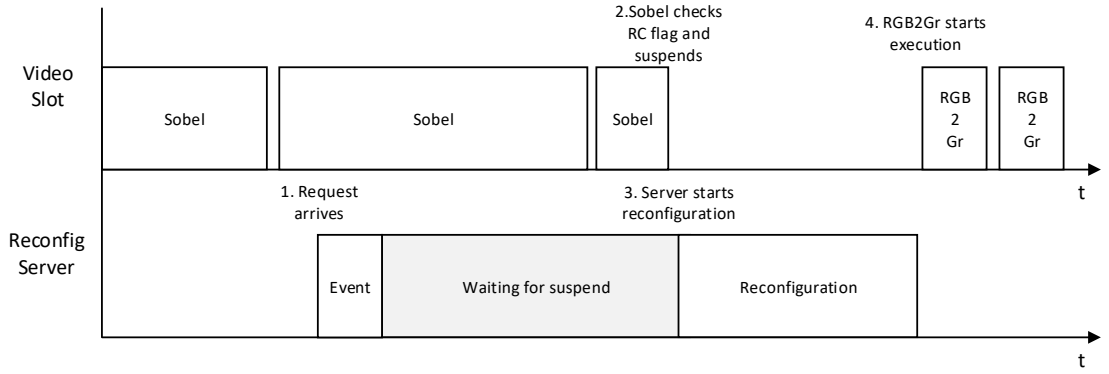

Figure 3.13: Schedule of the Reconfiguration Process for the Video Slot

For the request of a new reconfiguration, a client software is implemented. The command line tool allows the reconfiguration on any computer in the local network by the arguments `./reconfig_client <ip> <port> <request>`. The request definition is set to `rgb2gray` for the RGB2Gray processing unit and `sobel` for the Sobel filter. Since UDP-communication is a connection-less and unconfirmed communication, successful reconfiguration cannot be guaranteed from the client's point of view.

Resulting Scheduling Constraints

Due to the described properties of the set of tasks, there are dependencies for the scheduling algorithm. These dependencies are visualized in the precedence graph in figure 3.14. The scheduling dependencies are results of the considerations during this chapter and must kept in mind for the scheduling of the threads.

The constraints essentially affect the threads from the control loop, since only these have data dependencies to other threads. On the other hand, the two threads for video editing

could also be combined to form longer processing chains, although this is not necessary for correct functioning of the single threads.

The Remote Reconfiguration Server is the only thread, which provides event-triggered behavior. In principle, the thread can occur at any time and then leads to an interruption of the currently running video thread.

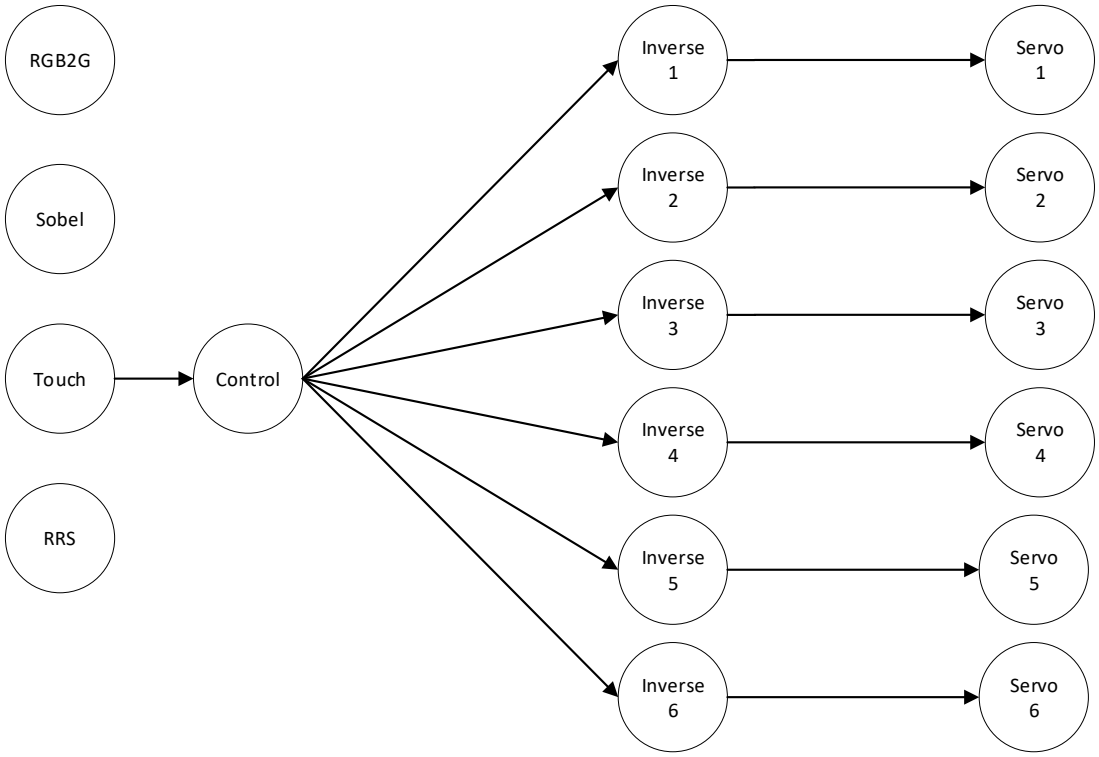

Figure 3.14: Resulting Precedence Graph

3.3 Control Loop Partial Reconfiguration

Partial dynamic reconfiguration allows to write the reconfiguration bitstream for only parts of the whole FPGA during run time. The untouched configuration remains and executes during the reconfiguration process.

Due to this technique, a behavior similar to software multi-threading on a CPU can be achieved. Designated parts of the programmable logic are sequentially configured with different bitstreams during run time.

This behavior is used for the control and the inverse thread during this work, the option to use it for other threads is also provided. Due to the multitasking with reconfiguration, the control thread and the inverse thread for all demonstrators are executed on only one slot per thread-type, so that only two slots are needed instead of a total of six for the two thread types.

In the background chapter 2, different approaches for multitasking on FPGAs are described. For simplicity, the multitasking in this scenario is done in a cooperative way. The architecture of the reconfiguration support is shown in figure 3.15.

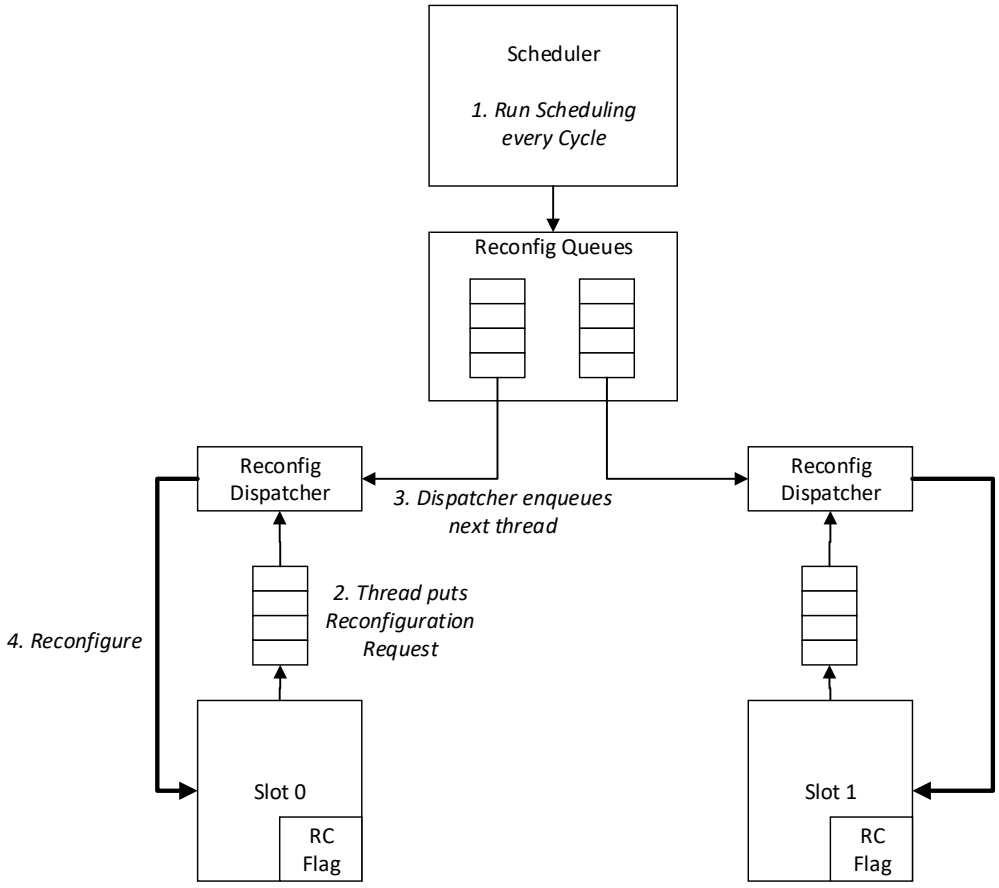

Figure 3.15: Reconfiguration Setup for Control Loop Threads

In every control loop cycle, the scheduler fills the reconfiguration queue with threads, which have to be executed during that cycle. The scheduling strategy is flexible. For the purposes of this thesis, every thread is executed in every control loop cycle. On the other hand, other scheduling strategies with different orders and execution frequencies can be determined by the scheduler. The scheduler uses the existing cycle timer implementation as time basis.

Each slot has access to a Reconfiguration Flag (RC Flag), which is polled after every execution. When this flag is not null, the slot sends a reconfiguration request to the slot specific dispatcher. The reconfiguration request is sent by mailbox communication. After request sending, the thread blocks. The dispatcher enqueues the next waiting thread from the queue and performs a reconfiguration on the slot. In case there is no new thread in the queue, the delegated thread also blocks until the scheduler refills the queue.

Before performing a reconfiguration request, the thread must eventually save its actual context for later calculations. For example the control thread must save the status matrices of the Kalman filter and the PD-controller, since results from the previous cycle are included in the calculation.

During reconfiguration, the dispatcher also has to change the pointer to the initialization data for the thread. Due to the initialization structure, the starting thread has information about the demonstrator and the memory locations of its servo and touch modules. The initialization data also contains information about the location of the context which has to be restored after reconfiguration by the thread itself.

For faster and more deterministic reconfiguration times, the bitstreams for the threads are cached in the local main memory since the file system is often based on a network file system. Therefore, non-deterministic times for the access of the bitstreams are negligible.

3.4 Chapter Conclusion

In this chapter, the extensions on the existing demonstrator are described. The most visible modification on the demonstrator is the addition of two platforms. The demonstrator is extended by an HDMI expansion board with an input interface.

For the control of the platforms and the video processing, different threads are implemented. The available threads can be divided into three classes: the threads from the control loop represent a group of time-triggered threads that have to meet hard real-time requirements. Missing the deadlines for the thread could lead to the loss of the ball on the platform.

On the other hand, the threads for video processing represent a class of periodic time triggered tasks that have to fulfill soft real-time requirements. Missing a deadline only leads to a loss of output quality. The start time or period is determined by the system itself through the execution time of the processing.

The last available class are the non-periodic tasks triggered by the remote reconfiguration server. In general, these tasks can be classified to the group of event-triggered tasks, where the event in this case is the arrival of a new request through UDP.

The partial reconfiguration for control loop threads enables multitasking for the threads control and inverse. Therefore, resource-saving implementations in hardware are possible. Additional to that case of partial reconfiguration, the remote reconfiguration server also uses partial reconfiguration for the change of the filter kernel.

4 ReconOS Real-time Investigations

The implementation of the control software based on ReconOS described in the last chapter contains many components whose real-time properties are not known or not reliable. This includes for example the scheduling of the software threads or the latency of communication via mailboxes. Therefore, the focus in this chapter will be on improving these real-time properties and on measuring and modeling the behavior. First, modifications on ReconOS are introduced, which ensure a more deterministic behavior. For archiving better results, also changes on the Linux kernel are made.

In the next step, the execution times of the individual software and hardware threads are determined. No formal proof is provided for this but rather the execution times are measured several times in a defined test environment. Afterwards, remaining uncertainties are determined and modeled in real-time system behavior. The following part of this chapter deals with the question of communication times between threads and execution times of other system calls from hardware and software threads.

The determination of execution times and communication times, including the effects of parallel use, enables the prediction of execution times for complete schedules. The estimation of execution times for exemplary schedules is carried out in the following part of this chapter. The evaluation of these schedules is done in the following chapter 5.

4.1 Real-time ReconOS based-on Linux

By default, the Linux scheduling algorithm CFS (Completely Fair Scheduler) aims to allocate the available CPU time to the threads depending on the already used CPU time and the past waiting time (aging). This behavior should optimize the overall system throughput and ensure a fair distribution of processor times for all threads. For a time-triggered real-time system or real-time systems in general, this scheduling policy is not sufficient, because of the fact that some threads are more critical for the system functionality than others.

In the case of the demonstrator, the video processing thread is less time critical than the control thread. Assuming a pure software implementation of all threads, the Linux scheduler could preempt the control loop algorithms and run the video processing thread instead. This behavior would lead to disturbances in the control loop and thus to the loss of the ball.

The actual ReconOS framework does not allow priority assignment to pure software threads or the delegated threads of the hardware threads. Therefore, all threads are scheduled with the default `SCHED_OTHER` policy, which is described before. To avoid waiting times for time critical threads, the scheduling policy of ReconOS is extended by

the option for real-time scheduling due to the `SCHED_FIFO`-policy. Due to that, a software thread with a higher priority than the running thread is always allowed to interrupt them and can be only interrupted by threads with a higher priority.

On the other hand, running threads with high priorities have to free the processor co-operative. This increases the deterministic behavior and allows shorter cycle times but brings the risk that threads can starve due to threads with higher priorities. Therefore, an estimate of the execution time of the threads must be available at design time. An alternative to the `SCHED_FIFO` scheduling policy of Linux is the `SCHED_DEADLINE` policy, which is not considered during this thesis.

For both software threads and delegated threads, some extensions in the ReconOS framework have been made. The changes affect the functions in ReconOS in which software threads or delegated threads are started.

- **Lock Memory** The `mlockall`-function is used to lock the virtual address space of the thread in the main memory. Therefore, memory pages are not allowed to be moved to the hard disk due to swapping since page faults would lead to a highly non-deterministic behavior.

- **Scheduling Policy** As already mentioned, the scheduling policy of threads is set to `SCHED_FIFO`, which allows high priority threads always to interrupt threads with lower priorities.

- **Minimum Stack Size** The minimum stack size is set to `PTHREAD_STACK_MIN`. For software threads, this value may need to be increased if the stack size exceeds the specified size.

- **Priority** Regarding the priority of the threads, there are two cases. First, the priority for all delegated threads is set to 80. Delegated threads have usually short execution times but are critical for the calling hardware thread to execute the request with low latency. For all other threads, the priority can be set due to a parameter in the thread creation function. The priority can be chosen in the interval 1..79. For maintaining backward compatibility of non-real-time ReconOS applications, a software thread with priority 0 can be set. Then all extensions will be omitted, and the thread will be executed with the `SCHED_OTHER` scheduling policy.

Additionally to the changes on the ReconOS framework, the *PREEMPT_RT* patch is applied on the kernel sources. The Linux kernel version 4.14 is used for the improvements. As already described in the background chapter, this further reduces the number of uninterruptible parts in the Linux kernel. This results in a lower average latency.

4.2 Execution Time Measurement

For the design of a reliable time-triggered real-time system, the precondition for a feasible scheduling has to be met (equation **??**). Therefore, for ensuring the feasibility of

an existing schedule or for determining the minimal possible period P, the worst-case execution times $WCET$ of the taskset $\tau_k \in T$ have to be determined (equation 4.1).

$$\sum_k WCET(\tau_k) \leq P \tag{4.1}$$

Modern processors have a considerable number of speculative and non-deterministic units at their disposal to increase performance. These units make it difficult or even impossible to determine the maximum execution time analytically. This mainly includes super scalar processors with branch predictions, speculative execution, multiple cache levels and dynamic scheduling.

However, since the used ARM Cortex-A9 processor supports these features except for dynamic scheduling, the analytical determination of the worst-case execution is a challenging task. On the other hand, a worst-case execution time analysis would provide a very pessimistic prediction, since for example performance gains due to speculative processor features would be completely eliminated. Above all, the elimination of caches for the analytical determination of worst-case execution times would lead to an unrealistic estimate of the execution time.

Due to this, the measured execution times MET of the taskset is determined in a test environment as an approximation for the worst-case times of the taskset $WCET(\tau_k) \approx MET(\tau_k)$. For the later design of a schedule based on these execution times, this fact must be considered.

The measurement of the execution times of control loop hardware threads can provide the true worst-case execution time, if there is no communication with other threads or the with the processing system. In this case, the execution time is constant and therefore deterministic. For hardware threads in the video processing chain, this is not the case since these threads accesses memory not only in the beginning and the end of the execution but also in every loop iteration.

The measurement of the execution times only considers the calculation of the output values, not initial phase of the threads or the blocking time on mailbox or other synchronization objectives. The mailbox communication at the beginning and at the end of the processing, which occurs with most tasks, is modeled later on (Listing 4.1).

In addition, when measuring the execution times, it must be ensured that only the execution time of the thread is measured. Overheads through the operating system and other software parts have to be ignored. This problem only occurs with software threads, not with hardware threads since they use their resources exclusively.

Listing 4.1: Typical Program Structure for Control Loop Threads

```
void THREAD()
{
        do_initialization();

        while(1)
        {
                input_data = MBOX_GET(mbox_input);
                //Start measurement here
                output_data = processing(input_data);
                //End measurement here
                MBOX_PUT(mbox_output, output_data);
        }

}
```

Hardware

The standard IP-core library of the Xilinx toolchain contains an AXI-Interface timer module, that provides capture inputs to latch the actual timer value into a register for different trigger conditions. This capture function is implemented two times, but with two different timers. Due to that, two captured values are not directly comparable, since both timers are started asynchronously.

Therefore, a new difference measurement module is implemented for execution time measurements (shown in figure 4.1). This timer provides four capture inputs, which triggers the write operation of the actual timer value into the related register. Both the counter register and the capture registers are accessible via the AXI interface. The register map is shown in table 4.1.

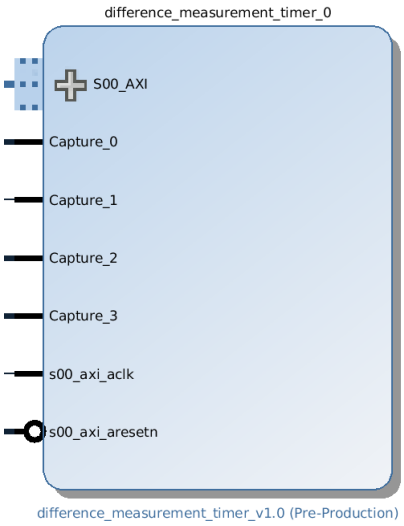

Figure 4.1: AXI Difference Measurement Timer Block

Table 4.1: Difference Measurement Unit Register Description

Byte Offset	Bit							
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
0x00	Don't Care							
0x04	Timer Counter Register							
0x08	Capture 0 Register							
0x0C	Capture 1 Register							
0x10	Capture 2 Register							
0x14	Capture 3 Register							

In order to determine the execution time, the hardware threads set a binary output signal for one clock cycle before and after processing. Both signals are connected to separate capture channels of the difference measurement unit and can be read by the evaluation software. Due to the clock supply with 100 MHz, the resulting theoretical resolution of the capture inputs is 10 ns.

For example, the communication time between hardware and software threads, a software-side capture is necessary for some measurements, which is made possible by the access to the timer counter register. This measurement necessarily has a higher inaccuracy, since the measurement result is falsified, among other things, by access via the AXI interface. Therefore, the theoretical accuracy of 10 ns cannot be achieved with this measurement method.

Software

For the measurement of the execution time of the software threads, the processing part of the regarding thread is isolated into the function `DUT(uint32_t * in, uint32_t * out)` of listing 4.2. The selection of the input data is important for a correct measurement, since the number of loop iterations or branches depend on the input data. For the measurement, input data which leads to maximal execution time should be used.

Conveniently, the execution time of none of the threads to be examined depends on the input data. Therefore, random numbers can be used for the investigation of the execution times.

For determining the current time, the Linux function `clock()` is used, which returns the required processor time used by the program. The difference between two measured values before and after the function then represents the computing time used.

Another aspect of the measurement is the execution time dependency on the cache hit rate, especially on the first iteration of the measurement loop. Due to that, the measurement is done several times, e.g. 10 times like in the listing. The measured time of the last iteration is used as the reference value. All measurement setups are compiled without compiler optimizations like the target ReconOS application.

Listing 4.2: Software Environment for Execution Time Measurement

```
#include <stdio.h>
#define N 10

int main()
{
    double dStartTime;
    double dEndTime;
    uint32_t input data[100];
    uint32_t output_data[100];

    srand(time(0));

        for(int j = 0; j < 100; j++)
        {
                input_data[j] = rand();
    }
    for (int i = 0; i < N; i++)
    {
        dStartTime = clock();
        DUT(in, out);
        dEndTime = clock();
        printf("Time: %d \n", dEndTime - dStartTime);
    }
    return 0;
}
```

For other software measurement purposes which require absolute time values, the global timer of the ARM Cortex-A9 processor is used. Therefore, the address-space of the timer is mapped into the virtual address-space of the application with `mmap()`. The global timer is a 64-bit counter which is available in the private address space for each processor core. The frequency of the timer is 333.33 MHz, that results in a theoretical resolution of 3 ns. The main advantage of the usage of this timer is the fast access at capturing, which only requires one memory access to the counter register of the timer.

Results

The results of the measurements in the different test environments are shown in table 4.2. Both the results for hardware and software threads are inserted.

As expected, the touch and the servo thread have minimum execution times, which are only a few cycles in software and hardware. The thread inverse needs the longest execution time of all control-loop threads. This is mainly due to the numerical solution of the servo angle.

The Sobel filter benefits from the parallel processing by loop unrolling and can reach a speedup of almost 7.5 compared to the non-optimized software implementation. Compared to the optimized implementation, the filter benefits from the higher clock of the processor. Since the RGB2Gray filter is not less compute intensive, the speedup compared to the non-optimized implementation is lower here.

Table 4.2: Measured Execution Times

	HW Implementation	SW Implementation	
	Default	Default	Optimized
Servo	$<$ 0.001 ms	0.001 ms	0.001 ms
Control	0.030 ms	0.040 ms	0.017 ms
Inverse	0.196 ms	1.780 ms	1.430 ms
Touch	$<$ 0.001 ms	0.001 ms	0.001 ms
Sobel	127.390 ms	965.465 ms	53.0 ms
RGB2G	30.010 ms	58.810 ms	8.4 ms

For the software implementation, the program is also compiled with the optimization level three and loop unrolling (`-O3 -funroll-all-loops`). In this case, the execution times for the video filter software implementation is comparable or better than the hardware implementation. Especially the RGB2GR-filter benefits from the faster memory connection compared to the hardware implementation.

For later investigations, not only the execution times of the threads are required, but also the duration of individual reconfiguration of slots. These are the slots for the control loop (slot 3 and slot 4) and the slot 12 which is controlled by the remote reconfiguration server. The results of the measurements are shown in the table 4.3. The values for the reconfiguration also depend on the size of the block to be reconfigured, which is why these values, like the other measured time values, are only valid for the current implementation.

Table 4.3: Duration of the Reconfiguration

Slot	**Time**
Slot 3	37.5 ms
Slot 4	39.6 ms
Slot 12	37.9 ms

An execution time for the remote reconfiguration server independent from the time for the reconfiguration cannot be determined, because the time strongly depends on the current execution status of the video thread.

4.3 Communication and System Call Modeling

As mentioned in the last sub chapter, the mailbox communication between threads is not considered in the execution time measurement, since the communication between threads is expected as relevant more non-deterministic than the calculation parts. This fact motivates the measurement and investigation of the timing behavior of such thread activities.

Due to the fact that the needed time for the communication depends on various factors within the whole system, the result is modeled as an unknown distributed random variable. The related distributions are shown in histograms later. For real-time systems, the most relevant result of such measurements would be the worst-case communication

time $WCCT$. The determination of this value without any uncertainty would require an infinite number of measurements, which is not feasible.

For a weaker modeling of the timing behavior, the measurements are described by the mean value of the time $\bar{t}_C$ and the standard deviation s, which are shown in equation 4.2 and 4.3. For the estimation of the maximum latencies the maximum communication time of the measurement is given by the value t_{Cmax} (equation 4.4).

$$\bar{t}_C = \frac{1}{N} \sum_{i=1}^{N} t_i \tag{4.2}$$

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (t_i - \bar{t}_C)^2} \tag{4.3}$$

$$t_{Cmax} = \max(t_i) \tag{4.4}$$

For the determination of communication times, five different measurement setups are created. The setups differ in the type of threads and the type of communication. The structure of this setups is described in the following. For a better comparison, the results are summarized later in one plot.

Software-to-Software Communication

In the first measurement, the communication time between two software threads is investigated. Therefore, the first thread reads the actual timer value from the Cortex-A9 global timer and starts a new mailbox put operation. During that, the second thread blocks on a mailbox get operation and waits for new data. When the operation unblocks and the data is read, the second thread accesses the same timer register from the Cortex-A9 timer and compares the value with the value from the second thread (Figure 4.2). The difference between them is the needed time for the communication process. The measurement is done about 50000 times.

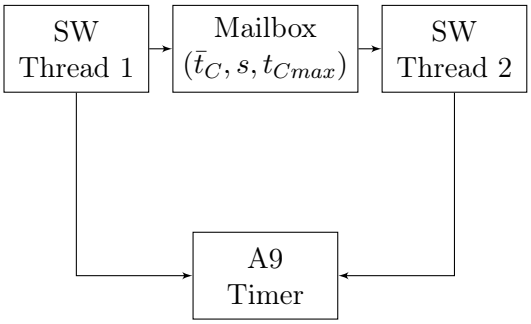

Figure 4.2: ReconOS Mailbox Model SW-Thread to SW-Thread

Hardware-to-Hardware Communication

The Difference Measurement Timer is used to measure the communication time between two ReconOS hardware threads. For this purpose, the presented difference timer measurement unit is used. The measurement is done in the same way as for the software to software time, except for the timer. The first hardware thread sets the trigger signal directly before it starts a mailbox put command through the OSIF-interface. After receiving the data by thread 2, its sets another signal. The difference between both capture inputs is the time for the communication. The setup is shown in figure 4.3.

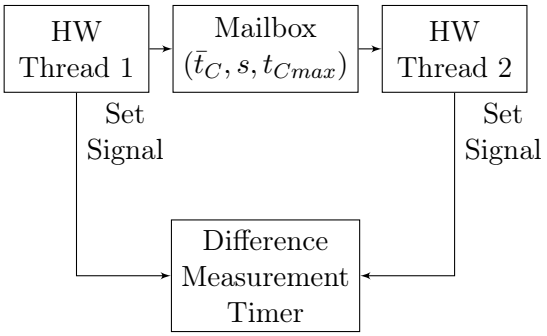

Figure 4.3: ReconOS Mailbox Model HW-Thread to HW-Thread

Hardware-to-Software and Software-to-Hardware Communication

For the measurement of the communication time between hardware and software threads, the assumption that the access time over AXI to the difference measurement unit is negligible compared to the overall communication time has to be made. Therefore, the duration can be measured due to the comparison of the capture input and a software capture of the timer register, which is accessible by the software thread.

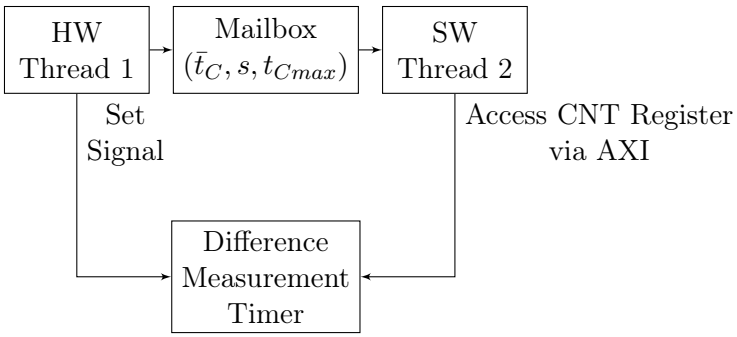

Figure 4.4: ReconOS Mailbox Model HW-Thread to SW-Thread

The measurement of the communication time in the other direction is done in the same way. The regarding setups are shown in figure 4.4 and figure 4.5.

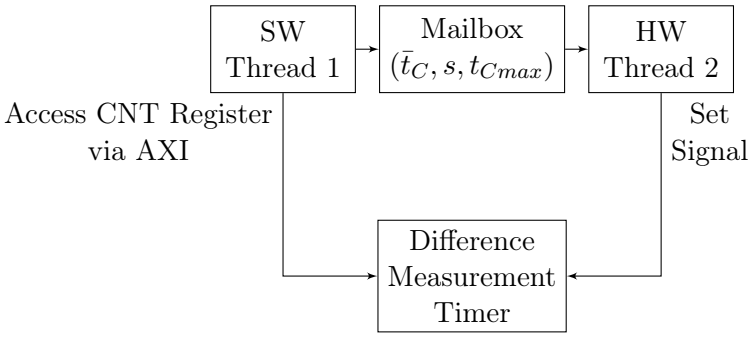

Figure 4.5: ReconOS Mailbox Model SW-Thread to HW-Thread

System Call Communication

This measurement setup is representative for all requests of a hardware thread that only involves the thread that submits the request. This test-case automatically covers measurements like semaphores, mutexes, mailboxes etc. where the resource is not used, and the delegated thread is not blocked.

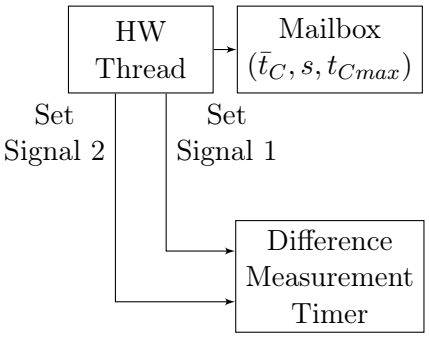

Figure 4.6: Setup for System Call Measurement

The intended setup is shown in the figure 4.6. The requesting hardware thread sets the first signal and starts the mailbox request by the OSIF interface. After the request is done and the response from the delegated thread has arrived, the hardware thread sets the second signal. The time difference between the two signals is calculated later in the software running on the processing system.

Results

The results of all the four measurement setups are shown in figure 4.7. Each measurement is repeated 50000 times and the results are shown in the histogram. The values in the legend of the figure are calculated according to equation 4.2, 4.3 and 4.4.

The results of the measurement show that a mailbox put operation has an average duration of 31.62 us. For this operation, the delegated thread is unblocked by the resulting kernel interrupt and the message is inserted into the mailbox object by the delegated

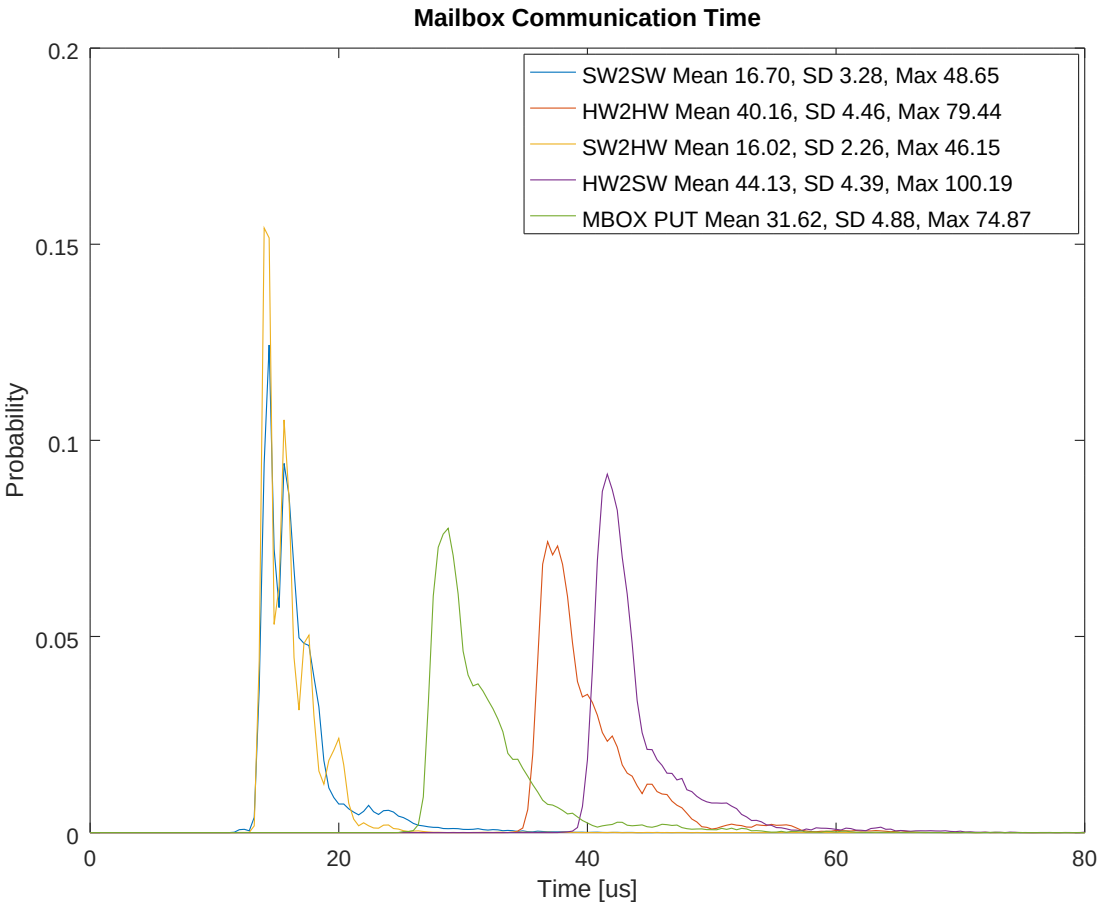

Figure 4.7: ReconOS Mailbox Communication Times

thread. This process cannot be interrupted by another thread, since the delegated thread is executed with a maximum priority of 80.

A mailbox communication between two software threads takes on average 16.7μs. The combination of the two mentioned communications corresponds temporally but also functionally to hardware to software communication. In this case, the delegated thread unblocks the corresponding software thread. The addition of both mean values corresponds approximately to the time required for hardware to software communication.

The measurements indicate that communication from software to hardware generates low latencies. For example, if the communication from a software thread to a hardware thread is considered, this corresponds roughly to the latency of software to software communication. A similar statement can be made about the difference between the mailbox put operation and the hardware to hardware communication. The difference of the mean values corresponds to the processing time of the second delegated thread.

4.4 Resource Sharing

In a ReconOS-based system, there are parts which are shared at least between the hardware threads. Due to the limited number interfaces to the processing system and the general requirement of serialization of memory accesses and interrupt triggers, the number and behavior of hardware threads has an influence on the whole ReconOS-system and therefore for the run times of other thread in the system. Specially in for real-time systems, these influences have to be considered at design time. This section deals with these influences under different experiments and extends the results of the last section.

OSIF Communication

In the first experimental setup, the behavior of parallel mailbox access is investigated. The system is shown in figure 4.8. Due to the setup, up to 8 hardware threads can be started for a mailbox at the same time. For archiving exact same starting times, the threads are connected to an AXI-GPIO IP-core, which output signal can be set by the processing system. This will activate all hardware threads at the same time.

A conditional variable provided by the operating system is used for the triggering of different numbers of software threads. Due to the limited number of processor cores, a serialization of the processes is inevitable.

In the first test scenario, a mailbox put on an empty mailbox of up to 8 threads is executed simultaneously. This scenario represents all system calls that are executed by a hardware thread. In this case, an interrupt is triggered by the OSIF controller and the kernel unblocks the corresponding delegated thread. The delegated thread executes this call (in this case the Mailbox Put operation) and sends the return value back to the hardware thread via the OSIF-interface.

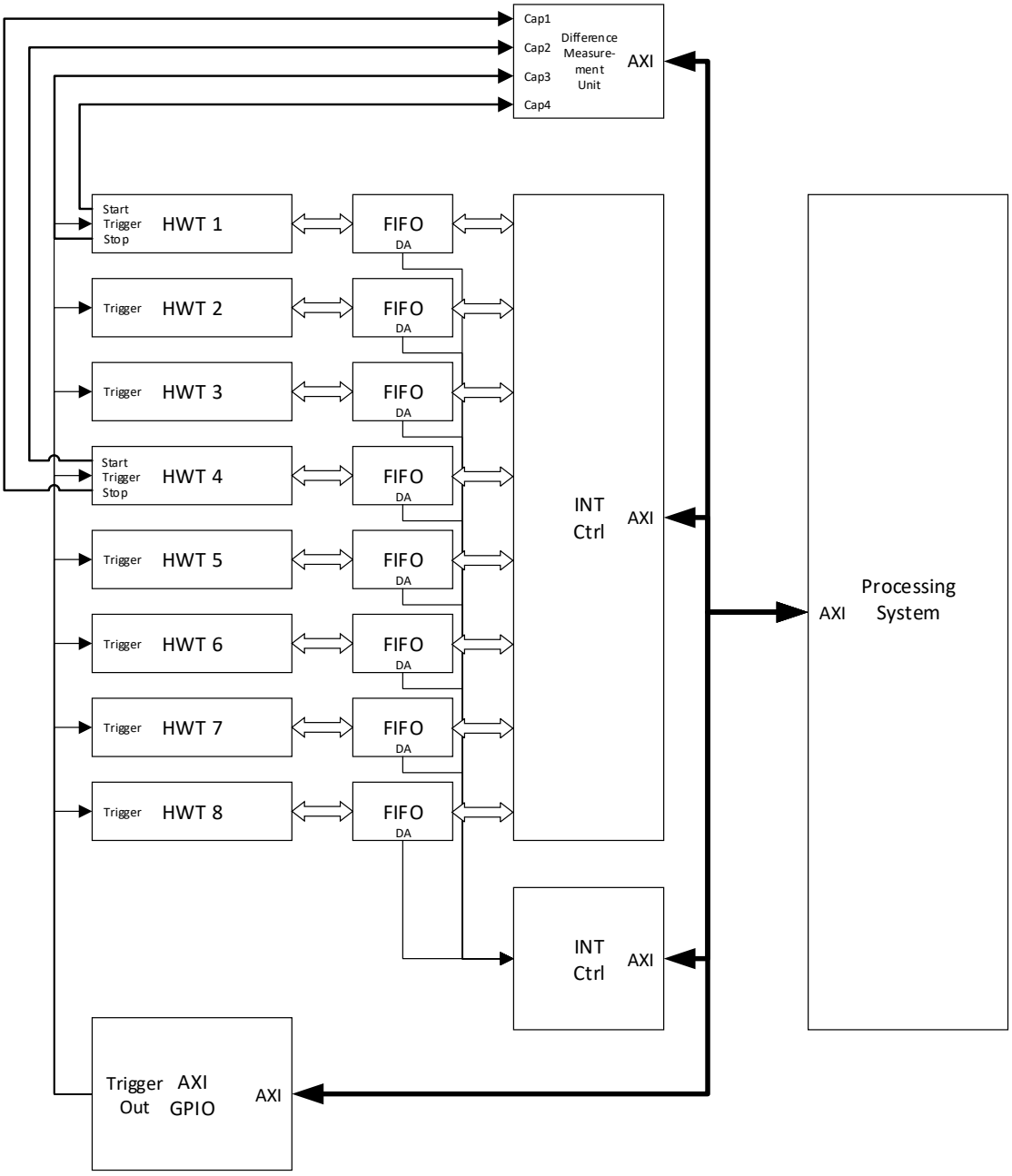

Figure 4.8: Parallel Mailbox Access Setup

In the first test scenario, a mailbox put on an empty mailbox of up to 8 threads is executed simultaneously. This scenario represents all system calls that are executed by a hardware thread. In this case, an interrupt is triggered by the OSIF controller and the kernel unblocks the corresponding delegated thread. The delegated thread executes this call (in this case the Mailbox Put operation) and sends the return value back to the hardware thread via the OSIF-interface.

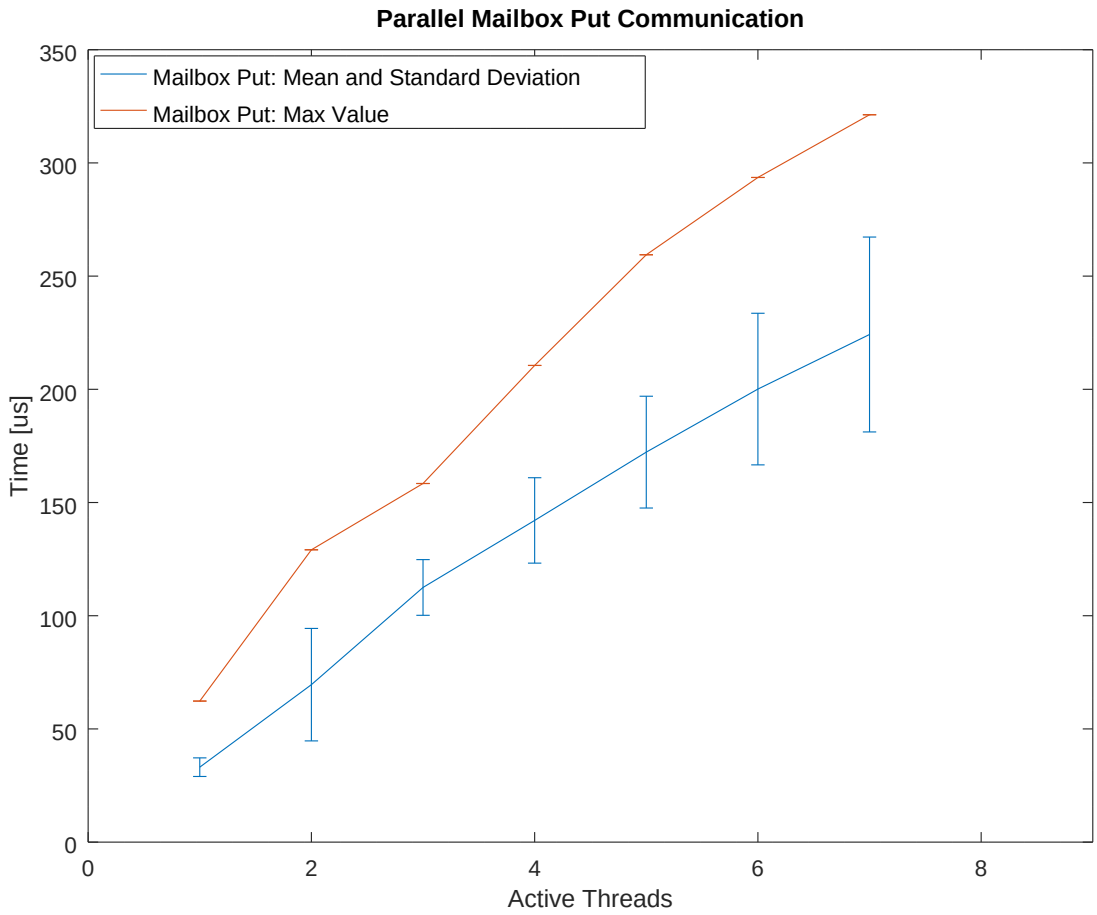

Figure 4.9: Parallel Mailbox Put operated by up to 8 Threads

The results for the latency measurement are shown in figure 4.10. The plot shows an approximately linear increase of the mean latency with the number of threads involved. This also applies to the respective measured maximum value.

For the following test scenarios, a communication between two threads is considered. For this, the point in time before the communication and the point in time after the communication is measured once in each case. The difference of the measurements and thus the elapsed time is shown in figure 4.10 for the different test scenarios.

The plot above shows that the communication time between two threads for hardware to hardware communication increases circa linearly. This behavior also applies approximately to the communication between hardware to software.

This fact does not apply to the communication from a software thread to a hardware thread. The time between the start of the communication and the arrival of the message at the counterpart remains approximately constant. This is due to the serialization by

the limited number of processor cores, which becomes visible through the lower plot of figure 4.10. The partial illustration shows the total duration of all threads of the entire communication for the software to hardware communication. As expected, this time increases almost linearly with the number of participants. The small increase from one thread to two threads can be explained by the existing two processor cores of the Zynq, which allow a true parallel execution.

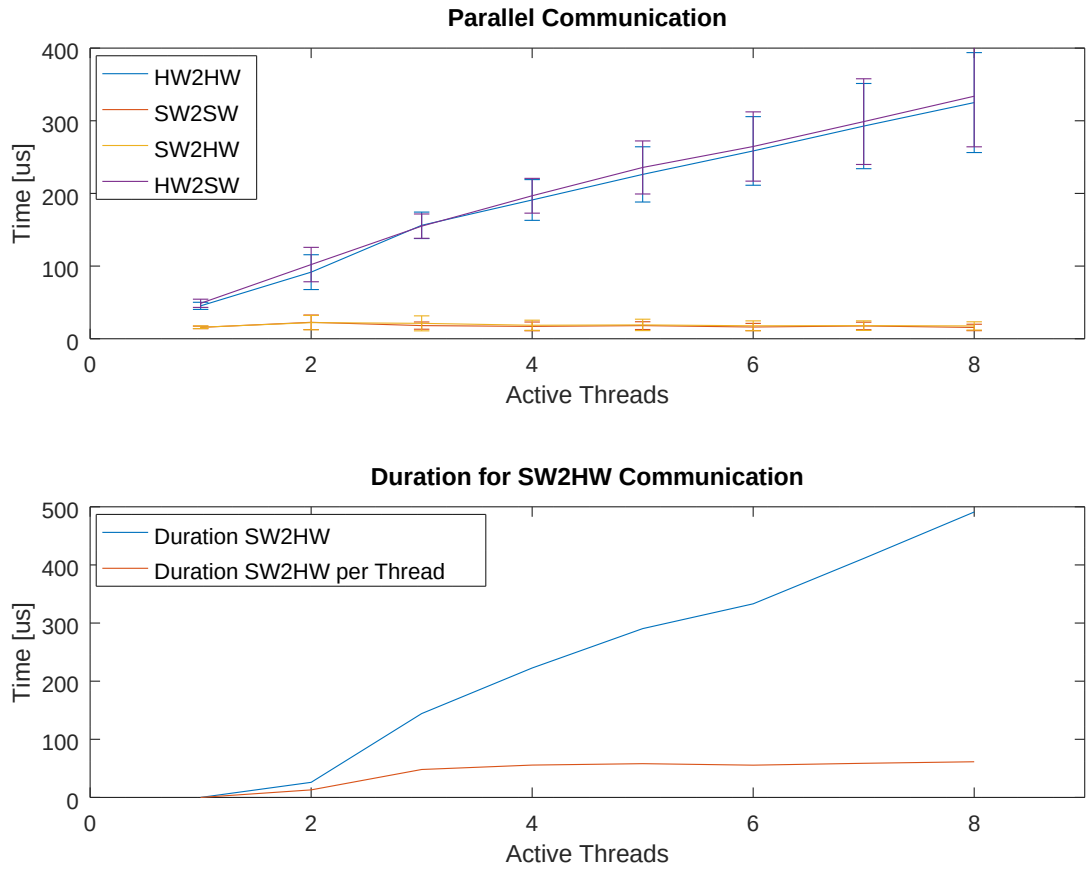

Figure 4.10: Parallel Mailbox Access Setup Execution Time

Memory Access

The next setup is constructed for the investigation of the memory access behavior in parallel. Like the previous setup, all eight threads are triggered by the AXI GPIO module. This leads to simultaneous starting times for all threads.

On the other hand, an additional AXI-memory block is added to the design. The reasons for this are the AXI-Touch and AXI-Servo modules which are added to the design. Since

the control program needs to access them during run time, both AXI-modules must be considered in these investigations. This block contains a block of 32 kByte local memory and can be accessed by the AXI-interface. The address space of the block is mapped onto the virtual address space of the ReconOS application. The overall setup design is shown in figure 4.11.

The difference compared to the previous setup is the communication interface to the processing system. In this experimental setup, the communication is done through the memory interface, not the OSIF interface. For a memory access through the processing system, all threads involved share the ReconOS memory subsystem, which is why the behavior is of interest for parallel accesses.

The investigations should differentiate between two different test scenarios: burst memory access and single access. The burst memory access investigations give insight into the bandwidth distribution for parallel accesses. For this, a block of 32 kByte is transferred from and to the hardware thread to the main memory or the AXI memory. Through the Round-Robin scheduling policy of the arbiter of the memory interface, a linear increase of the transfer time depending on the parallelism expected.

The results of the burst memory access are shown in figure 4.12. The figure shows both the read and write access to the main memory (left) and the AXI-memory (right). The results of the measurements are as expected. The transfer time for access to the main memory increases linearly with the number of threads. The access to the AXI-memory is slower than the access to the main memory due to the slower AXI interface.

In the second test case, single memory accesses are done on the main memory and the AXI-memory. In order to load the memory system, this access is repeated several thousand times by all threads. This ensures that the memories such as the FIFOs in the memory subsystem are filled.

The results of the measurement are shown in figure 4.13. Like before, the results for the main memory are shown on the left side and for the AXI-interface on the right side.

For the AXI-interface, the mean time for a single access on the memory increases linearly with the parallelism both for write and read accesses. Therefore, the memory throughput per thread decreases reciprocally. Read accesses are slower by the factor of two, since the read access require a bidirectional communication compared to the unidirectional communication of write access.

This is also valid for the access by a thread for the examinations at the main memory. With increasing parallelism, the difference between the two access types decreases, which is due to the optimizations by the main memory controller in the processing system.

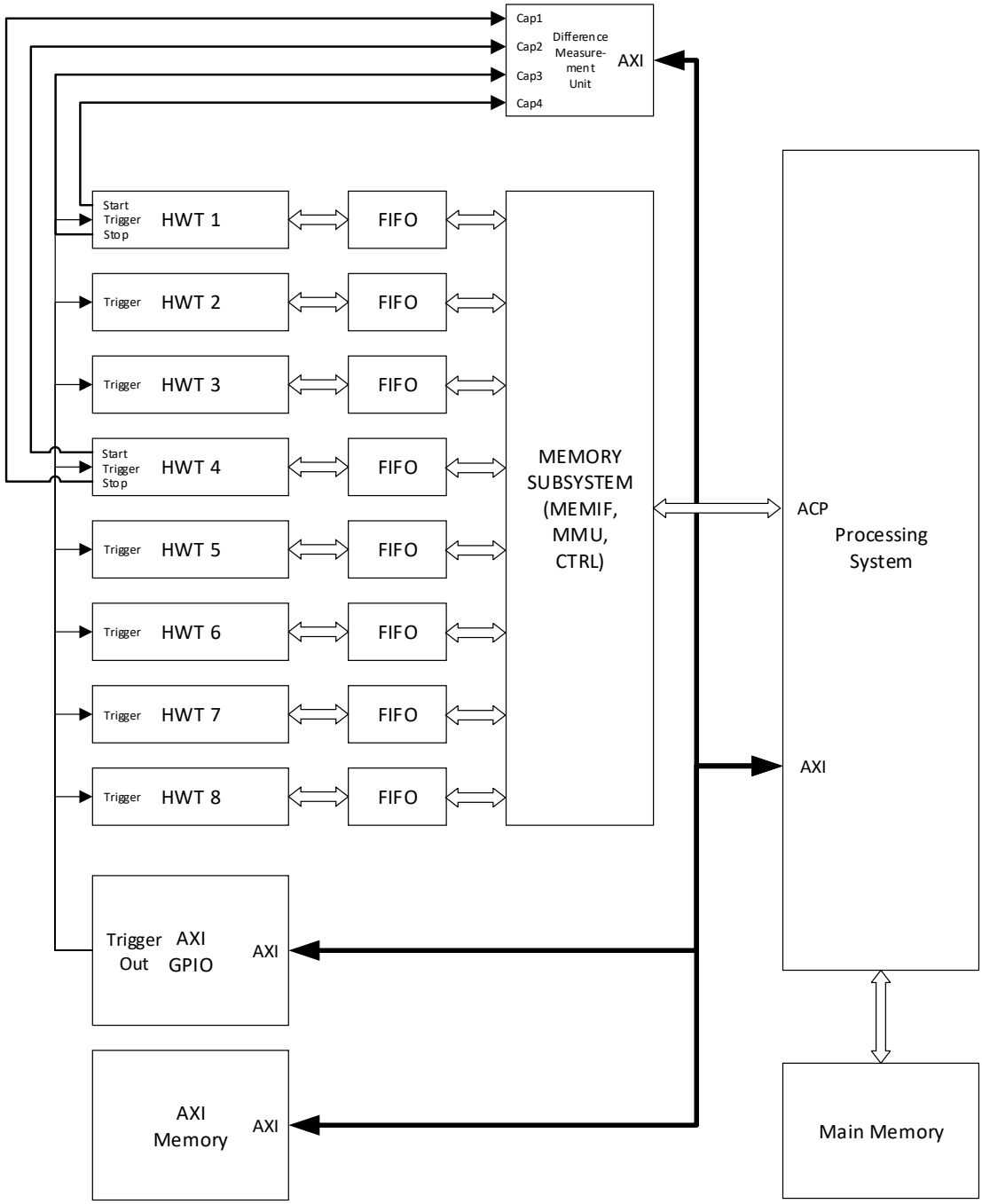

Figure 4.11: Parallel Memory Access Block Design

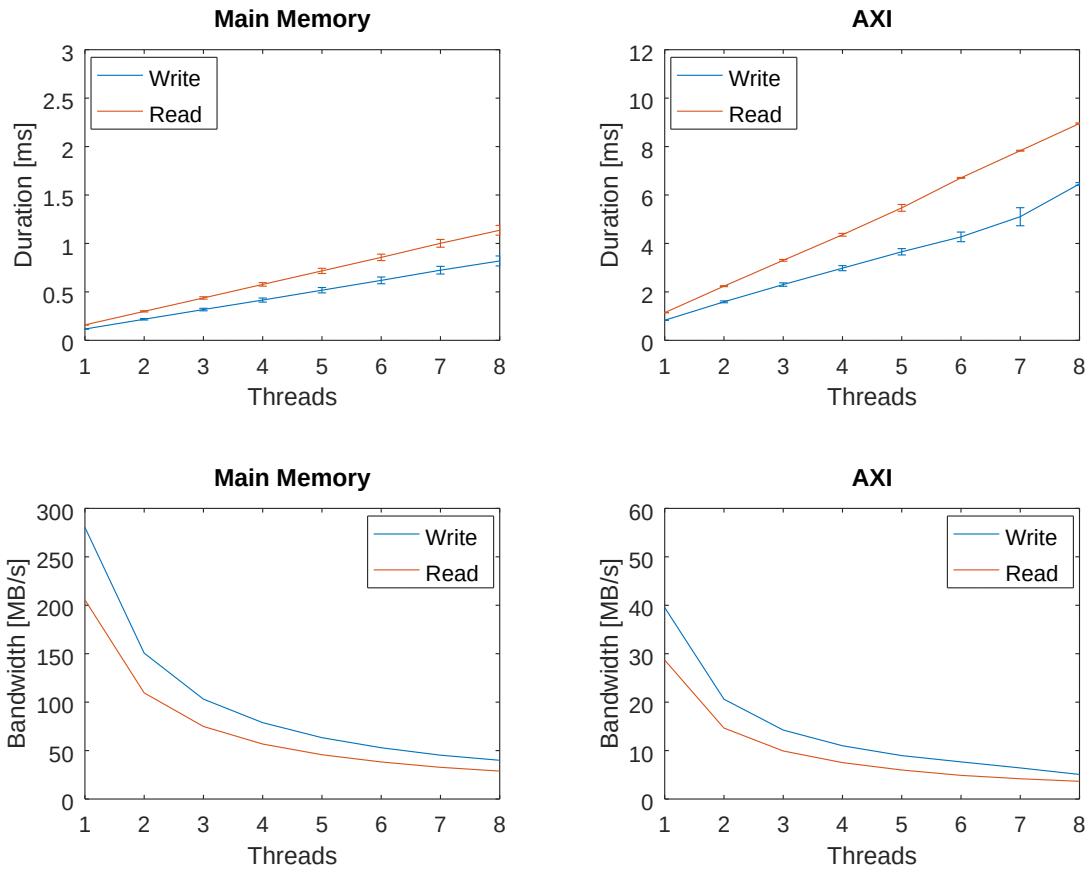

Figure 4.12: Burst Access to the AXI Memory and Main Memory. Top: Time per Access, Bottom: Bandwidth per Thread

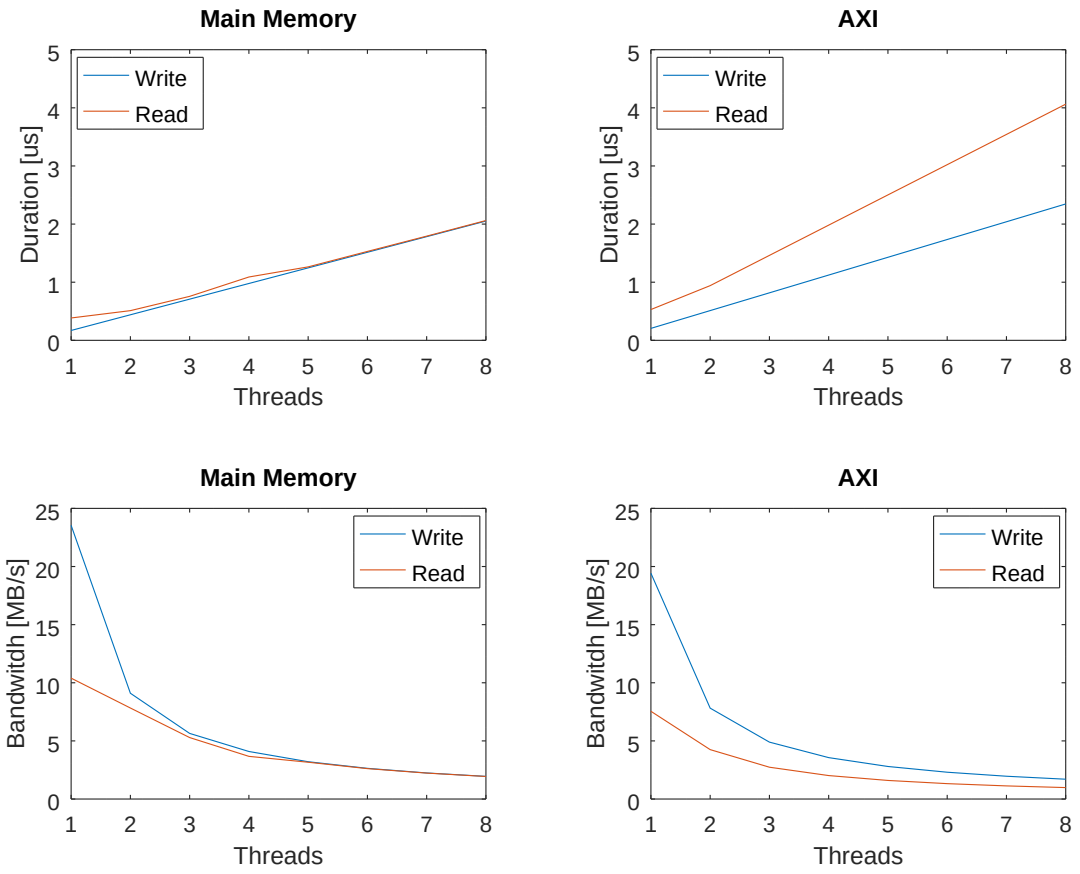

Figure 4.13: Single Double Word Access to the AXI Memory and Main Memory. Top: Time per Access, Bottom: Bandwidth per Thread

4.5 Scheduling and Parallelism

This part investigates the question which degree of parallelism should be used for the implementation. Generally, all tasks should be executed in hardware, if there are enough resources available, because

- the size of the FPGA is fixed and due to the evaluation board, it is not changed to a smaller type, even if in principle a smaller FPGA would be sufficient. Therefore, a smaller FPGA design does not lead to a more cost-efficient implementation of the overall demonstrator.
- in general, the execution on the FPGA is faster and more energy-efficient since lower clock frequencies and higher parallelism are expected. The execution time measurement shows that in all cases the hardware implementation is faster than the regarding software implementation.

The disadvantages of a FPGA implementation in contrast to a software implementation, such as the higher development costs, are omitted in this case, because the functions have already been implemented in hardware. Besides that, the usage of high-level synthesis allows shorter development times compared to classical hardware description languages like VHDL (Very High-Speed Integrated Circuit Hardware Description Language).

During this sub chapter, four different approaches are presented. The first approach is the pure parallel approach, which was already mentioned. The approach uses instances of all hardware threads for all of the three demonstrators in parallel.

The second approach uses dynamic partial reconfiguration for multitasking on the FPGA. The motivation for this approach is the investigation of dynamic reconfiguration in a time-triggered system within the period.

The third approach uses the fact that the implementation of the threads is the same for all three demonstrators. Therefore, every thread-type has only to be implemented once and then reused multiple times during one period.

The last approach is the pure software implementation, which works without any hardware threads. This implementation is made possible by the extensions on ReconOS regarding priority-based scheduling, since the default scheduling policy would not allow such systems.

Pure Hardware Approach

Fortunately, the entire design including both video processing threads can be implemented in the FPGA without having to run parts of the control program in software space constrained. Therefore, the placement problem contains the trivial solution of being able to run all threads in hardware.

The chronological sequence of the operations for the first platform is shown in the figure 4.14. For all other of the three platforms the calculation takes place simultaneously and in the same chronological order.

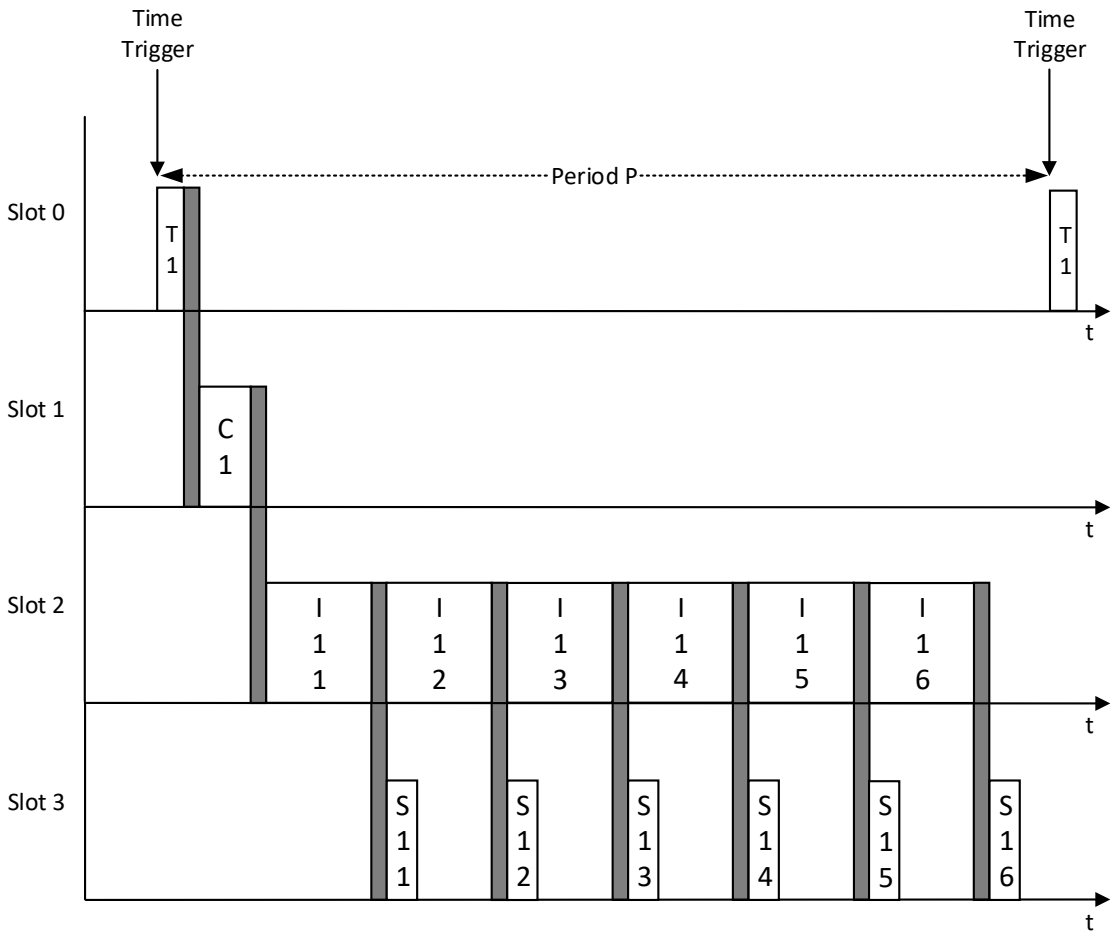

Figure 4.14: Full Parallelism Scheduling (T: touch thread, C: control thread, I: inverse thread, S: servo thread)

Additionally to the control loop threads, the hardware video processing thread is instantiated. Because of the parallel execution, the throughput of the video processing is bounded by the execution time of the thread. Other limitations, e.g. the execution time of the HDMI driver are neglected due to the parallel execution in software.

The gray gaps in the scheduling indicates the overheads, which are generated by the communication via mailbox. The temporal behavior of system calls like mailbox communication and the execution times of the threads are based on the measurements in this chapter.

The processing time of an overall control-loop cycle is the time between the reading of the position data from the AXI-touch module and the writing of the last angle in the AXI-servo module of the regarding demonstrator. The pure calculation time by the threads is given by equation 4.5. The run time of the servo thread is taken into account only once, because the processing usually runs at the same time as the much longer inverse thread.

$$t_{HWEXEC} = 6 \cdot t_{INVERSE} + t_{CONTROL} + t_{TOUCH} + t_{SERVO} = 1208\mu s \tag{4.5}$$

In addition, there are the times for mailbox communication, in which the changed behavior through parallel use must be taken into account. Therefore, the communication between the inverse thread and the servo thread is de facto a double mailbox communication, because the inverse thread already makes a new request for new input data shortly after the angle has been set. The point in time between the control threads and the inverse thread is also assumed to be double communication, since the control thread executes several mailbox-put operations in this case. According to the measured behavior of sub chapter 4.4, the hardware-to-hardware executions take twice as long as a single communication.

Therefore, the overall communication time is given by equation 4.6. The capital letter T indicates the property as a random variable.

$$T_{COMM} = 2 \cdot T_{HW2HW} \cdot 6 + T_{HW2HW} \cdot 2 = 14 \cdot T_{HW2HW} \tag{4.6}$$

This leads to the overall calculation time $T_{PARALLEL}$ of

$$T_{PARALLEL} = t_{HWEXEC} + T_{COMM} \tag{4.7}$$

However, the presented calculations are only valid for one platform. In case the control program calculates for several platforms at the same time, a parallel use of the OSIF-interface must be assumed. The communication time therefore increases approximately linearly with the number of platforms. Assuming a mean communication time of $\bar{T}_{HW2HW} = 44.14\mu s$ (shown in figure 4.7) for the mailbox communication, every additional platform contributes a communication overhead of $617,96us$ for every platform.

According to the equation and the measured values from the models in these chapters, a mean execution time of about $t_{PARALLEL} \approx 1.83ms$ for a demonstrator results, whereby the execution time increases in about $617,96\mu s$ with each additional platform.

As already mentioned, the video processing throughput is bounded by the execution time of the filter kernel and the communication time for the input frame pointer (equation 4.8). Since the address is already in the mailbox at the time of request, the communication time for a system call is considered instead of the communication time for software-to-hardware communication. By using the measured values from table 4.2 and the figure 4.7, a mean value of $T_V = 127.7ms$ is determined for video processing by the Sobel-filter and $T_V = 30.3ms$ for the RGB2GR-filter.

$$T_V = T_{SYSCALL} + t_{FILTER} \tag{4.8}$$

Since the communication time is a random variable in general, the usage of the mean value for the period would lead to a deadline miss in most of the time. Therefore, an accepted probability must be defined for the system design in which the deadline may be exceeded. In the case of the control loop, no deadline misses should be accepted, which is why much higher periods are used. As described in the background chapter, however, the use of Linux as an operating system can also lead to unexpected latencies. Therefore, even the use of the maximum values does not represent security for a hard real-time system, as it is required for example for security relevant systems.

Reconfiguration Approach

For another application scenario, it is assumed that the controller application will run on a different FPGA with limited resources. Therefore, the current implementation does not fit in the actual FPGA type. The possibility of executing parts of the control loop in software should also be ignored for the motivation of this approach. The problem results in the need to execute hardware threads sequentially on the FPGA, which leads to the need of dynamic reconfiguration of the FPGA, especially dynamic partially reconfiguration (DPF). For the realization of the reconfiguration of the threads, the mechanisms described in chapter 3.3 are used.

The reconfiguration should focus on the control and the inverse thread, because these two thread types have the highest resource consumption of all threads in the control loop. The low resource savings by reconfiguration of the servo or touch slots do not justify the effort of a reconfiguration.

However, non-state-less threads like the control thread require additional memory to save the context of the thread. This can be done either by additional local memory BRAM in the FPGA but also by the usage of the main memory of the processing system. For the reconfiguration case in this thesis, the main memory option is chosen. The threads to be reconfigured are assigned a memory area via the initialization data, where they can store or unload their context.

Since the time for the reconfiguration according to table 4.3 far exceeds the time for the calculation of the control loop, the minimum period is limited by the sum of the reconfiguration times. The execution of the hardware threads for the control loop can be done during the reconfiguration.

$$P_{min} = 3 \cdot (t_{recon_3} + t_{recon_4}) \tag{4.9}$$

Furthermore, the remote reconfiguration server cannot be used in this implementation for preserving P_{min}, since reconfiguration by the server would cause the period to be extended by another reconfiguration phase. If the reconfiguration server should still be used, the period of the control program has to be extended by the time for reconfiguration of slot 12.

$$P'_{min} = 3 \cdot (t_{recon_4} + t_{recon_4}) + t_{recon_{12}} \tag{4.10}$$

Since the reconfiguration cannot be interrupted in the existing implementation, a full time slot must be kept in each period in which a possible reconfiguration of the video filter would be carried out. The reconfiguration would always be done in the same phase of the period, otherwise the sample accesses to the touch module and the write accesses of the servo module would no longer be done equidistantly. The maximum latency in this case for a reconfiguration request to the remote reconfiguration server would be approximately $3 \cdot (t_{recon_3} + t_{recon_4})$.

The execution of the control loops is partially overlapping. So, the processing of the next cycle can already be started after the last reconfiguration.

The resulting schedule for the reconfiguration approach is shown in figure 4.15. For reasons of readability, the standards in the schedule are partly distorted. In fact, the phases of the reconfiguration dominate the schedule considerably more.

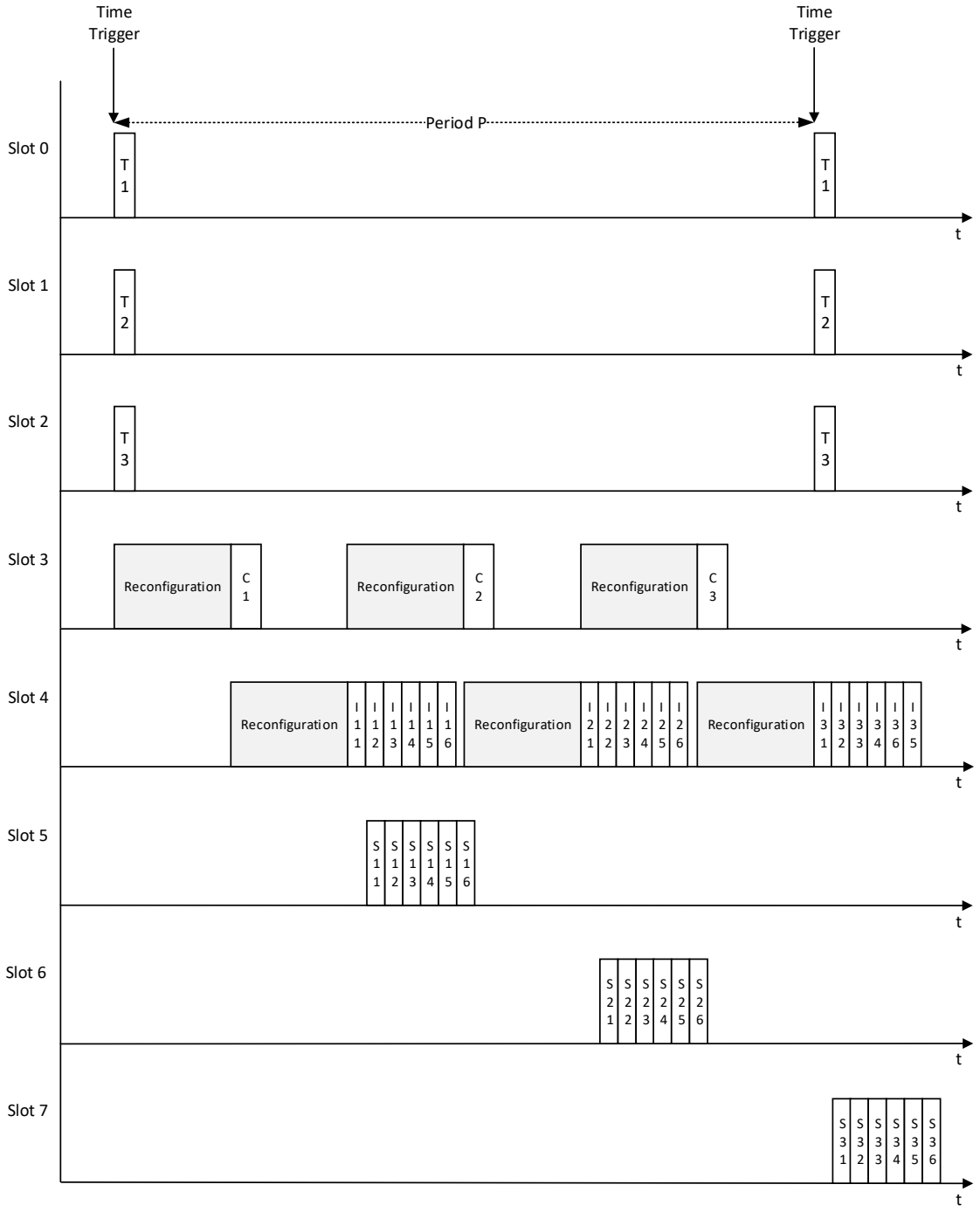

Figure 4.15: Schedule with Reconfiguration in the Control and Inverse Slot (T: touch thread, C: control thread, I: inverse thread, S: servo thread)

Reset Approach

However, due to the poor utilization of the slots for the servo and touch threads in particular, the question arises as to whether reserving resources really makes sense. An alternative would be, for example, to reuse the corresponding slots for other threads by partial reconfiguration. However, the reconfiguration of the slots for the touch and servo thread, which are used only slightly in terms of time, is omitted here, since the overhead in terms of time is in a bad relation to the resources saved due to the reconfiguration.

A more resource and time optimized scheduling uses the context-change mechanism, which was also used for the partial reconfiguration, but without subsequent reconfiguration. This approach avoids the long times for the reconfiguration of the slots.

For a context change in a slot the corresponding reset flag is set first. The RC-flag is recognized by the thread and afterwards a request for the context change is set via the mailbox. After that, the reset signal for the slot is set and the pointer to the initialization data is changed. The reset signal is cleared, and the thread starts the processing.

The resulting overhead T_{CS} for the context switch is shown in equation 4.11. The communication time for the request T_{SW2HW} and the communication time for request of the initialization data $T_{SYSCALL}$ determine the main offset for this approach.

$$T_{CS} = T_{SW2HW} + T_{SYSCALL} + T_{OTHERS} \tag{4.11}$$

The overhead added by reading and writing the context from main memory can be neglected since the size of the context are only a few bytes. The low duration for memory accesses in order of microseconds has been shown by the memory bandwidth experiments. In addition to the mentioned values there are further overheads like setting and resetting the reset signal etc. These values are summarized under T_{OTHERS}.

Due to the overlapping execution of the control loop threads with the context-switch mechanism, the processing time for each control loop is not extended due to the overhead T_{CS}. Therefore, the run time matches the processing time of a single platform at the full parallel design.

$$t_{HWEXEC} = 6 \cdot t_{INVERSE} + t_{CONTROL} + t_{TOUCH} + t_{SERVO} = 1208us \tag{4.12}$$

$$T_{RESETAPPROACH} = 14 \cdot T_{HW2HW} + t_{HWEXEC} \tag{4.13}$$

Pure Software Implementation

In contrast to the fully parallel approach, which was presented before, a pure-software based approach is discussed here. In contrast to the hardware approaches presented, a pure software approach also requires reserve times for the operating system in the period. This time includes, for example, computing time for the scheduler and context switching, the network thread, etc. Additionally, CPU time is needed for the video processing as well as for the kernel and userspace drivers for the video processing chain. Therefore,

the minimum period for the time-triggered system cannot be determined solely from the execution times of the software threads.

This possibility for implementation is made possible by the enhancements to ReconOS regarding priority-based scheduling. Without this change, it is not possible to classify the threads according to priority. This allows threads with a long execution time but low priority like video processing threads to displace the threads of the control loop.

The assignment of priorities for this implementation approach is done by deadline monotonic scheduling. This scheduling policy assigns priority by its deadline inversely proportional. The precedence graph described in 3.2.3 is used to determine the deadlines. Priorities are assigned in increasing order from right to left starting at 75.

The resulting priorities are shown in table 4.4. Since all threads including the video processing are running in software, the remote reconfiguration server is not considered here. The Sobel and RGB2Gray are scheduled with the `SCHED_OTHER` policy since these threads do not behave cooperatively.

Table 4.4: Priorities for the pure Software Implementation

Thread	**Priority**
Delegated Threads	80
Cycle Timer	79
Touch	78
Control	77
Inverse	76
Servo	75
Sobel	0
RGB2Gray	0

4.6 Chapter Conclusion

In this chapter, investigations and improvements regarding the operating system and ReconOS are described. ReconOS is extended by the possibility for priority-based scheduling. This feature enables the implementation of pure-software implementations.

For the providing of execution times of the threads, two measurement environments are presented. They allow the measurement of the execution time of both hardware and software threads.

Additionally, overheads due to system calls and the parallel usage of parts of the ReconOS infrastructure are determined. Some of the gained insights are used in a later part of this chapter to provide execution time estimations for the control loop threads. Therefore, four different approaches are presented in which different degrees of parallelism are used for the calculation of the control program.

5 Evaluation

This chapter will evaluate the results and compare them with the questions of the problem definition. First, the changes to ReconOS and the used Linux kernel are evaluated by the latency. For this, the wake-up time after the `nanosleep` function is measured due to the global timer of the ARM Cortex-A9 cores.

Afterwards, a comparison of the different implementations based on the approaches from the previous chapter is made. The implementation is evaluated based on the used resources and the processing times. As described in chapter 3, the Kalman filter is used for reducing noise and disturbances. The reaction of this filter to disturbances is shown in the following part of this chapter.

Finally, a comparison is made with the questions that were mentioned in the introduction to this work. Due to that, the completeness of the answers should be shown, but also an overview and a reference to the detailed answers should be given.

5.1 ReconOS Real-time Improvements

The previous chapter 4.1 describes changes made to the Linux kernel and the ReconOS framework used. The effect of these changes will be investigated in the following.

Since the control loop is designed as a time-triggered real-time system, disturbances in the period of the cycle-timer effects the period of the data sampling of the whole system. Therefore, the cycle-timer should provide a deterministic behavior with less deviations in the cycle-time. This motivates investigations in the improvements by the changes which have been made.

For the investigations, four different test cases are executed over five minutes of time. The cycle time of the thread is set to 20 ms. In these test cases, the period of the cycle timer thread is measured by the global timer of the ARM Cortex -A9 processor. The resulting period is calculated by the difference of two-time captures. The listing 5.1 shows the modification which is made on the cycle timer thread for the measurement. The test investigates not only the behavior of the `nanoleep()`-function but also the effect to the `pthread_mutex_lock`, `pthread_cond_broadcast` and `pthread_mutex_unlock` in combination with the `nanosleep`-function.

Listing 5.1: Cycle Timer Loop extended by Time Measurement

```
while(1)
{
    pthread_mutex_lock(cycle_timer->mutex);
    pthread_cond_broadcast(cycle_timer->cond);
    pthread_mutex_unlock(cycle_timer->mutex);

    nanosleep(&tim , NULL);
    //Capture actual timer value
    a9timer_caputure(a9timer, &(log_cycle_timer.a9timer_capture),
        A9TIMER_CAPTURE_START);
}
```

The four tests cases investigate different combinations of modifications that are made during this thesis. This includes the combination of a Linux kernel which was modified by the *PREEMPT_RT* patch and the origin Linux kernel without this modification. On the other hand, the two scheduling strategies `SCHED_FIFO` and the default scheduling policy `SCHED_OTHER` are tested.

For additional stressing of the system, the test program *cyclictest* is executed in the background with four threads running. *Cyclictest* is part of the official Linux kernel sources and is usually used to measure the latency of real-time threads. The general approach is similar to the presented approach in the cycle timer thread modification. In the test case for this thesis, the tool is running without real-time behavior, since the purpose is only to generate a higher workload through additional periodic threads.

The results of the tests are shown in figure 5.1 in the histogram. For each test case, the mean value, the standard deviation and the maximum measured value are given in the legend of the figure.

As shown, the combination of the *PREEMPT_RT* patch together with the real-time scheduling policy `SCHED_FIFO` leads to the best results in the test. For this test case, the mean value, the standard deviation and the maximum value are the minimum of all test cases. The test also shows small improvements due to the *PREEMPT_RT* regarding the maximum latency of about 40 μs compared to the non-patched Linux kernel. On the other hand, not using a real-time scheduling strategy leads to significantly higher latencies (e.g. comparison of the first two test results). Not only the mean value of the latencies increases without the changes on the scheduling policy, but also the deviation of the latencies.

The essential insight from this measurement is the fact that only the change of the scheduling policy results in significant changes on the real-time behavior. The deviation of the cycle timer is not only smaller, but the cycle timer also reacts faster after returning from the `nanosleep` function. The improvements through the patch are rather marginal.

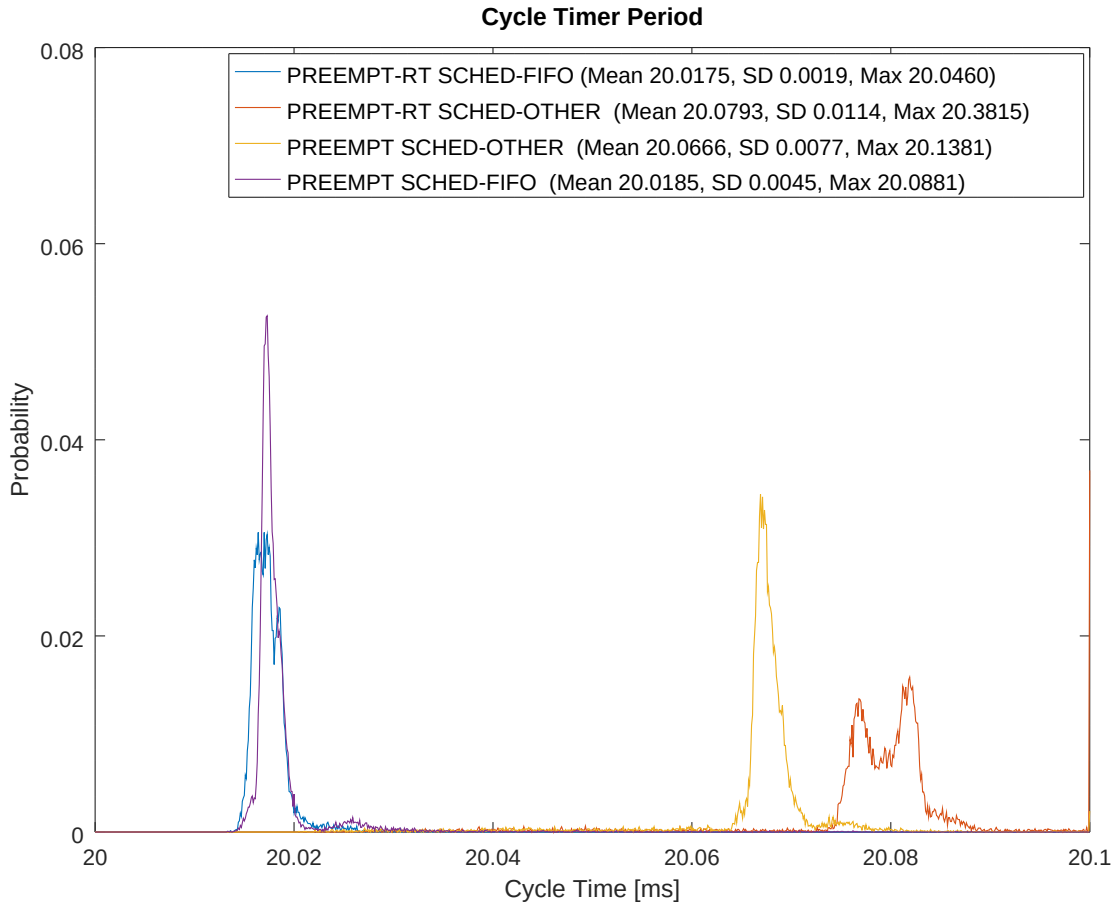

Figure 5.1: Cycle Time Period on a fully preemptive kernel / not full preemptive kernel with real-time scheduling / without real-time scheduling

5.2 Implementation Evaluation

The evaluation of the different implementations is divided into two parts: the first part compares the resource utilization of the four approaches. The second part shows the resulting execution times of the implementations for each platform.

Resource Utilization

For the evaluation of the implementations, the utilization of the available resources used is shown in figure 5.2. The comparison is limited to the number of used look-up tables (LUT), distributed ram (LUTRAM), flip-flips (FF), block memory (BRAM) and digital signal processing units (DSP). Other parts of the FPGA, e.g. the number of used input-output pins, are not considered in this comparison since most of them are equal for all implementations. All compared implementations contain the modules for the video

processing support, e.g. the direct memory units or the HDMI transceivers and receivers. Of course, all implementations include the AXI-modules for the servo and touch interface but also the ReconOS infrastructure.

As expected, the parallel implementation of all hardware threads consumes the most resources for all types. This implementation is mainly bounded by the availability of digital processing units which are almost completely used. Due to that, no more instances of the most time-consuming threads are possible.

The two approaches (Reset-based implementation and DPR-based implementation), which are based on sequential processing of the platforms, use approximately the same number of resources. Differences are the result of different constraints due to the design flow of the partial reconfiguration.

For the purely software-based approach, only the infrastructure for video processing and the AXI-touch controller the AXI-servo controller is required. Therefore, this implementation allows the estimation of the resources needed for this infrastructure.

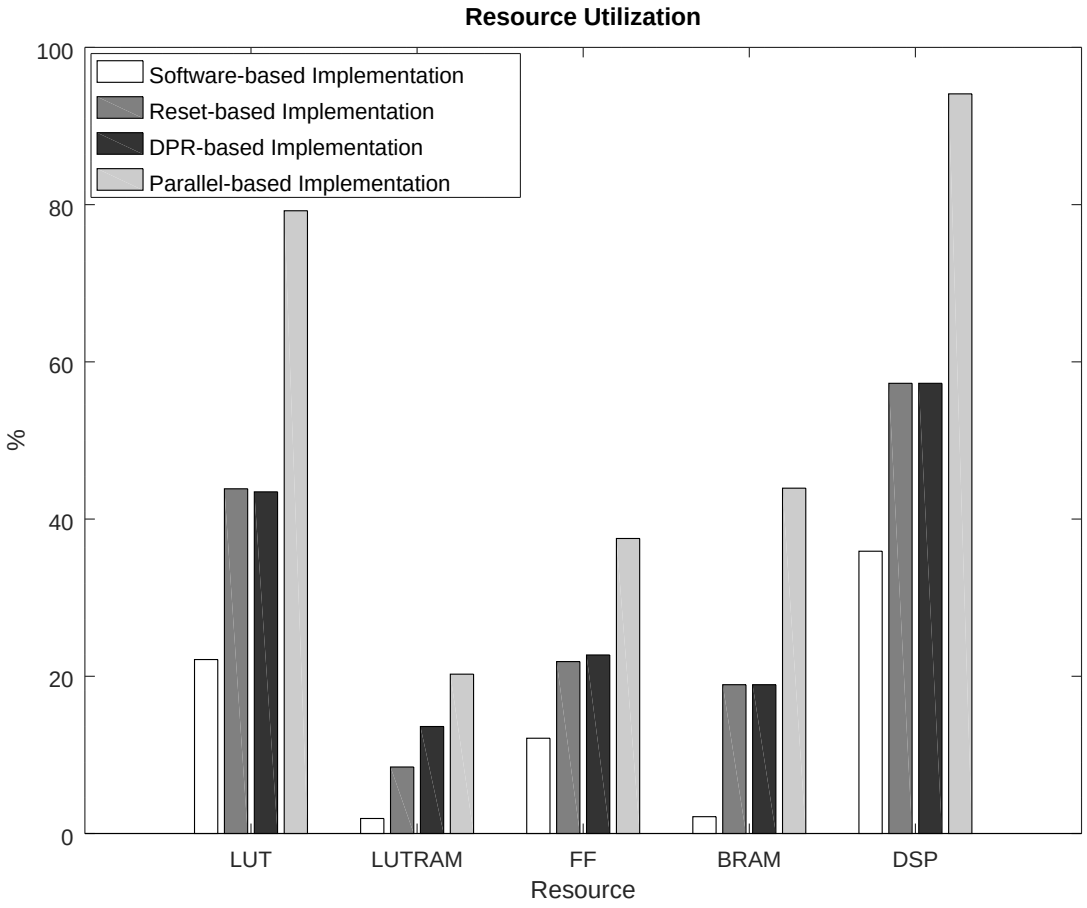

Figure 5.2: Resource utilization of the different implementations

Full Parallel Implementation

For the evaluation of the execution time of the various implementations, the time between reading the position from the touch module and writing the sixth angle into the servo module is measured for each of the three platforms. The results of the processing time measurements are shown in table 5.1.

Table 5.1: Measured mean execution times for the parallel implementation

Number of Platforms	Platform 0	Platform 1	Platform 2
1	1.9887 ms	-	-
2	2.5837 ms	2.5859 ms	-
3	3.3390 ms	3.3288 ms	3.3344 ms

As expected, the processing time of the control loops increases with the increasing number of platforms. Since the processing of the algorithms completely takes place in parallel in hardware, the increased processing time must be due to longer communication times through the mailbox communication.

The increase of the average execution time of about 0.6 ms (two demonstrators compared to one demonstrator) and about 0.75 ms (three demonstrators compared to two demonstrators) fits the predicted orders of magnitude predicted in 4.5.

The execution time of the video filters is independent from the number of demonstrators. The measured mean execution time for the Sobel-filter is 127.44 ms and for the RGB2Gray-filter the mean execution time is 30.03 ms. In other words, the implementation allows frame rates of 7.9 frames per second for the Sobel-filter and 33.3 frames per second for the RGB2Gray-filter.

Partial Reconfiguration Implementation

As already mentioned in the previous chapter, the approach of partial reconfiguration is not suitable for the control of demonstrators. This is due to the long delay times caused by the partial reconfiguration phases, which do not allow the control of an unstable control system with double-integral behavior with the presented controller design. The corresponding tests inevitably resulted in a loss of the ball.

The approach of partial reconfiguration during the period could be used for less dynamic systems. This includes, for example, temperature control of objects with large time constants.

Reset Approach Implementation

For the implementation based on the reset approach, processing times are expected that are close to the measurement results for the full parallel approach in hardware for one

platform. The measured mean values in table 5.2 confirm this result. The processing time remains constant even if the number of demonstrators involved is reduced.

Unlike the full parallel approach for processing, the processing times for this approach must be partially added to obtain the total processing time for all platforms.

Table 5.2: Measured mean execution times for the reset implementation

Number of Platforms	Platform 0	Platform 1	Platform 2
3	2.0012 ms	2.0059 ms	2.0956 ms

Pure Software Implementation

As already described in the previous chapter, the running time of the inverse thread dominates the total running time of the control loop. the result of the running time measurement reflects this result. In the histogram in figure 5.3 it becomes visible that in most runs the processing time of all three demonstrators takes about 8.5 ms, which corresponds almost exactly to six times the running time of the inverse thread.

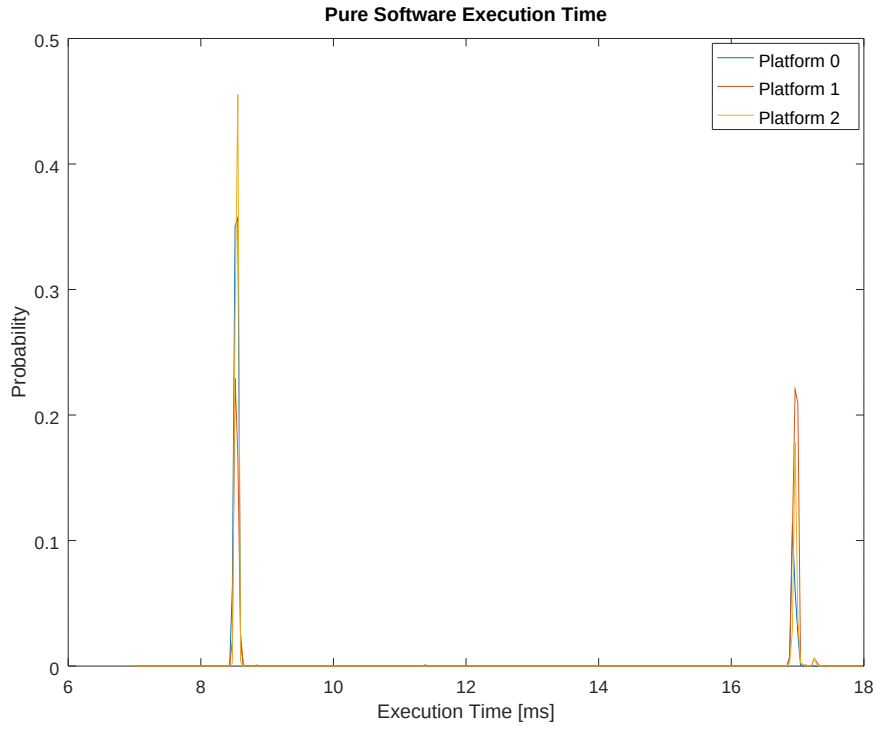

Figure 5.3: Execution Time for the pure software implementation

In a significant number of cases, the runtime is about twice the value, circa 17 ms. In this case, the inverse thread of another demonstrator was executed after scanning by the

touch thread and processing the control thread, resulting in a wait time of about 8.5 ms. The probability of both events is different due to the dual-core processor architecture. Two of the three demonstrators are processed directly, the remaining third must wait for a free core.

5.3 Controller Evaluation

For the evaluation of the Kalman filter, the controller is given a reference value that corresponds to a circular motion on the surface. The resulting controller values are shown in figure 5.4.

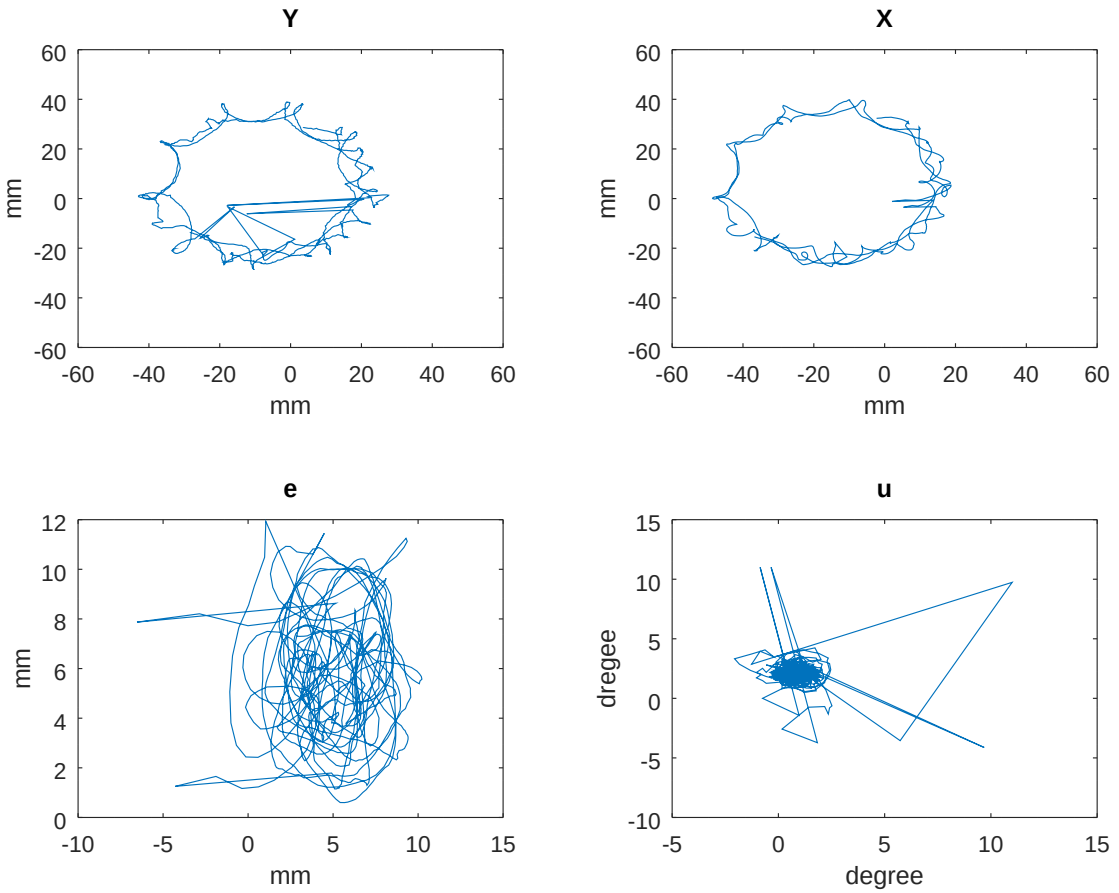

Figure 5.4: Position of the Ball on the Surface with measured position Y, Kalman estimated position X, control error e and resulting control value u

Due to transmission problems, which are mainly caused by electromagnetic interference caused by strong position changes of the servo motors in the transmission line to the touch controller, the incoming position data may be incorrect. Also, an aging of the touch screen or a loss of contact of the ball due to fast movements of the Stewart-platform are possible

sources of interference. The used PD-controller is very sensitive to these disturbances, as these are often large gradients.

In the subplot Y in figure 5.4 such a disturbance can be observed. The used Kalman filter reduces these disturbances which can be seen in the X subplot. Without this weakening there would have been a loss of the ball due to a fast movement of the platform as a reaction of the high gradient.

Moreover, the Kalman filter shows an overall filtering of the measured data, estimating a smoother motion. This filtering therefore leads to a softer control behavior.

5.4 Question Comparison

As the last part of the evaluation of this thesis, a comparison is made with the problems and aims of this work. In this way, an explicit answer to all questions is given. Additionally, the reader should be able to compare the results with the asked questions.

In general, the extend demonstrator provides a set of tasks with different properties regarding real-time constraints. The tasks of the control-loop have to be scheduled under harder real-time constraints than the video processing threads. The remote reconfiguration server contributes an event-trigger task to the set of tasks.

Q1 *How are tasks interfaced with sensors and actuators?*

The connection with sensors and actuators is described in the sub chapter 3.2 as AXI-touch module and AXI-servo module. The existing direct connection to hardware threads has been replaced by an AXI modules, as this allows a more flexible connection to the application. This enables, for example, pure software solutions for the control software.

Q2 *How to provide upper bounds for hardware and software task execution times?*

Approximations for upper bounds for hardware and software execution times are provided in the sub chapter 4.2. Since the analytical determination of the execution time for software threads is hard because of the non-deterministic behavior of the executing processor, the measured results are approximated as execution times.

Q3 *How to model and determine the overheads posed by operating system calls and the hardware/software communication in ReconOS?*

Overheads due to operating system calls and hardware software communication are analyzed in the real-time investigation chapter 4.3. Different setups are used to measure the latency of system calls and inter-thread communication.

Q4 *How to deal with resources shared between tasks, in particular buses and memories?*

Parts in the ReconOS system which performance is influenced due to parallel usage, are investigated in the sub chapter about resource sharing (4.4). The influence of parallel inter-thread communication and parallel system calls on the run time of these is investigated. After that, the influences of parallel memory access by ReconOS-threads is determined. For this, the bandwidth but also the latency for single word access is determined.

Q5 *What degree of parallelism can and should be used, and what are the resulting scheduling and placement problems?*

In the subchapter 4.5, the question of the degree of parallelism is answered. in general, a completely parallel processing in hardware is recommended for this implementation, since the entire design can be accommodated in the FPGA. Nevertheless, the mentioned sub chapter also propose other degrees of parallelism that might be useful for similar problems.

Q6 *Is there a case for partial reconfiguration?*

This thesis describes two application cases of partial reconfiguration. First, the remote reconfiguration server uses this mechanism to replace the filter kernel after a request. The corresponding description can be found in 3.2.3 in the demonstrator chapter. As a second application, partial reconfiguration is used to examine a reasonable degree of parallelism. The description can be found in the section 4.5.

6 Conclusion and Future Work

In the last chapter of this thesis, the results of this work are summarized, and a short summary of the work steps is described. Afterwards, an outlook is given on further possible work on and with this demonstrator as well as on the results achieved.

6.1 Conclusion

The demand for energy-efficient real-time systems as "enablers" for the increasing use of technology of our everyday life requires on the one hand the use of combinations of hardware and software and on the other hand an increase in productivity of the existing development capacities. ReconOS with its programming model offers the possibility to put hardware / software decomposition into practice.

The work presented here is based on the objective to build a real-world example for a real-time ReconOS system. This demonstrator is based on an existing demonstrator, which is extended for the purposes of this work.

The existing demonstrator is extended by two additional Stewart-platforms, which are able to balance a ball on its surface. The position of the ball on the surface is recognized by a touchscreen. All three platforms are controlled by one platform FPGA, which is embedded in a FPGA development board. The Zedboard is also extended by a HDMI input extension board, which provides physical support for receiving HDMI input signals.

After the practical part of this work, the control software for the demonstrator is designed. This includes the software for the balancing of the balls but also digital image processing with two different filter kernels. To enable video processing in the ReconOS framework, userspace drivers for HDMI input and HDMI output are presented.

Since dynamically partial reconfiguration allows the usage of FPGA resources in time-space, two mechanisms for multitasking on the programmable logic are described. This includes the possibility to change parts of the control loop during run-time but also to change the filter kernels. For the last option, a remote reconfiguration server accepts network requests and initiate a reconfiguration on this request.

After demonstrator construction and implementation of the demonstrator hardware software design, real-time investigations are described. This includes the extension of ReconOS by priority-based real-time scheduling but also the applying of the *PREEMPT_RT* patch to the Linux kernel. In the following, execution times for the ReconOS threads are provided and timings of inter-thread communication and system calls are investigated. Since some ReconOS infrastructure is shared by the available hardware threads, the influences of the parallel usage are determined by different measurement setups.

Due to the properties of the demonstrator, there are different possibilities for scheduling and the parallel usage of FPGA resources to execute the control software. Four different approaches are investigated in this thesis.

The evaluation chapter shows the improvement that could be archived due to the Real-time modifications on ReconOS and Linux. Additionally, measurements on the four approaches of parallelism are shown. Finally, the questions described in the problem definition are compared with the results of this thesis.

The investigations in this thesis show that ReconOS based on Linux can be used for time-triggered real-time applications. Improvements in the Linux kernel as well as in ReconOS regarding the real-time scheduling allow the implementation of pure software solutions as well as combinations of hardware and software functionalities.

For use on FPGAs with limited resources the presented approaches can be selected, where hardware threads are executed sequentially on parts of the FPGA. For systems with more stable behavior, dynamic partial reconfiguration can be used. For applications with short cycle times, it is possible to use the approach presented here, in which the initialization data is exchanged during a hardware reset.

By measuring the communication times between different thread types and the performance effects due to parallel usage, the effects can be estimated and taken into account for the later design of a time-triggered system based on ReconOS. This allows the usage of ReconOS for future real-time implementations.

6.2 Future Work

The development and implementation of the Ball-on-Plate demonstrator can be used as a platform for further investigation. This platform offers a playground for the implementation of real-time systems due to its mechanical structure and the functions already been implemented.

In the background chapter, different approaches for providing real-time behavior are described. While this work disregards the investigation of approaches with an additional co-kernel, there would be room for further analyses. Also, the usage of non-Linux-based systems together with the demonstrator is possible. One possible example would be the real-time operating system FreeRTOS.

As a further focus for further work, an implementation from a time-triggered system to a purely event-based system could take place. By the described structure of the control software, time-triggered systems are mainly examined in this work.

Bibliography

[1] Real-time processing – the basis for pc control. `http://www.pc-control.net/pdf/special_25_years_pcc/products/pcc_special_0811_realtime_e.pdf`. Accessed: 2018-08-19.

[2] Zynq-7000 soc data sheet: Overview. `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`. Accessed: 2018-08-19.

[3] J. Ackermann. Entwurf durch Polvorgabe. *At-Automatisierungstechnik*, 25(1-12):209–215, 1977.

[4] Andreas Agne, Marco Platzner, Christian Plessl, Markus Happe, and Enno Lübbers. ReconOS. *FPGAs for Software Programmers*, 9(1):227–244, 2016.

[5] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. pages 331–338, 2006.

[6] M. A. Arshad, M. M. Gulzar, J. K. Qureshi, A. Hayat, M. Shamir, F. Ahmed, and S. Rasheed. Six degrees of freedom robotic testbed for control systems laboratory. In *2017 International Symposium on Recent Advances in Electrical Engineering (RAEE)*, pages 1–6, Oct 2017.

[7] H. Bang and Y. S. Lee. Implementation of a ball and plate control system using sliding mode control. *IEEE Access*, 6:32401–32408, 2018.

[8] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance comparison of vxworks, linux, rtai and xenomai in a hard real-time application. In *2007 15th IEEE-NPSS Real-Time Conference*, pages 1–5, April 2007.

[9] S. R. Bdoor, O. Ismail, M. R. Roman, and Y. Hendawi. Design and implementation of a vision-based control for a ball and plate system. In *2016 2nd International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)*, pages 1–4, May 2016.

[10] Ilja N. Bronstein, Konstantin A. Semendjajew, Gerhard Musiol, and Heiner Muehlig. *Taschenbuch der Mathematik*. Europa Lehrmittel Verlag, 9. edition, December 2013.

[11] Jeremy H. Brown. How fast is fast enough ? choosing between xenomai and linux for real-time applications. 2010.

[12] George Charitopoulos, Iosif Koidis, Kyprianos Papadimitriou, and Dionisios Pnevmatikatos. Hardware task scheduling for partially reconfigurable FPGAs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9040:487–498, 2015.

[13] W. Chong, S. Ogata, M. Hariyama, and M. Kameyama. Architecture of a multi-context fpga using reconfigurable context memory. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 7 pp.–, April 2005.

[14] Cosmin Copot, Yu Zhong, Clara Ionescu, and Robin De Keyser. Tuning fractional pid controllers for a steward platform based on frequency domain and artificial intelligence methods. *Central European Journal of Physics*, 11, 04 2013.

[15] Cosmin Copot, Yu Zhong, Clara M. Ionescu, and Robin De Keyser. Tuning fractional pid controllers for a steward platform based on frequency domain and artificial intelligence methods. *Central European Journal of Physics*, 11(6):702–713, Jun 2013.

[16] Marcel Eckert, Dominik Meyer, Jan Haase, and Bernd Klauer. Operating System Concepts for Reconfigurable Computing: Review and Survey. *International Journal of Reconfigurable Computing*, 2016(December):1–11, 2016.

[17] Otto Foellinger. *Regelungstechnik - Einfuehrung in die Methoden und ihre Anwendung*. VDE Verlag, 2013.

[18] S. Gilliland, J. Saniie, and F. M. Vallina. Implementation of elementary functions for fpga compute accelerators. In *2016 IEEE International Conference on Electro Information Technology (EIT)*, pages 0179–0182, May 2016.

[19] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[20] Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. Open source freertos as a case study in real-time operating system evolution. *Journal of Systems and Software*, 118:19 – 35, 2016.

[21] Markus Happe, Andreas Traber, and Ariane Keller. Preemptive hardware multitasking in reconos. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 79–90, Cham, 2015. Springer International Publishing.

[22] Xilinx Inc. Zynq-7000 soc technical reference manual. Online, 2018. UG585 (v1.12.2) July 1, 2018.

[23] Aws Ismail and Lesley Shannon. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. *Proceedings - IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2011*, pages 170–177, 2011.

[24] 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. *IEEE Transactions on Computers*, 62(8):1542–1556, 2013.

[25] S. Jovanovic, C. Tanougast, and S. Weber. A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems. *Proceedings - 2007 NASA/ESA Conference on Adaptive Hardware and Systems, AHS-2007*, pages 358–364, 2007.

[26] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering, Transactions of the ASME*, 82(1):35–45, 1960.

[27] H. Kopetz. Event-triggered versus time-triggered real-time systems. In Arthur Karshmer and Jürgen Nehmer, editors, *Operating Systems of the 90s and Beyond*, pages 86–101, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[28] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. The Springer International Series in Engineering and Computer Science. Springer US, 1997.

[29] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[30] Enno Lübbers and Marco Platzner. Reconos: An RTOS supporting hard- and software threads. *Proceedings - 2007 International Conference on Field Programmable Logic and Applications, FPL*, pages 441–446, 2007.

[31] Enno Lubbers and Marco Platzner. Communication and Synchronization in Multithreaded Reconfigurable Computing Systems. *In Proceedings of the 8th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, July 2008*, (July), 2008.

[32] H. Lutz and W. Wendt. *Taschenbuch der Regelungstechnik: mit MATLAB und Simulink*. Edition Harri Deutsch. Verlag Europa-Lehrmittel, 2014.

[33] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Comput. Surv.*, 52(1):18:1–18:36, February 2019.

[34] Christoph Rüthing. Self-adaptation in programmable automation controllers based on hybrid multi-cores. Master's thesis, Paderborn University, 2015.

[35] Claudio Scordino and Giuseppe Lipari. Linux and Real-Time: Current Approaches and Future Opportunities. *Anipla 2006*, 2006.

[36] H. Simmler, L. Levinson, and R. Männer. Multitasking on FPGA Coprocessors. pages 121–130, 2007.

[37] Dan Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley-Interscience, New York, NY, USA, 2006.

[38] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 2012.

[39] D. Stewart. A platform with six degrees of freedom. *Proceedings of the Institution of Mechanical Engineers*, 180(1):371–386, 1965.

[40] 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. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2179–2192, 2013.

[41] K. Yaovaja. Ball balancing on a stewart platform using fuzzy supervisory pid visual servo control. In *2018 5th International Conference on Advanced Informatics: Concept Theory and Applications (ICAICTA)*, pages 170–175, Aug 2018.

Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen als Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Ort, Datum Unterschrift