Bachelor Thesis

Time-multiplexing for Remote-controlled Applications

René Küttner

27. July 2015

Technische Universität Dresden
Faculty of Computer Science
Institute of Systems Architecture
Chair of Operating Systems

Supervising professor: Prof. Dr. rer. nat. Hermann Härtig
Supervisors: Dr. Marcus Völp
M.Sc. Nils Asmussen

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und dabei keine anderen als die angegebenen Hilfsmittel verwendet habe.

Dresden, den 25. July 2015

René Küttner

Abstract

Systems are getting more and more heterogeneous due to the performance and energy advantages of special purpose cores. However, today's operating-systems cannot treat these cores as first-class citizens, i.e. run untrusted code on them and provide OS functionality. M^3 (Microkernel for Minimalist Manycores) is an operating system that strives to support arbitrary cores while treating all of them as first-class citizens. Currently, M^3 supports only one application per core, which does eliminate the costs of context-switches. On the other hand, it wastes resources for applications that are not performance-critical like printer services, log services or virtual terminals. This thesis presents an approach for remote-controlled time-multiplexing in M^3 that ensures per-application isolation. It also shows interesting aspects of the associated proof-of-concept implementation and discusses the results of its evaluation. Finally, possible future work based on this thesis is suggested.

Contents

List of Figures

List of Tables

1 Introduction

Not long ago, improvements in computational power of processors have been achieved mainly by increasing the clock frequency. However, there are some fundamental technological limits when doing that [Tan15]. Computer systems with more than one processor became available to cope with that issue. They allowed for a gain in processor performance aside from clock speed. Unfortunately, having many general purpose processors has lead to new problems. For example, cache coherency protocols have become very complex with the increasing number of processing units in a multicore system.

Recently, so called many-core systems have become available. These systems come with hundreds or even tens of thousands of processor cores. Manycore platforms are expected to be increasingly populated with different specialized cores, leading to more and more heterogeneousness. On one hand, these specialized cores can provide lower energy consumption, which makes them interesting for mobile devices and other markets where energy is a scarce good. On the other hand, these cores can be optimized according to their purpose. This allows for great performance boosts as seen in Graphics Processing Units (GPUs) for example.

While hardware platforms evolve in the direction of heterogeneousness, operating systems have to deal with this new situation. The traditional approach is to have privileged kernels that run on each physical core. It does not scale very well on heterogeneous platforms where cores may implement a lot of different features and architectures. This prevents today's operating systems from fully utilizing the power of such platforms.

Microkernel for Minimalist Manycores, or M^3, is an operating system concept that aims to cope with this situation. It is intended to support highly heterogeneous platforms. All the different cores are treated as first-class citizens in M^3 by providing them with operating system functionalities and support for isolated execution of untrusted code. This is achieved by placing a small hardware component next to each core called the Data Transfer Unit (DTU). It manages the core's access to external resources and is controlled by the operating system's kernel.

Since all cores are expected to be different, M^3's kernel is run on its own core, controlling other applications remotely by supervising their DTU. In the current implementation of M^3, applications are executed on dedicated cores only. Operating system functionality is provided remotely via the DTU and by abstractions in M^3's system library. This is beneficial for performance due to the omitted overhead of local context-switches. Since

the number of cores in manycore platforms is expected to outreach the usual number of running processes [WA09], this approach should scale well in the future.

However, there are situations where the concept of assigning whole cores solely to a single application may be a waste of resources. Imagine a service program that occupies a core and only receives requests once in a while. The computational power of the core that is unused by the service application most of the time is wasted. This thesis solves this situation by presenting an approach for switching among mutually distrusting applications on the same core over time (time-multiplexing). It also shows that the presented solution is flexible enough to support migration of applications among cores as a side-effect.

The rest of this thesis is structured as follows: Chapter 2 introduces the foundation that this work is built on and discusses related work. Details of the proposed design for time-multiplexing for remote-controlled applications are presented in Chapter 3. An overview of my implementation is given in Chapter 4. It also covers a selection of challenging aspects encountered while implementing the proposed design. The implementation is then evaluated regarding to code-size, binary-size and context-switch performance in Chapter 5. Chapter 6 completes this thesis with a summary of the presented approach as well as a short discussion of possible future work.

2 Technical Background

This chapter provides the basics that are needed to understand the rest of the thesis. It also contains a discussion of related work. To get started, a general introduction to time-multiplexing is given in Section 2.1. Section 2.2 continues with a detailed explanation of the M^3 operating system and introduces important terminology that is used throughout the following chapters. M^3 relies on a small hardware component called data transfer unit that is depicted in Section 2.3. Finally, Section 2.4 completes the chapter with a selection of available manycore operating systems and a discussion of their respective support for time-multiplexing among applications.

2.1 Time-multiplexing in General

When an application is started, operating systems normally create a process to execute it. A single processing core is only able to run one process at a certain point in time. If there are not enough cores available to run all required processes, which is the case for most single- and multi-processor systems today, the need for some mechanism of sharing cores among processes over time arises.

Controlling the access of multiple processes to some limited resource is called *multiplexing*. When the limited resource is time, this procedure is called *time-multiplexing*. In contrast to time-multiplexing there is *spatial-multiplexing* where processes are multiplexed on available space (e.g. cores).

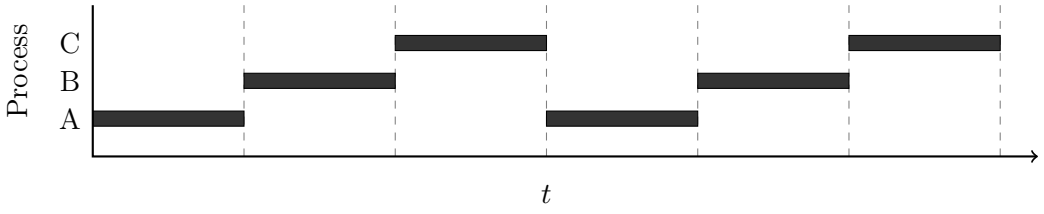

Figure 2.1: Simple time-multiplexing of three processes A, B and C with time slices of a fixed length.

When time-multiplexing of processes takes place, the system has to suspend the currently running process and resume the next one. For this to work, the state of each process

has to be saved somewhere and restored when it is going to be resumed. Data, like the contents of the CPU registers, that forms the state of a process is called the *context* of the process. Switching the state is therefore called *context-switching*.

Traditional approaches for time-multiplexing can be classified into two categories: *non-preemptive* and *preemptive*. Non-preemptive time-multiplexing waits for a process to block or release the core. The system then switches the context to the next ready process. While this is easily implemented, it has some serious drawbacks. For example, a malicious process that is not cooperative could just run forever without blocking. Adding a timeout after which it gets forcibly interrupted may solve this issue. However, this requires hardware support for timer interrupts to give back control to the system after a given amount of time. Additionally, if timer interrupts are available, a more sophisticated mechanism can be implemented: preemptive time-multiplexing. It is fully transparent to applications. In the worst case, the system grants a fixed time to the process that is next to run as depicted in Figure 2.1. If the time is up and the process is still running, the system interrupts it and runs the next process.

Preemptive time-multiplexing is implemented by the majority of today's traditional operating systems. It enables them to run a lot of processes when only a small number of cores is available. Their kernels implement a scheduling algorithm that picks the next process according to its policy. Such policies could be the meeting of certain deadlines for real-time systems or giving high-priority processes more time to run than low-priority ones. For interactive systems, a common goal is to deliver the impression of all processes running concurrently. This is achieved by assigning a higher priority to interactive processes.

2.2 Microkernel for Minimalist Manycores

The results of this thesis are based on the *Microkernel for Minimalist Manycores* (M^3) operating system developed at the Chair of Operating Systems at TU Dresden. At the time of writing this, its implementation is in a very early state and highly experimental. It strives to fully support all the processing power in heterogeneous manycore systems, where different kinds of cores may exist. Thus, it must not rely on certain processor features that are only found in some of the cores, like *Memory Management Units* (MMUs) for example. Such features may be supported optionally, though.

In the context of M^3, a core with its local scratchpad memory is called *Processing Element* (PE). Despite the possible absence of core features like privileged mode or an MMU, each PE is able to execute an isolated piece of software. This is achieved by attaching a trusted hardware component to each PE: the *Data Transfer Unit* (DTU). It isolates the PE from the rest of the system by controlling access to the Network on Chip (NoC).

Communication from PE to PE is implemented via message passing through established channels between their DTUs. There is no other way for software running on a PE to access PE-external resources. In a usual setup of M^3, there will be a kernel, some system services and applications. Figure 2.2 shows a typical scene of M^3.

M^3 assigns applications to PEs exclusively. Hence, an application owns its PEs until the kernel removes it. If there are no PEs left, no application can be run until another one is removed. As already mentioned in Chapter 1, there are cases where this behavior is not desired. One example are applications that are idle most of the time. Therefore, they occupy valuable computational power and preventing the system from utilizing it. The goal of this thesis is an approach to cope with this cases by adding support for time-multiplexing among different applications on a single PE.

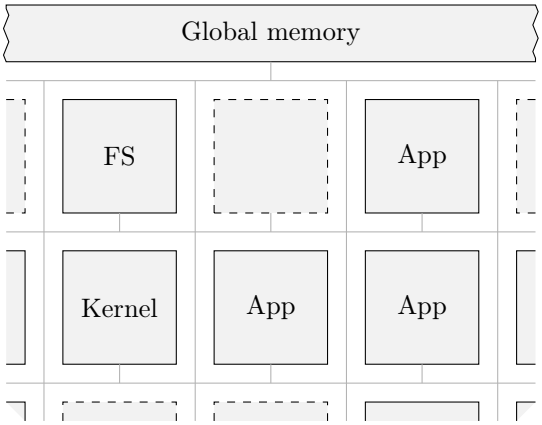

Figure 2.2: A typical setup of M^3. There is a core running the kernel, one that executes a filesystem service (FS) and user-applications on other cores (App). Global memory is visible to all cores. Idle cores are depicted with a dashed border.

M^3 is a microkernel-based operating system. Therefore, the kernel only implements the absolutely necessary features. In the case of M^3, this includes the control of communication between applications, restriction of access to system resources and the management of global memory. However, applications are communicating directly once permissions have been granted in order to increase performance and reduce kernel load. This is possible due to the trusted DTU hardware component at each PE, which the kernel controls. There may also be more than one kernel around allowing for distribution of load across multiple kernel cores[1]. The main components of the M^3 kernel are:

- The *Syscall Handler*, which implements the operating system's syscalls.

[1] A single instance of the kernel may quickly become a bottleneck. Just imagine 10 concurrent applications doing syscalls at 10% of the time. Thus, the involvement of the kernel in frequent tasks has to be kept to a minimum.

- *KVPE* as the abstraction of processing elements in the system. It allows for loading, executing and controlling applications. It also provides runtime information like capabilities.
- The *PEManager* that keeps track of available processing elements and the applications running inside them.
- The management of receive-buffers: If an application wants to attach a receive buffer to a receiving channel of its DTU, it has to ask the kernel for permissions. This way the kernel enforces some security constraints.
- A name-service for system services. Applications can register themselves as *services* with the appropriate syscall.
- The *MainMemory* manager.

When M^3 is started, each PE has the ability to become a privileged kernel-PE in the system. As soon as the kernel has been initialized, it flips a special bit on all other DTUs which turns them into unprivileged PEs.

Like with other microkernel-based operating systems, services provide some kind of abstraction to other applications. Examples are the filesystem service or device drivers. Services register themselves with the kernel. Processes can then request access to a given service from the kernel. Once the kernel has accepted the request, it will establish a channel between the service and the application to allow direct data transfers. M^3 uses a concept called *capabilities* as a mean of access control. The kernel may grant capabilities to applications in order to allow them to access memory or transfer data.

Functionality of the operating system that does not necessarily need to remain in the kernel is provided by M^3's system-library *libm3*. Each application that is linked to this library gets the necessary abstractions for accessing features of the operating system like executing syscalls, accessing system resources or passing messages.

Global memory is managed by the kernel. As the name states, it is visible to all PEs in the system. Processes can request memory trough the kernel's syscall interface. An application can allocate global memory and is awarded a memory capability if the allocation succeeds. This capability allows for direct access to the memory without the need for additional kernel interaction.

2.3 Data Transfer Unit

M^3 strongly relies on a special hardware component called the *Data Transfer Unit* (DTU). Each PE has its own DTU attached to it. Its configuration controls the PE's

access to the Network-on-Chip (NoC). Software that runs on a PE can use the DTU to transfer data between the PE's local memory and other resources of the system.

A PE that wants to transfer or receive data requires a configured *endpoint* (EP) for the desired channel on its DTU. Since only the kernel is able to write the endpoint configuration registers of the DTU, it controls all data transfers between PEs in the system. With the kernel-controlled DTU as the only means for software running on a PE to access resources outside of itself, isolation between applications at NoC-level is achieved. Thus, the DTU decouples isolation of applications from availability of appropriate processor features.

Applications just ask the kernel for permissions to open a channel by executing a syscall. If the kernel grants access, it configures the channel endpoints of the respective application-PE's DTUs. If the channel is established, data may be transferred directly from *source* EP to *drain* EP. The kernel may also revoke permissions at any time.

Data that is received via the DTU is stored in a receive buffer. Applications attach buffers to EPs of their local DTU with a syscall, passing the buffer's address, size and the message size as arguments. If the kernel receives the syscall and allows the requested receive buffer to be attached, it writes the buffer parameters to the respective registers at the requesting PE's DTU. The buffer is organized by the DTU as a ring buffer with a distinct read- and write-position. It allows for parallel reading of received data while already storing the next packets in the buffer. Figure 2.3 visualizes the transfer of data between PEs using their DTUs.

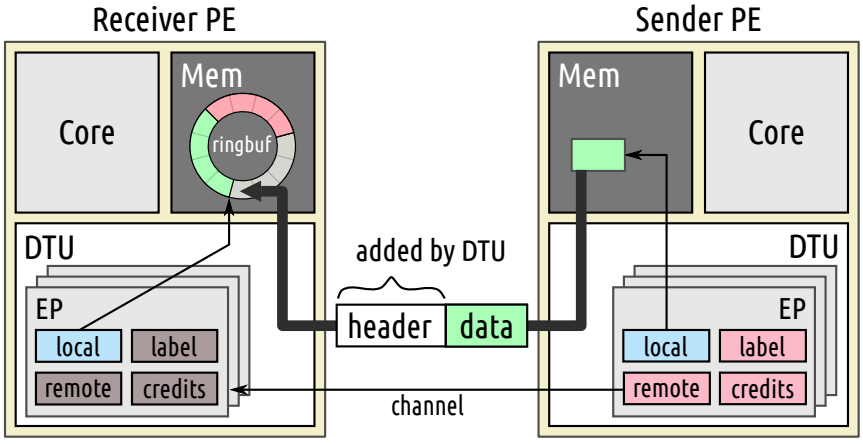

Figure 2.3: The Data Transfer Unit transfers data between PEs. On the sender side, the local register is used to point to the memory that should be transferred. At the receiver side, the register points to the current write-position of a ring buffer, where data is stored. The other registers have been configured by the kernel to establish the channel. When sending, the DTU attaches a header to each packet.

In order to prevent a malicious application from flooding the receive buffer, the DTU supports the concept of *credits*. Each EP gets a limited amount of credits. If it has used all its credits, it cannot send data through it until credits have been refilled by the received or the kernel.

Authenticity of received data is ensured by attaching a *label*. The content of this label is defined at the receiver side but is immutable to the sender. This concept is inspired by the inter-process communication mechanisms of the L4 Fiasco.OC [LW09] microkernel.

Additionally, the DTU provides the ability to trigger interrupts at the attached core. This can be done both from the inside and the outside of the core. It requires the core to have support for interrupts, though.

2.4 Related Work

Before we dive into the actual design of the time-multiplexing mechanism introduced in this thesis, I will briefly discuss a selection of other operating systems that aim for heterogeneous manycore platforms and show if and how they implement time-multiplexing for applications.

The multikernel approach by Baumann et al. [BBD+09] puts a monitor process on each core. A privileged CPU driver that is local to each core mediates between local hardware and core features. Tasks share a virtual address space between cores. Thread states are synchronized by monitors. Like with M^3's approach, the architecture aims to support heterogeneous cores but tries to achieve this through core-specific CPU drivers. Monitor processes run unprivileged and share state with other core's monitors. Communication is done through message passing over a local intercommunication channel.

The presented architecture allows for multiple tasks on each core. Time-multiplexing is done by the CPU-driver that schedules the local threads temporally. There is no remote controlled time-multiplexing in the proposed multikernel architecture. They always assume support for timers and interrupts.

Wentzlaff and Agarwal [WA09] propose what they call a factored operating system (fos). They describe an architecture based on a fleet of microkernels, which run on each core of a manycore platform. System services, like in M^3, are executed on dedicated cores and communication between cores is based on message passing.

Fos aims to replace the need for time-multiplexing with spatial-multiplexing. This is what M^3 in its current form implements: Each core runs a single application. The decision for striking this path is based on the assumption, that the number of cores on manycore platform will soon outperform the usual number of threads.

The Helios operating system is another design that is intended to work on top of heterogeneous processing units. Nightingale at al [NHM+09] suggest an approach that is based on what they call *satellite kernels*. These kernels run on each core and try to do as much work as possible locally. Communicating with other core's kernels is done via a message passing mechanism if necessary. By implementing software isolation, Helios does not rely on the availability of privileged mode and MMU.

Helios has no implementation of remote controlled time-multiplexing. Threads are scheduled locally by the satellite kernel as implemented in traditional monolithic kernels. This also means the satellite kernels of Helios require some basic hardware features like timers, interrupts and traps to handle exceptions. This prevents Helios from utilizing some types of cores that lack the required primitives like today's Graphics Processing Units (GPUs) do. However, the authors expect these features to be available on all hardware in the future.

With NIX, a case for manycore cloud computing, Ballesteros et al. [BEF+12] suggest to assign certain roles to cores of a manycore system. Some of the roles demand certain features to be available at the respective core. The main roles proposed are:

- *Time-sharing cores* that have features like timers or interrupts and allow for preemptive time-multiplexing, for example.
- *Kernel cores* which are a subset of time-sharing cores and running the operating system's kernel code.
- *Application cores* which just run a single application in non-preemptive mode like it is done with M^3. These cores do not need complex features like the others do.

There has to be at least one time-sharing core available on the platform for NIX to be applicable. The authors state that it would suffice if a manycore platform of n cores provides $\sqrt{n}$ cores capable of becoming a time-sharing core. In contrast to the other approaches mentioned, the NIX project is using shared memory for inter-core communication. However, when it comes to time-multiplexing it does the same as all the other projects: It just provides support for local time-multiplexing on cores that allow for it (time-sharing cores) and no time-multiplexing at all on application cores.

3 Design

This thesis proposes how remotely controlled time-multiplexing can be added to M^3. The presented approach introduces a small piece of software, called the *Remote-Controlled Time-Multiplexer* (RCTMux) that runs on an application-PE and does the actual work of switching process contexts at that processing element. It is supported and controlled remotely by M^3's kernel running at another core. The context-switch controller and the kernel cooperate with each other based on a strict communication protocol. Below, the PE where the context-switch takes place is called *remote-PE*.

This chapter starts with a description of the main goals of my design in Section 3.1. Section 3.2 will then introduce the RCTMux software that runs at the remote-PE and does most of the context-switching while controlled by the kernel. In Section 3.3, the protocol that defines the cooperation of kernel and context-switch controller is introduced. Afterwards, the proper handling of malicious processes with the presented approach is discussed in Section 3.5. Section 3.7 completes this chapter with a list of the hardware-assumptions that have been made and their respective justification.

3.1 Goals

The goals of the proposed design are as follows:

- The **context-switching time** should be minimal[1].
- Contexts have to be strictly **isolated** from each other. A malicious program should never be able to influence another program in any way nor should it be possible to prevent proper storage and restoration of a foreign process' context.
- RCTMux should do as much work locally as possible to keep the **kernel load** at a minimum. The kernel should only be involved if it is unavoidable to achieve a desired functionality.
- The **code and binary size** for both the additions to the kernel and RCTMux should be as small as possible. The microkernel ought to only implement whats really necessary to be done by the kernel. Furthermore, since RCTMux consumes

[1] The importance of this goal is proportional to the frequency of context-switches.

some memory on the PE that is taken from memory otherwise available to applications, it has to be small es well.

- Additional **hardware-requirements** should be avoided. However, some functionality like proper isolation from processes are only feasible with appropriate support from the hardware. Hence, the guideline for my design is as follows: Only rely on hardware-assumptions if really necessary. If new requirements have to be added, prefer those broadly available[2].

3.2 The Remote-Controlled Time-Multiplexer

RCTMux is a small piece of software for remote-controlled time-multiplexing. It is loaded from *read-only memory* (ROM) during processor initialization and processor-reset events, respectively. When restored from ROM, RCTMux is fully trusted by the kernel. Since the program is loaded to the PE's memory, an application running on that PE may be able to modify or overwrite it, if there is no hardware support for protection available and activated at the core. To make sure that RCTMux is not modified by a malicious application, the kernel can reset the core remotely, which implicitly restores RCTMux from ROM. This mechanism allows for remote-controlled time-multiplexing and isolation of switched processes on cores that do not provide any support for protecting software like distinct privileged and unprivileged modes or memory protection. In the proposed design, a malicious application is only able to harm itself during time-multiplexing.

There are two events that trigger an implicit action of RCTMux: *processor-reset* and *hardware-interrupt*. At processor-reset, RCTMux is restored from a small ROM at the PE. The program initializes itself and installs an interrupt handler to respond to hardware interrupts. Afterwards, it reads a read-only register of the DTU (see Section 3.4 for details) to check if the kernel has requested the restoration of a suspended process. If this is the case, RCTMux contacts the kernel to do the restoration according to the context-switch protocol as described in Section 3.3. If there is nothing to restore, the core is put into idle mode[3]. When a hardware-interrupt occurs and the interrupt-handler that has been installed by RCTMux is intact, it checks the context-switch flags register of the DTU to decide whether or not this interrupt is intended to start a context-switch. If not, the interrupt-handler returns. If a context-switch is requested, RCTMux proceeds according to the context-switch protocol.

The process running at the remote-PE may also install a dispatch routine to use the interrupt for other purposes, if desired. This way we can re-use existing interrupts. To

[2] Keeping assumptions to hardware low conforms to M^3's goal of fully utilizing very heterogeneous hardware-platforms.

[3] Idle mode is usually a low power state. The processor waits for an interrupt or reset from the kernel.

have its context properly saved in case of a context-switch, the application should take care of not destroying RCTMux' handler, though.

3.3 The Context-switch Protocol

A context-switch consists of a sequence of consecutive steps. These steps are executed by either the kernel or RCTMux at the remote-PE. When one side does its part of the job, the other one has to wait for completion before proceeding with a subsequent step. Thus, their co-operation has to be properly coordinated by agreeing on an appropriate communication protocol. In this section, the basic protocol is introduced. It is then extended in Section 3.5 to cope with malicious applications at the expense of an additional hardware assumption. The basic context-switch protocol can be divided into four logical phases, where two of them are optional[4]. Figure 3.1 shows all phases of the context-switch protocol.

1. In the *mandatory* **initialization** phase, the kernel sets some flags in a special register at the remote DTU to configure the behavior of RCTMux.
2. During the *optional* **storage** phase, the currently running process at the remote-PE is interrupted preemptively. Its context is then stored to memory.
3. In the *mandatory* **reset** phase, the remote core is reset. This will also trigger a reinstall of RCTMux from read-only memory (ROM) at the remote-PE.
4. Finally, the *optional* **restoration** phase may happen. The saved context of a process is restored at the remote-PE. On success, the process is resumed.

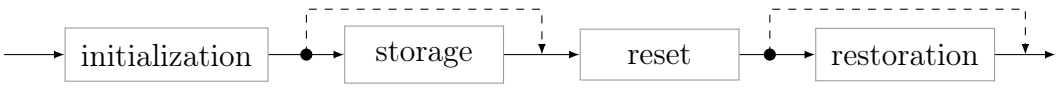

Figure 3.1: The phases of the context-switch protocol. Phases two and four are optional.

3.3.1 Phase 1: Initialization

The initialization phase allows the kernel to set everything up for an upcoming context-switch. This is done by setting some flags in the context-switch flags register at the remote-PE's DTU. The layout of this register is presented in Section 3.4. The flags control if the optional restoration phase has to be performed. There is also a flag that

[4]By making them optional, the protocol is flexible enough to handle additional functionalities such as process migration. They will be mentioned in Chapter 6.

indicates if a context-switch has actually been requested. The latter allows for recycling an interrupt, if no dedicated interrupt is available. When the interrupt occurs, a simple dispatch routine can check for the flag and decide what action to take. If the storage phase is not required, the kernel skips it and directly proceeds to the reset phase.

3.3.2 Phase 2: Storage

During the storage phase, the context of the process currently running at the remote-PE is stored to memory. While its context is stored, the process is called *suspending* below. Accordingly, when the storage phase is complete, it is called *suspended*.

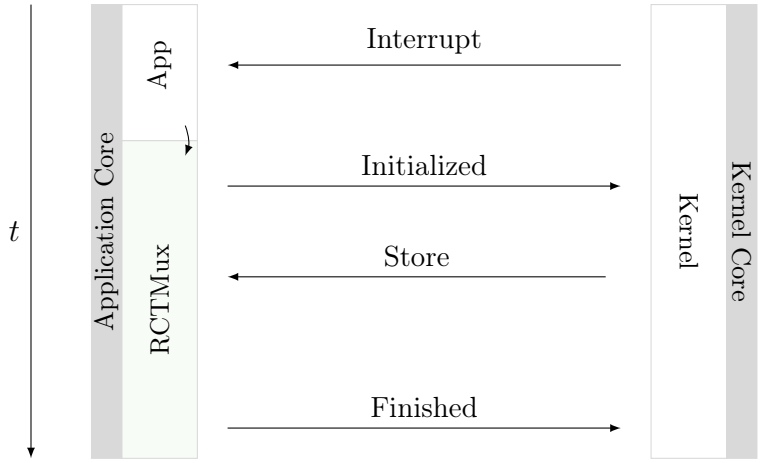

Figure 3.2: Communication between kernel- and remote-PE during the storage phase

The communication between kernel- and remote-PE in the storage phase is depicted in figure 3.2. When entering this phase, the DTU of the remote-PE is still configured for the suspending process. Therefore, the kernel has to reset the endpoints and attach the memory for storing. We also need to invalidate all sending endpoints at other DTUs to avoid inconsistencies. The actual storing is done by RCTMux, which informs the kernel when done. Finally, the kernel has to clean everything up. All of these goals are achieved by the following five steps:

1. The kernel initiates the storage phase by invalidating all sender endpoints in the system that are configured to send to the suspending process. This will force processes that want to send data to the suspending process to contact the kernel, which initiates a context-switch to restore the suspended process.

   Subsequently, the kernel sends an interrupt request to the remote-PE. This is done by writing a special register at the remote-PE's DTU. The interrupt forces the core to stop execution of the current process and jump into RCTMux' interrupt handler

routine. Since this will only take a few cycles in most cases, the kernel polls an implementation-specific memory location at the remote-PE's local memory after sending the interrupt request.

2. If intact, the interrupt handler installed by RCTMux moves the contents of the CPU registers to a safe place in memory. It then waits for the DTU to finish all active transfers. Finally, the kernel is informed by inverting the value of the memory location the kernel is polling and then polls the same location to wait for the kernel to complete its turn.

3. The kernel now resets all endpoints at the remote-PE's DTU and sets the internal state of the application to *suspended*.

   Since the storage of a process' context usually takes more than just a few cycles, polling would block the kernel for too long. Therefore, a communication channel to the kernel is configured at the remote-PE. This channel will allow RCTMux to indicate completion. Finally, an endpoint is set up to establish a channel to the global memory for storing the suspending process' context.

   When finished, the kernel inverts the value at the memory location that RCTMux is polling.

4. At this point, everything is prepared to actually store the context of the current process at the remote-PE to global memory. Therefore, RCTMux writes the context-data to the memory channel that has been attached by the kernel. Upon completion, it sends a *finished* message to the kernel through the communication channel that has been created in the previous step. Afterwards, RCTMux puts the core into idle mode.

5. The kernel completes the storage phase by resetting all endpoints at the remote-PE. As a side-effect, this also detaches the global memory where the context has been stored.

3.3.3 Phase 3: Reset

The reset phase consists of only one action. That is the kernel sending a reset request to the remote-PE's DTU to force a local hardware-reset. The reset triggers a reinstall of the RCTMux-software into the local memory of the remote-PE. This ensures the integrity of the trusted RCTMux software.

If the restoration of a process is necessary, the kernel polls on a special memory location at the remote-PE's local memory. Its value is inverted by RCTMux as soon as it has been restored from ROM and initialized. Since this only takes a few cycles, polling is acceptable. If the restoration phase is not required, the kernel resumes its normal

operation and the protocol is finished from the kernel's point of view. In this case, RCTMux puts the core into idle mode and waits for the kernel to wake it up again.

3.3.4 Phase 4: Restoration

The restoration phase contains the necessary steps to restore a suspended process' context and resume its execution where it has been interrupted. RCTMux can initiate this phase only when it is run for the first time (e.g. after a hardware-reset). The communication between kernel and remote-PE is depicted in figure 3.3. It consists of the following five steps:

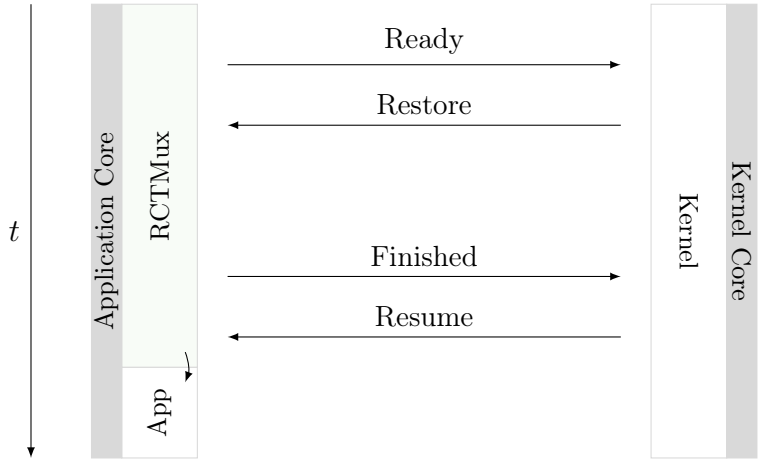

Figure 3.3: PE communication during the restoration phase

1. RCTMux initiates the restoration phase by inverting the value of the memory location polled by the kernel. This tells the kernel that RCTMux is ready. Afterwards, RCTMux starts polling the same memory location to wait for the kernel to complete its turn.

2. The kernel configures the endpoint for restoring the respective process' context. Additionally, a communication-channel is set up for RCTMux to send its *finish* message. The kernel inverts the value at the memory location that RCTMux is polling and resumes to normal operation.

3. RCTMux starts the actual restoration of the process' context. It loads all memory data to the local memory and restores the CPU state. It then sends the kernel a *finished* message through the established message channel to indicate completion of the context restoration. It then polls the context-switch flags register of the DTU and waits for its reset by the kernel.

4. The kernel finally resets the endpoints at the remote-PE's DTU and restores the endpoint-configuration of the suspended process. When done, it resets the context-switch flags register at the remote-PE's DTU.

5. The process and its endpoints have now been fully restored. RCTMux completes the restoration phase by passing control to the resuming process.

3.4 The Context-switch Flags Register

The context-switch protocol described in the previous section makes use of a special register of the DTU. This register must only be writable by the kernel. It contains flags that control the behavior of RCTMux.

The register contains two flags at the moment: *switch* and *restore*. The switch flag indicates that a context-switch is requested by the kernel. When an interrupt occurs, it can be checked by a dispatch routine to determine if the interrupt is caused by the kernel to initiate a context-switch or if it has another cause. The restore flag is set if the optional restoration phase of the context-switch protocol is required. If the flag is not set, RCTMux will simply skip the restoration phase.

A similar flag for the storage phase is not necessary. If the optional storage phase is not required, the kernel just proceeds directly from the initialization phase to the reset phase. Hence, no cooperation of RCTMux is required in this case.

3.5 Protection from Malicious Processes

When a context-switch happens, two processes are involved at the remote-PE. The *suspending* process whose context is stored to memory and the *suspended* process whose context is restored from memory. For proper application isolation, both the suspending and suspended process must not be able to influence the storage or restoration phase of each other.

The introduced context-switch protocol assumes that all participants co-operate in an honest way. Unfortunately, this may not always be the case. Since the isolation concept of M^3 is located at NoC-level, processes may have full control over the remote-PE including its memory. An evil piece of code may overwrite RCTMux' interrupt handler with its own version or just ignore any context-switch request from the kernel. Thus, malicious software has to be taken into account for a proper design of remote-controlled time-multiplexing.

If we assume time support in the kernel, we can add a simple constraint to the protocol that prevents malicious software from mounting denial of service attacks by just ignoring the interrupt from the kernel. With the plain protocol as presented above, this would lead to a infinitely polling kernel in the storage phase (see Section 3.3 for details). To prevent this situation, we add an appropriate time limit δt to each critical step as depicted in figure 3.4. These time limits are managed and monitored by the kernel.

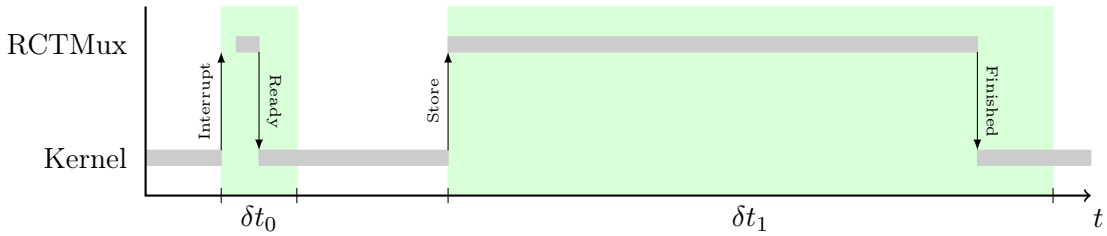

Figure 3.4: The storage phase of the context-switch protocol with two time limits δt_0 and δt_1 added. The gray bars represent the work that each of the participants do according to the protocol. δt_0 limits the time RCTMux has to initialize itself and respond to the kernel whereas δt_1 limits the time for the actual storage of a process' context.

Whenever the time limit is reached without a response from RCTMux, the currently running process at the remote-PE is killed and the kernel proceeds to the next phase, which will force a hardware reset of the PE's core and restores RCTMux from ROM. Even if the evil process conforms to the context-switch protocol and tricks the kernel, it can only influence the storage phase for its own context. In the worst case, it blocks the context-switch as long as possible without reaching the time-limit.

Furthermore, both processes must not be allowed to obtain any information about the respective other process' context. The context-information that must not be disclosed includes the memory segments, register contents and the DTU's endpoint configuration of the attacked process. Therefore, RCTMux has to destroy all confident information before the next process takes control of the core and memory. This can be done either when RCTMux is initialized after a processor reset or at the beginning of the restoration phase (see Section 3.3 for details). The respective memory segments and register contents have to be filled with with random data or zeros locally by RCTMux.

Finally, a process must never be able to get access to the storage of another process' context. This is already ensured by the context-switch protocol, since the kernel detaches the storage at the end of the storage or restoration phase, respectively.

3.6 Switching Context on Demand

In some scenarios it may be reasonable to switch contexts on demand. For example, a service that is not used very often has been suspended. At some point in time, a process wants to use the suspended service and creates a direct connection (*session*) by issuing the corresponding syscall. The kernel, which knows about the *suspended*-state of the service's process, initiates a context-switch to restore it. When the service has been restored successfully, the kernel returns from the syscall and the requesting process can use the session to contact the service as usual.

Processes need to know if the destination process of one of their communication channels has been suspended. This is done by setting their DTU's *permissions*-register of the respective endpoint to zero. The code in libm3 can check the register before executing a send or write operation and issue an *activate* syscall, if necessary. The syscall needs to be modified to initiate a context-switch for suspended targets.

When doing context-switches on demand, we have to deal with the following situations:

- A service should not be suspended until it has registered itself at the kernel. Otherwise, the kernel would not know about its existence (in the service-name directory) and no request could ever be made to wake it up again.
- If a client-request to a service is interrupted by a context-switch, the client may wait for the service's response indefinitely, if no other request wakes up the service.

A solution to resolve such situations is the use of *delayed preemption*. Like with the time limits introduced in Section 3.5 that cope with malicious software, a process may delay its suspension for a limited time by blocking the interruption in critical sections of its code. If it does not unblock until the deadline has been reached, the process is killed.

With available processor support for timers and timer interrupts, an easier approach can be applied: the kernel periodically interrupts all PEs that are capable of time-multiplexing. This is similar to traditional time-multiplexing but adds much workload to the kernel. The load even increases with a growing number of time-multiplexed PEs.

3.7 Hardware Assumptions

Since one of the main design goals is to keep hardware requirements at a minimum, each additional assumption has to be justified. This section shortly discusses the benefits of the additional requirements that have been proposed: *hardware interrupts*, *hardware reset* and *timers* along with *timer interrupts*.

Available *hardware interrupts* at time-multiplexed PEs allow for transparent and preemptive context-switching. Without them, the only way to initiate a remote-controlled context-switch would be message passing. The process that is running at the remote-PE would have to contact the kernel to initiate a context-switch or respond to received context-switch requests from the kernel. It would need to corporately interrupt itself and pass the control to RCTMux or implement its own version of the context-switch protocol. Every application would thus need extra support code for time-multiplexing. The benefits of hardware interrupts for initiating a context-switch make them a reasonable assumption. Furthermore, a register for triggering interrupts at remote PEs is already available in the DTU.

The absence or availability of hardware interrupts only affects the storage phase of the context-switch protocol. The initiation of the restoration phase does work without them. However, the possibility to trigger *hardware resets* remotely is essential. At hardware reset, the trusted code of RCTMux is restored from ROM and loaded into memory. RCTMux boots up, reads the context-switch flags register of the DTU and contacts the kernel, if a restoration is necessary. Without kernel controlled hardware-reset, there would be no way to protect RCTMux from malicious modifications besides adding even more hardware features to achieve protection. The hardware reset feature of the PE's processor is inevitable for isolating applications during context-switch.

For full protection against malicious applications, time limits have been added to the protocol in Section 3.5. They require *timers* and *timer interrupts* to be available at the kernel PE. Without time limits, the context-switch protocol will work just fine as long as all participants act in a honest way. This allows for time-multiplexing even on platforms where these features are not available at all. Although, we have to face the drawbacks that motivated the introduction of time limits in the first place there. With timers available, we can achieve the design goal of proper isolation among arbitrary processes, which is a great benefit that clearly justifies this additional requirement.

4 Implementation

To proof and evaluate the feasibility of the presented approach for remote-controlled time-multiplexing in M^3, I implemented the relevant steps for context-switching according to the context-switch protocol described in Section 3.3. Some details in the presented design require modifications to the DTU and core setup that are not yet available. These details have thus been worked around or skipped here. Chapter 6 discusses future work including the necessary modifications.

The context-switch protocol as described in Section 3.3 cannot be fully implemented with the current version of the DTU for the following reasons: There is no remotely controllable hardware-reset and no ROM available. Therefore, I implemented the Initialization, Storage and Restoration phase only. The reset phase is not covered by my implementation yet. It is subject to future work, though. Furthermore, the flags register (see Section 3.4) is not yet available. I worked around this by using a special location in memory instead. This has the drawback of the flags register being writable by the PE which the designed approach forbids to disallow manipulation by malicious software. However, my implementation suffices for evaluating the feasibility of remote controlled time-multiplexing with M^3.

The rest of this chapter is structured as follows: Section 4.1 introduces the target platform my implementation is based on. Section 4.3 follows with a discussion of my additions to the kernel. Section 4.4 continues with an outline of my RCTMux-implementation. Finally, Section 4.5 describes the elements of a process' context and how context-switches are triggered in my setup.

4.1 Target Platform and Setup

The target platform for my proof-of-concept implementation is a cycle accurate simulator of an extended version of the tomahawk2 platform [AMN+14], a multiprocessor-system on chip (MPSoC), that connects all its processors through a network on chip (NoC). The simulator is based on the Cadence Xtensa SystemC [Cad] framework and has been modified to include the DTU. For my implementation, I used a simulator setup of 18 Xtensa PEs and a DRAM memory-unit. Interesting elements of the used PE-configuration are listed in Table 4.1.

Feature	Configuration
Memory protection / MMU	region protection only
Number of general purpose registers	32
Data RAM / Instruction RAM	64k / 64k
Instruction/Data Caches	None
Timer	None
Level1 Interrupts	4

Table 4.1: Configuration of the simulated PEs

4.2 Layout of the Context-storage

A small scratchpad memory is the only memory locally accessible at PEs on the target platform. It can be read and modified without restriction by any application that runs on the PE. This requires the context of applications to be stored outside of the PE for space and security reasons, e.g. in global memory. It is thus necessary to copy the whole program, including its heap and stack, at every context switch. Instead of copying the whole available memory every time, only the parts of the memory that are actually used should be copied for performance reasons. To do this, information about size and position of the individual data segments of the application is required when switching the context. I solved this by adding a structure named *AppLayout* to libm3. Its declaration is shown in listing 4.1.

```
struct AppLayout {
    word_t reset_start;
    word_t reset_size;
    word_t text_start;
    word_t text_size;
    word_t data_start;
    word_t data_size;
    word_t stack_top;
    word_t _unused;
} PACKED;
```

Listing 4.1: The AppLayout structure

The saved elements are the respective address and size of reset vector, text segment, data segment (including heap) and the start address of the stack. The stack pointer is not saved this way. It can easily be collected when necessary. The whole structure is placed statically within the scratchpad memory that also contains other runtime data like the application's entry point or its arguments. Most of the elements are initialized

by code in libm3 during startup of the application and do require any further update. However, the data size needs to be updated when the size of the heap changes. This is implemented as an extension to the heap management code in libm3.

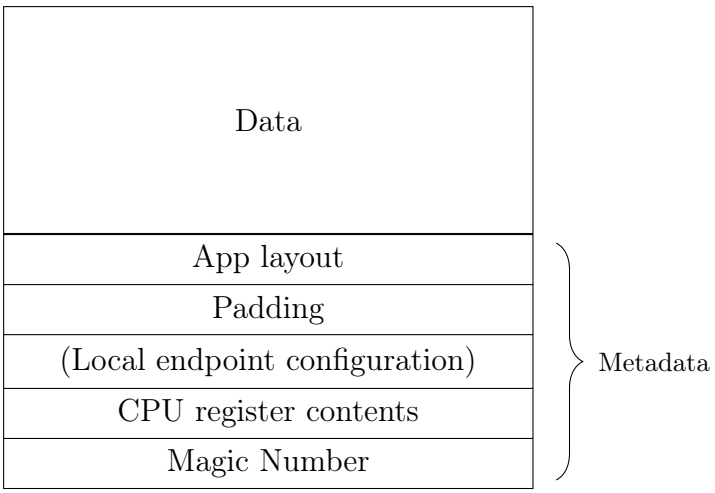

Figure 4.1: The layout of the context storage

I decided to organize the storage memory as depicted in figure 4.1. The magic number is just a constant value at the moment. It is checked when the memory is read in the restoration phase of the context-switch protocol. If it does not match, the restoration is stopped[1]. The stored data also contains the application's local endpoint configuration of the DTU. This includes address and size of local receive buffers along with their configured message size. It is a workaround that is necessary with the current implementation of the DTU, where the related registers cannot be read from a remote PE. Additionally, it is impossible to get the current write and read positions of receive buffers out of the DTU. With `struct AppLayout` at hand, the actual application data can be stored tightly packed on top of the meta data.

4.3 Kernel Additions

I extended the kernel with the necessary functionality for remotely controlling time-multiplexing. When creating a new *KVPE*, the `suspendable` flag can be set to enable time-multiplexing. The kernel will assign applications that have the flag set to one of the cores where RCTMux is available.

[1]The magic number may later be replaced by a checksum of the stored data.

The storage and restoration of the remote-PE's EPs is implemented by a workaround. The DTU in its current form does not allow to read the necessary data from remote. Therefore, the kernel has to iterate over the application's capabilities to find activated endpoints. This is an expensive operation that takes a lot of cycles (see Chapter 5).

M^3 without time-multiplexing assumes that one PE belongs to only one application (or none) at a certain point in time. These assumption can be found in many places of its code. For example, the core id has been re-used as the application's pid. This issue has been fixed by assigning continuous process-IDs to applications.

A more challenging example is the current implementation of message capabilities. These capabilities control the access to memory transfers, thus the communication among PEs. Currently, they restrict access per PE. When time-multiplexing is introduced, a PE no longer runs only one application. Instead, multiple applications can be bound to the same PE at the same time. An application can then create a new VPE with the suspendable-flag set and the kernel will assign this VPE to a time-multiplexed PE. Now, the application that initially created the new VPE has got a VPE-capability, which grants full access to the VPE's PE. This access permissions are independent of what VPE is actually running there. To fix this issue, well thought changes are necessary as they may affect security. This changes are subject to future work on this topic.

4.4 RCTMux

RCTMux (Remote-Controlled Time-Multiplexer) is a small program that deals with context-switch requests from the kernel. To keep the size of RCTMux small, it is not linked against libm3. Instead, the required functionality to transfer data via the DTU is re-implemented. The provided proof-of-concept implementation of RCTMux is invoked on interrupt request. The interrupt handler saves the current processor state to a reserved memory location and sets up a suitable environment for calling C++ code. The C++ code implements the storage and restoration phase of the context-switch protocol. When the interrupt-handler returns, the processor state is restored and control is given to the restored application, if requested. If something went wrong, the processor is put into idle mode and waits for the next interrupt to occur.

The Xtensa cores make use of a feature called windowed register mode [Ten13] that requires some special handling when saving the contents of the cpu registers. A subset of the large physical register file is made available through a *window frame*. The registers of a window frame appear to the software as the first physical registers. Whenever a subroutine call is made with the appropriate instruction, the window frame is rotated. This technique saves a lot of cycles: Often register contents can just be kept in the processor. However, at some point, all registers may be in use and another rotation of

the window frame would lead to an overflow. The processor throws an overflow exception in this situation. The overflow exception-handler puts the contents of a register window frame to the stack. This frees up space for a new window frame. When the stored window frame is accessed again, an underflow exception is thrown and the register contents are restored by the respective exception-handler. To take this behavior into account when implementing context-switching for this processors, RCTMux uses the macro `xthal_window_spill_nw` provided by the Xtensa development tools to spill the content of all register windows out on the stack. The stack pointer is decremented[2] appropriately to include this data in the storage phase. When restoring the process, the windows frames are automatically read from the stack on demand by window underflow exception-handlers.

The RCTMux code contains a workaround for reading the receive buffer configuration of the DTU's set of endpoints. This is actually intended to be done by the kernel. However, the DTU does not yet allow to read the respective registers from outside of the PE. Therefore, the registers are read locally and their contents are stored along with other context data.

4.5 Switching Contexts

The simulated PEs do not provide any timer or timer interrupt features which makes a proper implementation of periodically time-multiplexing impossible. To test the feasibility of the presented approach, I implemented a simple `tmuxswitch`-syscall that instructs the kernel to switch to the next process. Therefore, the context-switches are non-preemptive.

Preemptive context-switching in general is not possible with the current version of the DTU since there is no possibility to retrieve or store the write and read position of a receive-buffer. Thus, a receive buffer may contain messages that have not yet been read by the application when a context-switch happens. A message may also be read but not yet acknowledged. After restoring the endpoint and receive-buffer configuration when the application is resumed, the write and read position of a buffer are likely to have changed. The received messages are therefore lost or the data at the respective positions may be interpreted wrongly. My implementation is therefore only capable of reliably switching contexts of applications that do not use any receive buffers when the context is switched. Context-switches of applications that do not meet this requirement may work or they may not work. With an improved version of the DTU that allows for accessing all state parameters of receive-buffers, this limitation can be dropped.

[2] The stack grows downwards.

5 Evaluation

The evaluation of the presented design according to my implementation will answer the following questions:

- How many changes and additions have been made to the code-base of M^3?
- How much memory does the RCTMux binary occupy?
- How is the performance of remote-controlled context-switches compared to Linux?
- How does the kernel's load affect the performance of context-switches?

5.1 Lines of Code

I evaluated the *Source Lines of Code* (SLOC) of my implementation using the software *sloccount* [Whe]. The results are listed in Table 5.1. For clarity, I grouped the measurements by logical category and language. The categories I chose are:

- *RCTMux* contains the code of my RCTMux implementation without the architectural specific code (`apps/rctmux/*.[ch]*`).
- *RCTMux (architectural specific)* includes only the architectural specific code of RCTMux like saving CPU-registers and installing the interrupt handler (`apps/rctmux/arch/`).
- *Kernel context switcher* contain the code of the *ContextSwitcher* class that I added to the kernel (`apps/kernel/ContextSwitcher.*`).
- *Kernel other* is for all other changes I made to the kernel code of M^3(`apps/kernel/`).
- *libm3* is the group for my modifications to libm3 (`libs/m3/`, `include/m3/`).
- *Test utilities* contains the code of the utility programs that I wrote for benchmarks and evaluation of the implementation (`apps/rctmux/test-utils/`).

	SLOC	
Category	**C++**	**Assembler**
RCTMux	191	0
RCTMux (architecture specific)	51	157
Kernel context switcher	148	0
Kernel other	129	0
libm3	29	0
Test utilities	80	0
Total	628	157

Table 5.1: Added Source Lines of Code grouped by logical category.

5.2 Binary Size

One of my design goals is to keep code-size and binary-size of RCTMux as small as possible. Therefore, I measured the resulting size of the binary using the GNU *size* utility of the GCC tool-chain. I also compared the result to the binary size of the *idle* program of M^3. Idle is a small placeholder that turns a core into a low power state until a more useful application is loaded to the PE. It is deployed to all unused PEs at bootup. If only this functionality of idle is desired, RCTMux can easily be used as a replacement for idle while adding support for remote controlled time-multiplexing. The results of the size-measurement are listed in Table 5.2.

	size (byte)			
Application	**text**	**data**	**bss**	**total**
rctmux	2296	20	168	2484
idle	1518	328	0	1846
Difference	+778	−308	+168	+638

Table 5.2: Results of the evaluation of binary sizes.

A binary size of around 2 kilobytes is acceptable. If used as a replacement for idle, it adds only 638 bytes of occupied memory.

5.3 Context-switch Performance

Another important design goal of the presented approach of remote controlled time-multiplexing is an acceptable context-switch performance. To evaluate the performance, I measured the cycle count for interesting sections of my implementation.

The cycle-accurate simulator, which my implementation is based on, does not provide a cycle counter that can be read by the simulated program. Therefore, the profiling code writes a special register of the DTU and the DTU-code creates a log file entry that contains the measures cycle count. Each cycle counter that is created this way can be assigned an id to be distinguishable from other cycle counters. The code sample in Listing 5.1 illustrates how a cycle counter is applied to a section of interest.

```
Profile::start(1);
restore_endpoints(_currtmuxvpe);
Profile::stop(1);
```

Listing 5.1: Adding a cycle counter for profiling. The section of interest in this example is `restore_endpoints()`. The counter is assigned an id of 1.

I used the small script `bench.sh` contained in the `tools` directory of M³'s source tree to extract the cycle counters from the simulator log file. I also wrote a small tool[1] that creates a comma separated list of values from the data for plotting. Assuming a normal distribution, this small python script also calculates the mean ($\overline{x}$) and standard deviation (σ) for n measured samples according to equations 5.1 and 5.2, respectively.

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad (5.1)$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \qquad (5.2)$$

To make the results comparable, I split the whole process of remote controlled context-switching into logical groups and measured the cycles they consume separately. The groups are[2]:

0. The overall cycle count for a complete context-switch
1. Endpoint storage and restoration (workaround)
2. Storage and restoration (writing context to global memory)

[1] The tool is named `create-data.py` and can be found in `apps/rctmux/benchmark-data/`.

[2] The numbers of the list items above reflect the cycle counter IDs in the recorded benchmark data files located in `apps/rctmux/benchmark-data/`.

3. Storage and restoration of local endpoint configuration (workaround)

The evaluation of context-switch performance has been done for three different experimental setups, recording the cycle count of 50 consecutive context-switches each. In the first arrangement (*idle kernel*), the kernel had no other requests to handle while switching context. In the second round (*busy kernel*), I added a little application that permanently allocates and frees global memory via syscalls to produce additional load at the kernel. In the third round (*very busy kernel*), five of those load producing applications were running while the cycles for context-switches have been recorded. The results are listed in Table 5.3.

	Cycles $\pm \lceil\sigma\rceil$		
Group	**Idle kernel**	**Busy kernel**	**Very busy kernel**
Overall	14543 ± 53	20571 ± 34	42610 ± 38
Context transfers	7739 ± 31	7733 ± 28	8886 ± 662
EP storage	3769 ± 6	3769 ± 6	3740 ± 7
Local EP storage	307 ± 24	307 ± 24	309 ± 31

Table 5.3: Evaluation results of context-switch performance.

The number of cycles a context-switch takes is proportional to the kernel load. Figure 5.1 shows plots of the cycle count for the three experimental setups with no to heavy kernel load as described above. The plot on the left hand side also shows that the cycle count of context-switches is stable over time.

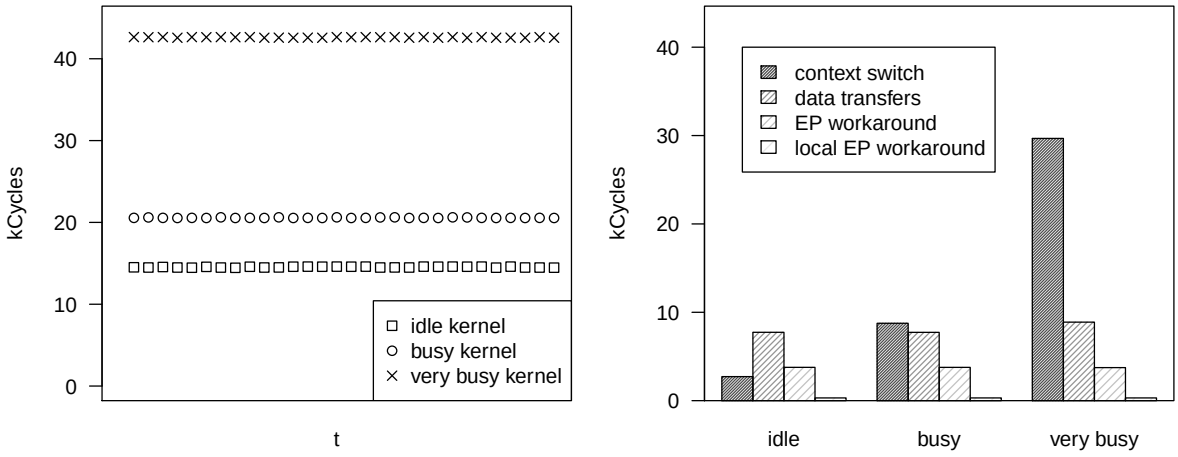

Figure 5.1: Context-switch performance in relation to kernel load.

To compare these results with context-switches of Linux running on a processor with caches, the cycle count of Linux' context-switches has been measured by a small program that repeatedly calls `sched_yield()`. The measured mean cycle count is 2304±559. The comparison with the remote controlled approach is depicted in Figure 5.2. With remote controlled time-multiplexing on PEs that only provide a simple scratchpad memory, the whole application and its data has to be transferred from and to global memory, respectively. The DTU is currently capable of transferring 8 bytes per cycle. With a program of e.g. $10kb$ of occupied memory, the amount of cycles to transfer it via the DTU is 1280. For storage and restoration of applications of this size, the process takes 2560 cycles. This makes the presented approach less comparable to Linux context-switches. Thus, the storage and restoration is colored differently.

Due to the missing DTU feature for reading and writing the endpoint configuration, the storage and restoration of the DTU endpoint configuration has been implemented by workarounds. These workarounds have to save the configuration of each EP separately and are very expensive in terms of cycles. With an advanced version of the DTU, they can be replaced by a simple read and write operation that will take approximately 5 cycles per endpoint (assuming registers of 64 bit width). With this in mind, the Linux context-switch and the remote controlled context-switch without workarounds and ignored memory transfers lie in the same order of magnitude.

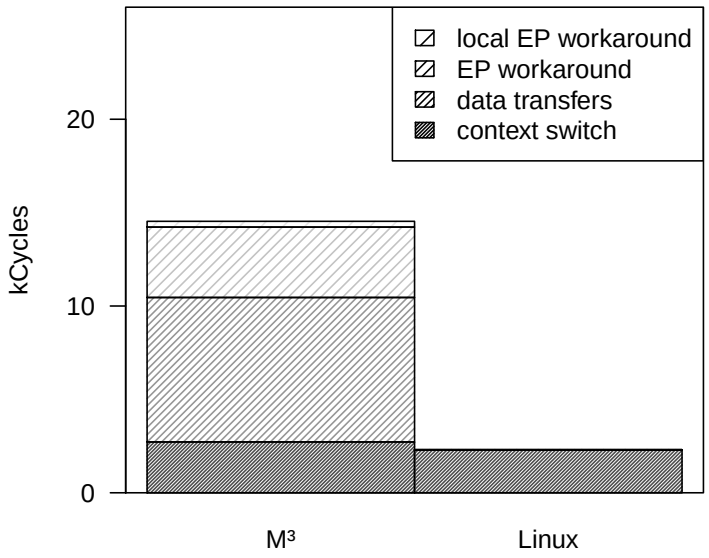

Figure 5.2: Context-switch performance compared to Linux.

6 Conclusion and Future Work

The design of a protocol for remote-controlled time-multiplexing as presented in Chapter 3 meets the desired design-goals well. It isolates arbitrary applications from each other, while keeping the kernel load at a minimum. The actual work for switching contexts is done by the small RCTMux program at the application-PE. With a code-size of around 2.5 kilobyte (Section 5.2) its memory overhead is acceptable. The whole process of remote-controlled time-multiplexing in its basic form works without any further assumptions to the hardware. However, for handling malicious applications properly, timer support is inevitable (Section 3.5). Its benefits clearly outweigh this additional hardware requirement. Furthermore, a very specialized core is not of interest to a broad set of applications. Thus, the interest for time-multiplexing on it will be low. General purpose processors, however, are very likely to provide the assumed hardware.

Chapter 4 proves the general feasibility of the presented approach. Essential elements of the protocol, like the storage and restoration of the DTU endpoint configuration or the reset of a PE, are not yet available and have to be circumvented by workarounds. These workarounds are very expensive in terms of cycles as the evaluation in Chapter 5 shows. However, the evaluation results also depict the great potential of performance improvements. As soon as the hardware allows for replacing the workarounds, the amount of cycles the whole protocol takes will drop considerably and the context-switch performance of Linux can be reached.

In the current implementation there is only one core with time-multiplexing enabled. This will probably change in the future. It has also been shown in Section 5.2 that RCTMux may be used as a replacement for idle, if just its features for setting the PE idle and starting applications are required. This will allow the kernel to turn any PE into a time-multiplexed PE when required while keeping the current behavior intact.

The DTU needs some modifications to implement the proposed design of the context-switch protocol properly. These changes may be subject of future work and include the addition of a register to trigger a reset and a special context-switch flags-register as described in Section 3.4. Moreover, M^3 needs some changes in its current implementation to properly deal with the fact that a single PE may now host multiple applications at the same time.

A side-effect of the introduced protocol for remote-controlled context-switching is its flexibility to allow for useful features that are desired but not yet available in M^3. If

used as a replacement for idle, RCTMux can put a PE back to a defined state when it gets out of control. The kernel simply needs to run the context-switch protocol without the optional storage and restoration phases. Another interesting use-case is the possibility to migrate applications from a source PE to a target PE of equal architecture. Migration can easily be achieved by running the storage phase on the source PE and the restoration phase on the target PE.

Glossary

DTU	Small hardware component next to each core (see Section 2.3)
EP	An endpoint of a communication channel in the DTU (see Section 2.3)
FS	Filesystem
KVPE	The kernel's version of a VPE (see Section 2.3)
libm3	M^3's system library
LOC	Lines of Code: see SLOC
M^3	Microkernel for Minimalist Manycores (see Section 2.2)
PE	Processing Element: combination of a processor core with its local scratchpad memory and DTU
RAM	Random Access Memory
RCTMux	Remote-Controlled Time-Multiplexer: program that implements the context-switch protocol at application-PEs (see Section 3.2)
ROM	Read Only Memory
SLOC	Source Lines of Code: number of source code lines
VPE	Abstraction of a processing element in M^3

Bibliography

[AMN+14] Oliver Arnold, Emil Matus, Benedikt Noethen, Markus Winter, Torsten Limberg, and Gerhard Fettweis. Tomahawk: Parallelism and heterogeneity in communications signal processing mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):107, 2014.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[BEF+12] Francisco J Ballesteros, Noah Evans, Charles Forsyth, Gorka Guardiola, Jim McKie, Ron Minnich, and Enrique Soriano-Salvador. Nix: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[Cad] Cadence. Xtensa customizable processor. http://ip.cadence.com/. Last accessed: 07/20/2015.

[LW09] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in l4. In *Proceedings of the second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30. ACM, 2009.

[NHM+09] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[Tan15] Andrew Tanenbaum. *Modern operating systems*. Pearson, Boston u.a, 2015.

[Ten13] Tensilica, Inc., 2655 Seely Ave., San Jose, CA 95134. *Xtensa ® System Software Reference Manual*, 10th edition, 2013.

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[Whe] Martin A. Wheeler. Sloccount. http://www.dwheeler.com/sloccount/. Last accessed: 07/24/2015.