Strasser Florian Johann BSc.

Profiling of Real-Time Operating System Services for Singlecore and Multicore Processors

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur
Master's degree programme: Telematics

submitted to

Graz University of Technology

Assessor: Dipl.-Ing. Dr. techn. Kreiner Christian
Supervisor: BSc. Dipl.-Ing. Höller Andrea

Institute of Technical Informatics

Graz, December 2014

Abstract

When designing a real-time system, early key decisions include the selection of the processor and real-time operating system (RTOS). During the comparison of the design options, performance characteristics are one of the most important features that should be considered. Additionally, in order to perform a scheduling analysis, it is necessary to be aware of the RTOS overhead. Nowadays, there is a trend to use multicore processors in hard real-time systems. However, how can we measure the runtimes of an RTOS on a multicore processor? It is common practice for companies to profile the RTOS for every platform and configuration. This is known to be an error prone task, because there is no company independent profiling suite for RTOS services. In this master thesis, an extendable and portable profiling suite including the most common RTOS profiling procedures for singlecore and multicore processors is proposed. The profiling suite contains singlecore RTOS metrics defined in Rhealstone. Furthermore, these profiling procedures are adopted to be compatible with multicore processors. For the implementation the singlecore platform i.MX28 and the multicore platform i.MX6Q are chosen and the QNX Neutrino RTOS and SafeRTOS are used.

Kurzfassung

Wenn man ein Echtzeitsystem entwickelt stellen sich am Anfang zwei Schlüsselfragen. Welcher Prozessor und welches Echtzeitbetriebssystem soll verwendet werden? Eine der wichtigsten Attribute bei der Auswahl ist die Performance des Echtzeitbetriebssystem. Die Performance des Echtzeitbetriebssystem ist auch ein wichtiger Faktor wenn man die Schedulability eines harten Echtzeitsystems berechnet. Es ist zu beobachten, dass bei der Entwicklung von eingebetteten Echtzeitsystemen immer häufiger Mehrkernprozessoren eingesetzt werden. Daher stellt sich die Frage, wie man die Laufzeiten von Echtzeitbetriebssystemen auf solchen komplexen Ein- und Mehrkernsystemen messen soll. Es ist gängige Praxis das Echtzeitbetriebssystem direkt auf der verwendeten Hardware auszumessen. Das ist allerdings ein sehr fehleranfälliger Prozess, da es dafür keine einheitliche und unabhängige Toolbox gibt. In der vorliegenden Arbeit geht es um die Entwicklung einer erweiterbaren und portablen Toolbox für die Profilierung von Echtzeitbetriebssystemen. Diese Toolbox enthält die gängigsten Profilierungs-Prozeduren für die am häufigsten verwendeten Echtzeitbetriebssystem Services wie sie in Rhealstone vorgeschlagen werden. Diese Echtzeitbetriebssystem Metriken wurden erweitert um auch Mehrkernsysteme zu unterstützen. Für die Implementierung wurden die Freescale Prozessoren i.MX287 und i.MX6Q und die Echtzeitbetriebssysteme SafeRTOS und QNX Neutrino verwendet.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das auf TUGRAZonline hochgeladene Dokument ist identisch mit der hier vorliegende Arbeit.

Graz,am ....
(Unterschrift)

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

....
date   (signature)

Danksagung

Diese Diplomarbeit wurde im (Studien)Jahr 2014 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Hiermit möchte ich mich beim Institut für Technische Informatik (ITI) und den Professoren für die Betreuung und Unterstützung bedanken. Mein spezieller Dank gilt Andrea Höller und Christian Kreiner. Dank ihrer qualifizierten und engagierten Betreuung konnte dieses Projekt im geplanten Zeitraum durchgeführt werden. Zudem konnte dank den Mitarbeitern des ITI die für diese Arbeit benötigte Hard- und Software schnell und unkompliziert Organisiert werden. Weiters danke ich dem Brandschutzbeauftragtem für die sofortige Entfernung der illegalen Couch aus unserem Büro. Durch den Wegfall meines Mittagsschläfchens auf selbiger Couch konnte meine Produktivität gesteigert werden. Neben den Mitarbeitern des Instituts möchte ich mich auch bei all meinen Freunden für das Verständnis während dieser schweren Zeit danken. Dank ihrer nur allzu häufigen Unterstützung als lebendige Debug Duck [1] konnte ich zahlreiche Fehler finden und vermeiden. Ein besonderes Dankeschön möchte ich hier meiner Familie und meiner Mutter, Helga Nagler, aussprechen. Ihre tatkräftige Unterstützung machte es erst möglich, dass ich mein Studium so problemlos abschließen konnte.

Graz, im Jänner 2015 Strasser Florian

[1] `http://en.wikipedia.org/wiki/Rubber_duck_debugging` Online accessed 19.01.2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In an industrial and safety-critical real-time system hardware and software has to match safety standards. In order to satisfy all constraints and requirements of a safety standard like the standard of International Electrotechnical Commission (IEC) 61508 an engineer has to pay careful attention to the choice of the components of such a system.

When designing a safety critical real-time system, the first important decision is the processor which needs to satisfy the requirements of the project. In some industrial applications it is required to provide a short reaction time to an event e.g. an analog value. In order to be more flexible and less expensive compared to a solution in hardware, the control system is often designed in pure software and runs as a task on a general purpose CPU. There is a trend to use a multicore processor for such a control system, in order to achieve high calculation speed, short reaction times and flexibility.

The second and also important decision is which safety-certified real-time operating system (RTOS) should be chosen. Such an RTOS provides helpful services, but these services cause overhead, the so called *RTOS Overhead*. This overhead is a necessary parameter to be known when evaluating which RTOS is suitable for an application or how much influence has to be expected by the RTOS services used in the application. A common way to get this information is to measure the runtime of the used RTOS services, which is called RTOS profiling. Most RTOS vendors provide benchmarks for frequently used RTOS services. However, those benchmarks are often performed on a single platform with a defined hardware and operating system configuration. This means that in most cases it is not possible to draw a conclusion from those benchmarked values for a project-specific setup. Also, as said in [Oer12] there is no independent and portable benchmarking or profiling suite for RTOS services. RTOS benchmarking means direct comparison between two or more RTOS. Profiling is more general, it does principally the same as benchmarking, but without enforcing a comparison. Comparing RTOS is possible on small scale processors, but hard and error prone on advanced or even multicore processors. This is, due to the high configurability of these processors and also the RTOS. However, developers of real-time systems still need to know the runtimes of RTOS services on their actual hardware and exactly their configuration of that hardware. Most profiling tools, as for example [JJN08] or [THW10], focus only on application profiling. An RTOS profiling suite would help developers to learn more about exactly their hardware and RTOS environment.

A profiling suite for RTOS is not only useful when deciding which RTOS should be used, because in a safety-critical development process the Worst Case Execution Time (WCET) bounds of a task has to be known in order to do a schedulability analysis. RTOS profiling procedures can also be seen as a special kind of dynamic timing analysis tool designed for operating system services. A dynamic timing analysis is in general not able to find WCET bounds. This is because measuring the start to end time of a function delivers the maximum observed execution time, which is less than the exact WCET [Wil+08].

However, the authors of [Wil+08] further said that safe bounds for the WCET can be found using static timing analysis. In order to do a static timing analysis it is necessary to have an abstract model of the used processor. With this mathematical, abstract and reduced model of the processor the binary file of the application is analyzed in order to find the worst case path of a function or method. The user of such a static analysis tool needs to provide some annotations like maximal expected loop iteration or maximum queue depth [Wil+08].

A static timing analysis achieves the safety by the additional effort of defining a model of the processor and calculating the worst case execution paths through the program using the users annotations and configurations. It is recommended to evaluate the results of the static timing analysis by measuring the runtime on the target system, because annotations taken by the user or the configuration of the processor may be wrong. It is clear that a lower calculated WCET than the measured execution time indicates a wrong annotation for the static timing analysis, but also a much higher value of the calculated WCET indicates a problem with the static timing analysis [Wil+08].

A profiling suite for RTOS would make it easier to assist static WCET analysis generating tight bounds for runtimes, because the profiling procedures can be used to check the annotations. As said in [Oer12] there is a lack of vendor-neutral benchmarking suits for RTOS. This is problematic, because vendors might have optimized their RTOS for their benchmarking procedures. Hence, our profiling suite is created without any RTOS vendor support.

Furthermore, there is also a trend for safety-certified RTOS to implement the support for multicore processors. Actually there is also a lack of profiling procedures for RTOS services that can handle multicore systems. That means that it would be helpful to have a vendor neutral profiling suite that also supports multicore processors when designing a safety critical real-time system and later to evaluate the performance of a static WCET analysis for system services.

This leads to the following questions:

- What is required to design a profiling suite that is portable to multiple RTOSs and compatible with single- and multi-processor systems?
- What are meaningful metrics of an RTOS that supports only singlecore processors?
- What are meaningful metrics of an RTOS that supports multicore processors?

- Which existing profiling methods RTOS services can be extended to be compatible with multicore processors?
- How can profiling methods be implemented, such that they are portable between different platforms?

1.2 Goals

The overall goal of this work is the implementation of a profiling suite for safet-certified RTOS which supports the following: vendor neutrality, meaningful profiling procedures, support for multicore compatible RTOS and simplicity of use and platform porting.

To reach this overall goal five subgoals are defined:

1. An evaluation of state-of-the-art RTOS benchmarking methods. The evaluation results deliver a list of possible profiling procedures for our profiling suite.
2. An evaluation of a defined selection of multicore architectures in combination with an also defined number of RTOSs. This evaluation results in a strategy for extending the profiling procedures in the first point to be suitable for multicore compatible RTOSs.
3. With the knowledge which profiling procedures will be implemented for our profiling suite, a user-friendly adaptation-layer is defined. This adaptation-layer is used to provide an interface for the profiling procedures. The user of the benchmarking suite will adapt the adaptation-layer to fit the specific hardware platform and RTOS.
4. Porting the profiling suite to the in point two selected hardware and RTOS. This results in a evaluation of these hardware and software platforms.

1.3 Outline

In Chapter 2 the necessary technical background to understand the further chapters is presented. After that a survey of related works is done and a comparison of important RTOS metrics is shown.

Chapter 3 introduces the concept and design of the profiling suite. The concept focuses on the coverage of important details of advanced processor architectures and how to deal with them. The design is made in a way such that porting between RTOS and hardware and implementation of additional profiling procedures is simple.

The implementation of the profiling suite is described in Chapter 4. Furthermore, problems and implementation details on every hardware and RTOS are stated.

The measurement results can be found in Chapter 5. The results cover singlecore measurements on SafeRTOS and QNX Neutrino and multicore measurements on QNX Neutrino in BMP mode.

The last two Chapters 6 and 7 conclude the work and recommend possible ways to further improve the profiling suite.

Chapter 2

Technical Background and Related Work

This chapter covers the basic background knowledge of real-time operating systems and timing analysis. Furthermore, the architecture of real-time systems is described with a focus on multiprocessor system-on-chip and the differences regarding profiling are stated. Thereafter the current existing approaches are described in the related work section.

2.1 Real-time Systems

A real-time system guaranties a reaction to an event within a given timeslot. The end of the timeslot is called a deadline. Real-time systems also provide deterministic and consistent behavior, which is necessary in order to verify the systems timing behavior. Real-time systems can be classified by the consequence of missing a deadline [LY03]:

Hard Real-Time System: Missing a deadline could have serious consequences.

Soft Real-Time System: A late value is not useless at all, but a missed deadline often means that the quality-of-service decreases.

In this work we focus on hard real-time systems, which can be found for example in an aircraft, a power station or a car. Missing a deadline in such a safety-critical system can endanger human life. In order to ensure that the system reacts within the specified time limits on the event, it is important to know how long it takes to deliver the result in the worst case. How worst case times can be found will be explained later in this chapter.

2.2 Real-time Operating Systems

Real-time Operating Systems (RTOS) are intended to make it easier to develop a real-time system. The RTOS provides features such as scheduling, interprocess communication, semaphores and resource management [LY03]. An RTOS reduces not just the development time of an embedded system, especially in a safety-critical system it is highly recommended to utilize a certified RTOS. It is beneficial, because it minimizes the probability of making a mistake when implementing RTOS services.

2.2.1 Processes and Tasks

One of the central responsibility of an RTOS is to manage more than one task at a time [Tan07]. Therefore the operating system schedules tasks sorted by their priorities. A task can either be a process or a thread.

Process: A process is the most heavyweight construct. Processes are scheduled by the RTOS and have their own address spaces [Tan07]. If there exists more than one process in the system, the processes need a control block and may have a priority. It is also necessary that the processor provides a memory management unit (MMU) in order to switch between the address spaces. Low-end microcontroller often lack of such an MMU. Thus small scale RTOS often have only a single process, which uses threads that share the same address space.

Thread: A thread is owned by a process and is also scheduled by the RTOS. Threads have their own control blocks including the priorities and the stacks [Tan07].

Coroutine: Coroutines belong to a process or thread and share their address spaces, stacks and heaps. Coroutines are scheduled by the process or thread they belong to. The scheduling is done cooperative.

Depending on the scheduling policy, processes and threads may be preemtable either at a special scheduling point in the code or at any time when an event occurs [Tan07].

2.2.2 Scheduling

The scheduler of an RTOS decides which task should be started next and which will be suspended. The aim of a scheduler in a real-time system is to comply with deadlines and to be predictable. Tasks might occur either periodic or aperiodic. Aperiodic means that the task may occur unpredictable.

In general a set of tasks is said schedulable, if the following condition holds:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1 \tag{2.1}$$

Where C_i is the execution time of the task i, P_i is the period of task i and m the number of tasks in the set [Tan07].

A set of tasks may be scheduled static, which means that the schedule is defined at compile time. However, exact information about the deadlines and the work has to be available beforehand. In contrast to static scheduling, dynamic scheduling decides at runtime which task should be scheduled next [Tan07].

Dynamic scheduling can be either preemptive or non-preemptive. That means, a preemptive scheduler can interrupt a running task at any time, but a non-preemptive scheduler has to wait until the running process ends or suspends voluntarily. An example for non-preemptive scheduling is cooperative scheduling. Preemptive algorithms are priority first-in-first-out (FIFO) with time slicing, round robin, rate monotonic scheduling and earliest deadline first [LY03] [Tan07]. In [TTN09] the authors present a comparison of RTOSs including their scheduling strategies. Most compared RTOSs use priority FIFO with time slicing due to the low overhead and good predictability.

2.2.3 Interprocess Communication

According to [Tan07] interprocess communication (IPC) contains three subtasks.

1. Communication between processes the so-called message passing.
2. Protection of shared resources and critical regions, the mutual exclusion.
3. The barrier, where a process A produces data, which will be used by another process B. B has to wait until A has completed the production.

For further information on IPC we refer the reader to [Tan07] Chapter 2.

2.2.4 Interrupts

One of the most important tasks of an RTOS is to react on an interrupt event. Such an event is handled by an Interrupt Service Routine (ISR). If an interrupt occurs the processor saves the state and jumps to the ISR [LY03].

It is a design decision of the embedded system designer where to do the full handling of an interrupt. It could either be done directly in the ISR or the ISR is only used to notify a task about the occurrence of the interrupt. The first method is faster because no scheduling is necessary, but the drawback is that while the ISR is executed, other interrupts may be blocked.

2.2.5 Priority Inversion

Assume, there are three tasks with different priorities, A, B and C, running on a RTOS. The lowest priority task A acquires a lock. After that the highest priority task C tries to acquire the same lock, which means that C has to wait until A releases the lock. However, that will not happen until task B stops the execution and A can finish its work. Due to this fact, C will probably miss its deadline. To avoid such a situation some RTOSs contain a priority inheritance protocol, in which the task that holds the lock, inherits the priority of the highest priority process that tries to acquire the same lock [LY03]. Using such a protocol task A in our example would get the priority of task C, so it would be scheduled before B.

2.2.6 Application Programming Interface

Operating systems have two main functions: abstraction for user programs and management of the hardware. The abstraction, which will further be called application programming interface (API), is mainly used by the users code [Tan07]. Most RTOSs provide a C language API for their users.

There is no global standard that is implemented by all RTOS. However, there are standardized APIs such as POSIX, defined by the international standard 9945-1 [IEE96], which is also implemented by desktop operating systems like UNIX, Linux, BSD, System V and many more [Tan07]. POSIX is well known by programmers and therefore a good choice because it is portable between POSIX compatible operating systems. Most RTOSs implement only a subset of the standard because some function definitions make it hard to keep predictability. Other RTOS vendors like μITRON [ST02], which is a frequently used RTOS in Japan, have developed their own API. Also FreeRTOS [Fre14] implements its own API.

2.3 Timing Analysis

In order to do a schedulability analysis, it is essential for a hard real-time system to know how long the execution of a program lasts in the worst and the best case. This is called the worst case execution time (WCET) and best case execution time (BCET). In general it is not possible to calculate the WCET and BCET for a given program without prior knowledge of further parameters, to achieve this it would be necessary to solve the halting problem [Wil+08]. The execution time of a program depends on the internal state and the input data. If the worst case state and the worst case input for a program is known, it is possible to find the exact WCET, but as stated in [Wil+08] the worst case input is hard to find in most cases.

Today in industrial practice, it is common to measure the end-to-end execution time of a program. This measurements determine the minimum observed execution time and the maximum observed execution time. In general, this values will overestimate the BCET and underestimate the WCET. Measuring execution time will only deliver estimations of the WCET and BCET, it cannot deliver guarantees for upper or lower bounds on the execution time [Wil+08].

Bounds on execution times can only be delivered by methods that consider all possible execution times of the program. Such methods abstract the hardware in order to make a timing analysis possible. However, an abstraction means that information is lost, which results in an overestimation of the WCET and an underestimation of the BCET [Wil+08]. For safety-critical applications there are two main criteria for timing analysis:

1. Does the analysis deliver *estimations* or *bounds*?
2. How *precise* are the results? How close are the results to the exact WCET and BCET?

Figure 2.1 shows a symbolic distribution of execution times of a task and also the measured execution times. Figure 2.1 also introduces the notations used in this work.

2.3.1 Static Timing Analysis

Static methods do not rely on measurements of executed code, instead the timing behavior of a program is calculated using the binary code for the target hardware. Therefore an abstract model of the hardware is utilized to analyse the executable file. The analyzer collects all possible paths of execution from the binary code and generates a Control Flow Graph (CFG). However, the range of possible input values and internal states is also too large to find the worst case execution path. Therefore the user has to provide some annotations for the analyzer. For example: ranges for the input values, loop bounds or shapes of nested loops [Wil+08].

Static methods are able to obtain bounds on the execution time, which is the biggest advantage of using such an expensive method. However, static timing analyzers needs an abstract model of the hardware, which is often not available, since detailed information

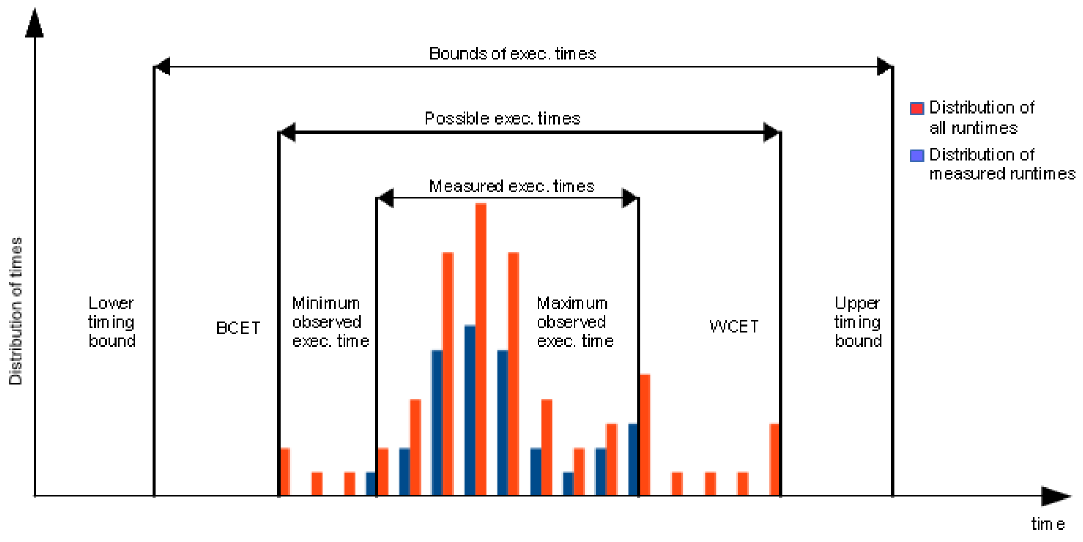

Figure 2.1: The image gives an overview of the notations used in this work. The red bar represents the distribution of all execution times of a task. The lower and blue bar represents the measured execution times. The blue bar is always lower than the red, because not all execution times are measured. It can also be seen that the upper and lower timing bounds are higher and lower than the WCET and BCET. Adapted from [Wil+08].

about the hardware architecture is intellectual property of the vendor. So the number of potential hardware is limited. Another problem are the annotations, over- or underestimated value ranges or loop bounds may result in an overestimated or even false bound for the WCET. Hence, it is recommend to verify the bounds found by such a tool by measuring the execution time on the real hardware [Wil+08].

2.3.2 Dynamic Timing Analysis

Measurement-based methods, in general, deliver estimates, not bounds of a WCET or BCET. Dynamic timing analysis measures the end-to-end time of a program, task or function. There are approaches that measure the runtime of small code segments and combine the results in a CFG. This measurement based methods replace the processor-behavior analysis, which is done by the abstract model in the static methods. However, this approaches can also not guarantee that the worst case execution time is found, because the measurement of the code segments might be unsafe [Wil+08].

The major advantage of dynamic timing analysis is the usability on various hardware and thereby the realistic results. The measurements are also simple to implement, because hardware timer or other timing features can be utilized to acquire the runtime of a task. Profiling of operating system services can be seen as a subset of dynamic timing analysis, because the measured times for system services can be utilized in a CFG as stated above.

2.3.3 Simulation

Using a simulator for estimating the WCET is also a measurement-based method. The key advantage is, that it is possible to derive rather accurate estimations for a given set of input values. However, as stated in [Wil+08] not all simulators offer reliable results, because they are not cycle accurate.

2.4 Measurement of Real-time Operating System Performance

2.4.1 Benchmarking and Profiling

Most related works focus on benchmarking of real-time operating systems. Benchmarking an RTOS means, measuring meaningful and especially comparable values, in order to tell that one RTOS is better than the other. Comparing RTOS is a so called apples to apples comparison [LO11]. They do the same job, but in different ways for also different purposes. Most RTOS distributions are designed to perform good in special situations or tasks. For instance, an RTOS might implement a mutex with a priority inheritance protocol. This mutex can not be compared with a common mutex without that feature.

Especially, benchmarking or comparing RTOSs on advanced hardware is a hard and error prone task. In order to provide meaningful results, all RTOS need to have the same conditions to run. Advanced or even multicore hardware requires complex configurations of all system components. To get the same conditions on such hardware it is necessary to configure all RTOS under test exactly the same way. In most cases this is not possible or may cause disadvantages for some RTOS implementations.

However, profiling in contrast to that, focuses on the creation of profiles of an existing system, in order to do monitoring, optimization or identification. The main difference between benchmarking and profiling is the perspective. Where RTOS benchmarking views many RTOS on one hardware, RTOS profiling considers exactly one RTOS and one hardware.

RTOS profiling procedures are the same as RTOS benchmarking methods, both measures metrics of an RTOS. For this work the term profiling is used for our design and implementation regarded to our hardware and RTOS. The term benchmarking is used for related works.

2.4.2 Metrics of an Real-time Operating System

A metric of an RTOS is a measurable property, which can be used for the comparison or the profiling of an RTOS. Measurements of RTOS metrics can also be useful for choosing an RTOS for a specific application. This section should give an overview of measurement based selection criteria of an RTOS. There are other selection criteria for an RTOS as for instance: suppliers reputation, overall cost, technical support and others [TTN09]. However, the remainder of this work focuses on the measurement of system services.

In literature same metrics are often named differently. In this work we will stick to the following definitions and if necessary explain the differences compared to these definitions.

System Service Metrics

- **Context Switch Time:** The time taken by the RTOS to switch between two tasks without any other task or interrupt executed between [KP89]. The priorities of the tasks are equal. This situation is illustrated in Figure 2.2.

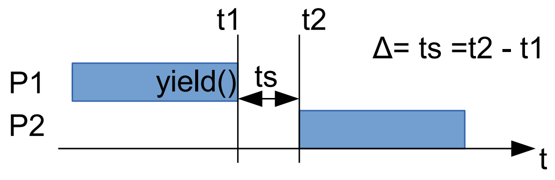

Figure 2.2: Context switch time as defined by the Rhealstone benchmark. The priorities of both tasks, P1 and P2, are equal. The parameter t_s is the time to schedule and switch the context to another task and Δ is the total latency of the system service. Adapted from [KP89].

- **Preemption Time:** Time to switch to a previously inactive, higher priority task [KP89]. In other words, the time to react on an external event using a task. In Figure 2.3 an interrupt service routine sets a high priority task ready to run.

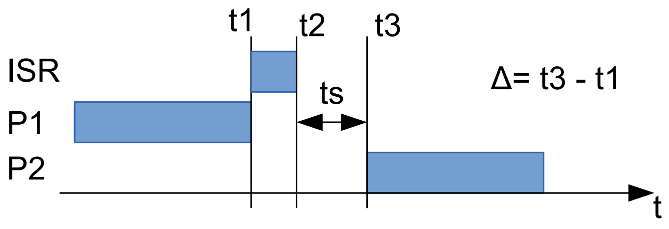

Figure 2.3: Preemption time as defined by the Rhealstone benchmark. The priority of task P2 is higher than the priority of P1. An interrupt wakes up P2 which interrupts the execution of P1. The latency Δ contains the time consumed by the ISR to wake up P2 and the time to switch context t_s. Adapted from [KP89].

- **Semaphore Shuffling Time:** Time between a request to lock a semaphore, which is locked by another process and the time the request is granted [KP89]. An illustration of this situation can be found in Figure 2.4.

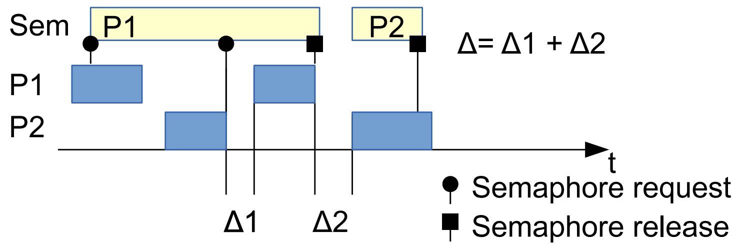

Figure 2.4: Semaphore shuffling time as defined by the Rhealstone benchmark. The task P2 tries to acquire the semaphore, which is held by P1, so P2 has to wait until P1 completes the work and releases the semaphore. The latency Δ is the total time consumed by the operating system to pass the semaphore to P2. Adapted from [KP89].

- **Deadlock Break Time:** Time consumed by the priority inheritance protocol to solve a conflict as stated in the Section 2.2.5. Figure 2.5 gives a graphical overview about this scenario.

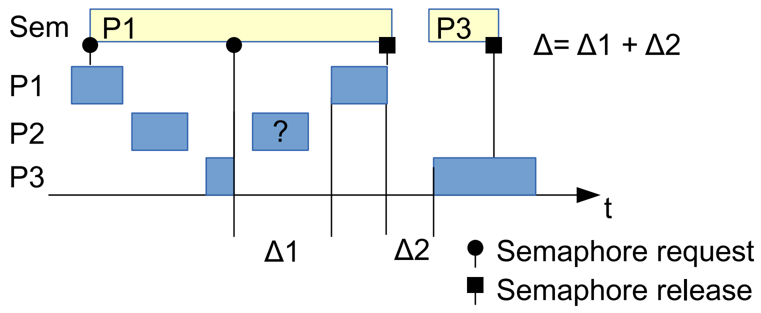

Figure 2.5: Deadlock breaking time as defined by the Rhealstone benchmark. P1 is the lowest priority process, which acquires the semaphore, P2 is another medium priority process that handles a high workload and P3 is the highest priority process, which needs to enter the critical section, which is protected by the semaphore. It depends on the priority inheritance protocol of the operating system if P2 is scheduled after P3 tries to acquire the semaphore. The latency Δ is also defined as the total time consumed by the operating system to pass the semaphore to P3. Adapted from [KP89].

- **Intertask Message Latency:** Time consumed for sending a message from one task to another [KP89]. A symbolic implementation can be seen in Listing 2.1 and 2.2, adopted from [TTN09].

Listing 2.1: Task P1 sends a message to P2

```
t_send = getCurrentTime();
send_message(P2);
...
```

Listing 2.2: Task P2 recives the message

```
...
msg = rcv_message();
time_rcv = getCurrentTime();
time = time_rcv - time_send;
```

- **Memory Acquire and Release Time:** Time to acquire and release a fixed amount of memory [TTN09]. The time can be measured as stated in Listing 2.3, which is adopted from [TTN09].

Listing 2.3: Memory acquire and release time

```
t1 = getCurrentTime();
mptr = malloc(block_size);
free(mptr);
t2 = getCurrentTime();
time = t2 - t1;
```

- **Frequently Used System Service Latency:** The time between calling a system service until the function returns [Xu+08]. The latency can be measured in a way as it can be seen in Listing 2.4, which is adopted from [Xu+08].

Listing 2.4: Latency measurement of frequently used system calls

```
t1 = getCurrentTime();
freq_used_syscall();
t2 = getCurrentTime();
time = t2 - t1;
```

Memory Usage Metrics

- **Memory Footprint:** The usage of RAM and ROM of the operating system [TTN09].
- **Heap Usage:** The consumption of heap by the operating system [Xu+08]. The heap usage of a RTOS function can be measured as demonstrated in 2.5, which is adopted from [Xu+08].

Listing 2.5: Measure the heap consumption of an operating system function

```
heap_ptr = getHeapPointer();
syscall(); // e.g. put values on a queue
heap_ptr2 = getHeapPointer();
heap_usage = heap_ptr2 - heap_ptr;
```

Other Metrics

- **Interrupt Latency:** Time between occurrence of an interrupt and execution of the first instruction in the ISR [KP89]. This metric depends only on the used hardware.
- **Network Throughput:** The amount of data sent or received in a defined time interval. Listing 2.6 gives an idea of the measurement.

Listing 2.6: Throughput measurement

```
while(time < max_time){
        send(data_block);
        count++;
}
sent_data = count * sizeof(data_block);
```

- **Peripheral Function Latency:** The time between calling a peripheral function and the return of the function[Xu+08]. A peripheral function can be for instance UART, I2C or SPI. The measurement of peripheral functions is done in a similar way as the measurement of system call latencies.

2.4.3 Benchmarking Methods

Rhealstone is a benchmark for RTOS services published in 1987. The authors defined six low-level metrics: context switching time, interrupt latency, preemption delay, deadlock break time, semaphore shuffling time and datagram throughput time [KP89]. Rhealstone metrics are the most frequently considered metrics of recent RTOS benchmarks. However, most implementations adapted the metrics or implements only a subset of the metrics [Oer12][Xu+08][TTN09]. As stated in [TTN09], Rhealstone should not be used unchanged. For instance, interrupt latency as defined by the authors of Rhealstone benchmarks only depends on the hardware. Thus it should not be seen as an RTOS benchmark.

Hartstone is also a benchmark for RTOS. The difference to Rhealstone is that the authors of Hartstone claim that it is risky to draw conclusions from low-level metrics. Therefore, the authors defined benchmarks, that stress tests the RTOS under different conditions using sets of tasks. The tasks have to meet their deadlines [Wei90]. However, the problem is that in a real application the tasks would not behave in the same way as the synthetic tasks used in the test sets.

EEMBC (Embedded Microprocessor Benchmark Consortium) [EEM14] defines numerous synthetic benchmarking suites for embedded systems which targeting the hardware performance. That means that these suites are designed to run without an RTOS.

Mibench is similar to the EEMBC benchmark suite but is public available [Gut+01].

2.4.4 Measurement

In our previous work [Str14] we did a comprehensive study of measurement methods for RTOS service runtime and application profiling. We evaluated four selected high speed methods for taking timestamps, Performance Monitoring Unit (PMU), timer, GPIO pin toggling and an RTOS native method called ClockCycles. It turns out that nearly all evaluated methods exhibits deviations in the time to take a timestamp.

Numerous recent works uses timers to take timestamps and GPIO pin toggling to verify the timer results. Others use only the timer to perform benchmarking. In [Str14] we showed that timers and GPIO pin toggling has to be handled different on advanced or even multicore hardware. GPIO pin toggling is no more a good choice on every hardware, because system bus communications to toggle a pin are time consuming and caching problems can also lead to unexpected behavior. Timers also exhibits deviations caused by caching problems and bus communication delays. That means, such deviations and delays needs to receive attention when evaluating measurements.

The PMU is shown to be a valuable alternative to the timer and the GPIO methods. It is mounted on the co-processor of numerous advanced processors as for instance the ARM Cortex-A9 CPU [ARM10]. Due to that, the PMU is able to take cycle accurate measurements of running code. Reading the cycle count register takes, as shown in [Str14], an almost constant amount of time. However, the PMU can not be used to take timestamps synchronously on a multicore cluster. For instance, measuring the message pass time from one core to another can not be done by this technique.

2.4.5 Results and Interpretation

RTOS benchmarks, in general, try to deliver comparable values in order to be able to show that one RTOS is better or faster than another one. Rhealstone [KP89] provides six averaged benchmark values of the RTOS combined in a single value, the so-called object or application Rhealstone per second value. However, it is clear that combining measurements in a single value is not meaningful at all, because developers want to know how the RTOS performs for the specific needs of their application.

It is also not recommendable to provide only average values in case of a real-time system at all. It is better to provide more information about a measured metric. This could be for instance, the standard deviation and minimum and maximum observed execution time. Such an evaluation can be used by developers to get a feeling about the jitter of a system service.

Some benchmarks deliver proportional values related to the benchmark. This can be for instance be the relation of uninterrupted work time divided by the total time, as it can be seen in [Kha+09]. This value can be compared when running different RTOS on the same hardware, but it does not tell the developer anything about the time, a system service takes to execute on a hardware.

Concluding, a profiling suite should provide:

- Maximum observed execution time
- Minimum observed execution time

- Average execution time
- Standard deviation of a metric

Another way is to deliver the resulting values condensed in a histogram. This is only possible if the minimum and maximum runtime can be estimated beforehand. Or, if enough memory is available, the raw measurements of the system service latencies can be delivered directly to the user, in order to provide a full overview and enable the developer to draw own conclusions.

2.5 Hardware Architecture of Real-time Systems

Complex hardware influences the results of profiling procedures in various ways. Such hardware includes many features that improves the average calculation speed, for instance: out-of-order execution, pipelines, caches or the presence of more than one processing element. However, these features will only decrease the best and average case runtime, but the worst case runtime stays the same or might even increase. That means that the jitter of an RTOS profiling procedure will be more important when using complex architectures, compared to simple architectures without caches or other features.

This chapter provides a brief overview about the architecture of improved single- and multicore systems and focuses on the hardware used for this work. For more detailed information on multicore architectures in general, the reader is recommended to read [Moy13].

2.5.1 Cache

Caches are used to bridge the memory gap between the CPU and the main memory. Figure 2.6 shows the typical memory architecture on a modern CPU like ARM. Furthermore, it can be seen that the first level cache is divided into an instruction and a data cache (I-Cache and D-Cache). The read only I-Cache has a greater impact on performance than the D-Cache [Moy13]. The typical size of L1 caches is in range of 8kB and 64kB.

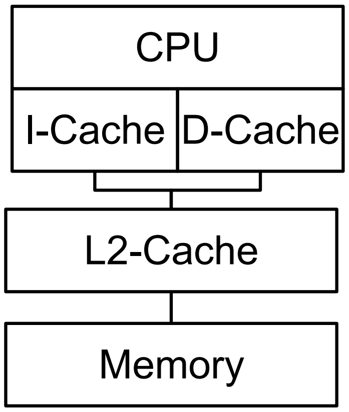

Figure 2.6: Typical cache structure of a singlecore CPU. Adopted from [Moy13]

Caches, in general, improve the average runtime of a program, but the worst case runtime is hard to predict and especially static timing analysis methods need to be adopted

[Cho+13][YZ08]. The authors of [Cul+10] show that the caching strategy also influences the predictability. Often processors are designed to be cheap and hence also the utilized caches does not support predictable caching strategies like last recently used (LRU). Instead of that the chips often support only strategies, which are cheaper or easier to implement as for instance: pseudo last recently used (PLRU), first-in-first-out (FIFO), pseudo round robin (PRR) [Rei08], or random replacement, which are harder to predict.

CPUs that use a memory management unit (MMU) typically contain a second kind of cache, a so-called translation lookaside buffer (TLB). This buffer improves the virtual-to-physical address conversation rate. A TLB hit takes about 1 clock cycle, while a miss can take up to 100 cycles [Moy13].

2.5.2 Singlecore Platforms and i.MX28

For this work we differentiate two kinds of singlecore platforms, processors with and without cache. The most works on benchmarking and static timing analysis is done on singlecore platforms without cache, due to the better predictability of these platforms [TTN09][Oer12][Xu+08]. During this work we will talk only about singlecore platforms with at least a first level cache, such as the i.MX28 platform.

2.5.3 Introduction to Multicore Platforms

"Welcome to the age of embedded multicore." Bryon Moyer [Moy13]

There is a wide scope of multicore platforms for nearly every purpose of computing. From a processing perspective there are two main distinctions, the homogeneous and the heterogeneous architectures [Moy13].

Homogeneous architectures contain N times the same core with the same features. That means that every core can do the same work as every other. Such a processor is for example the i.MX6Q [Sem14] quad-core Cortex-A9 ARM processor, which uses the same cores several times. It is possible to migrate a task to every core of the processor.

Heterogeneous architectures contain different cores. One might be a general-purpose processor and another one can be a specialized signal processor. Not all processing elements in a heterogeneous architecture can do the same work. Task migration is hardly possible.

2.5.4 Memory Architecture of Multicore Platforms

The structures of memory in multicore platforms can be categorized in two main categories, the first is the shared memory and the second one is the distributed memory. Depending on the kind of used memory, when designing a multicore platform it is also an issue how to achieve cache coherency. The used memory architecture and cache coherency protocol affects the performance of a real-time system significantly.

Shared Memory Architectures

All processors and programs have direct access to the memory. The program can address the whole memory regardless of the physical location.

In general it can be distinguished:

Uniform Memory Access (UMA) Multiprocessors with equal access to the main memory are classified as UMA [Moy13]. These processors are called symmetric multi processors (SMP). Latencies in memory access may occur, if two processors tries to simultaneously access the memory block. Such memory structures are often seen in general-purpose multicore processors as for instance Intel or ARM. The structure of an UMA model can be seen in Figure 2.7.

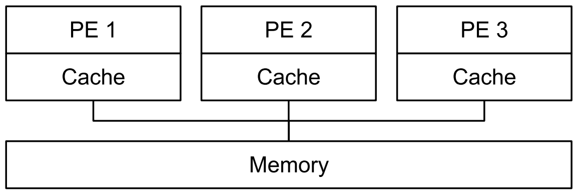

Figure 2.7: Uniform Memory Access structure. In case of a cache miss, all processing elements accesses the memory via shared memory bus, which might be a bottleneck if the bandwidth is too low. Adopted from [Moy13].

Non-Uniform Memory Access (NUMA) These multiprocessors consists often of two or more SMPs which are linked together with a memory bus [Moy13]. SMPs can access their own memory directly, but when accessing the memory of another SMP block there might be higher latencies due to the memory bus. Developers should reduce the amount of interprocessor communication to a minimum. The structure of a NUMA model can be seen in Figure 2.8

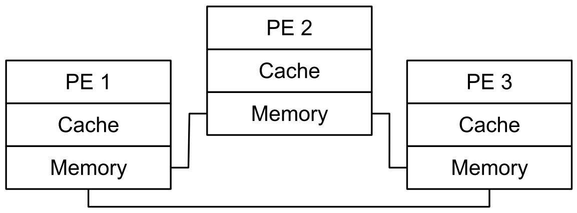

Figure 2.8: Non-Uniform Memory Access structure. Every processing elements is connected to each other processing element in order to have access to their memory. Addressing another processing elements can be done in the same way as addressing the own memory block, but it causes much higher latencies. Adopted from [Moy13]

When designing profiling procedures for an RTOS running on a multicore architecture it is necessary to consider latencies caused by the memory architecture. Therefore the communication between processing elements, or cores, is an essential value to be benchmarked.

Distributed Memory Architectures

The difference between distributed memory architectures and NUMA is that the processing elements do not share the same address space. The memory of other processor can only be accessed via communication network. Access times to another processors memory varies greatly, depending on utilization and speed of the network.

Cache Coherency on Multicore Platforms

Cache coherency is an important issue to consider when thinking of timing behavior of RTOS services, because locking and shared resources rely on coherent memory. There are two important categories of cache coherency protocols: Snooping and directory based. Snooping tends to be faster, if enough bandwidth is available on the bus. Directory based protocols are slower and more expensive, but they scale better than snooping schemes. The processor used for our work, the i.MX6Q, utilizes a snooping cache coherency protocol.

2.6 Real-time Operating Systems for Multicore Platforms

The main difference between a singlecore and a multicore RTOS is the presence of real parallelism. Thread related issues which are no problems in a singlecore environment may rise up in a multicore platform [Moy13]. A multicore RTOS has to pay special attention to the synchronization of shared resources, the occurrence and handling of interrupts and the allocation of tasks to processing elements.

All these operating system tasks depends on the way how an operating system views and manages the memory. This is also used to classify multicore RTOSs [Moy13] in symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP) RTOS.

2.6.1 Operating Modes

Symmetric Multi Processing (SMP): In a SMP architecture all processing elements are equal. They can execute the same code and share the main memory. That means the multicore system is required to be homogenous. The OS handles all cores at a time and manages the migration of processes and threads over the processing elements. The programmer might have no control or knowledge on which core the thread will be executed.

SMP systems are often used in end-user applications, where high processing speed or throughput is needed. However, due to its amount of shared resources, it is very hard to find bounds for the WCET of such a system [THT08].

Asymmetric Multi Processing (AMP): As defined in [Moy13], if an architecture is not SMP it is AMP. In an AMP system the user has control of a tasks affinity to a core, if a single OS instance is used for the system. It is also possible to run an OS instance per core. This feature is necessary when using a heterogeneous multicore system, where not all cores are equal and memory might be distributed. Knowledge about the task to core affinity and possible migration is also beneficial for bounding the worst case performance. For this reason, AMP RTOS are more frequently used in hard real-time systems [THT08].

Bound Multi Processing (BMP): BMP is a variation of SMP, where the programmer retains the ability to pin a task to a core at run time or at compile time [Moy13]. This operating mode is also possible to be used for hard real-time systems due to its improved predictability compared to a SMP architecture.

2.6.2 Thread Safety

When a thread wants to enter a critical section it is common in a singlecore RTOS, as for instance FreeRTOS [Fre14] or SafeRTOS [Wit14], it simply turns off the interrupts. This does not work anymore in a multicore system. Turning off the interrupts will only affect one core and the interrupt may fire on another core. A solution to this is freezing all cores while a thread is working in a critical section [Moy13]. A better solution would be to utilize hardware synchronization features such as atomic XCHG on Intel or LDREX and STREX on ARM processors. Sometimes it is necessary to use a mutex for locking a critical section, which means that other processes need to be queued. Queues are important factors when doing a timing analysis, because of the required memory allocation per new element. In addition to allocation, synchronization and sorting of the queue may be necessary, which causes noticeable overhead.

2.6.3 Debugging on a Multicore Platform

In a singlecore system it is common to use a JTAG device and breakpoints for debugging. If a breakpoint is reached the processor is halted and information about the actual system state are transferred to the development software.

A multicore system introduces several problems to the debugging process. Code is executed parallel on every core. It is possible to use breakpoints in the same way as it is done by singlecore systems, but the information has to be merged with all other cores to obtain an overview of the system. This is a complicated process and the developer might fail to see important states of the system.

A solution for this is to use a so-called trace tool. A trace tool that captures application and system events is called event-trace. A system event is any system service as for instance: state transition, context switch or interrupt. An event contains information about the system state and the related timestamp. These events are logged by the trace tool for subsequent evaluation. A JTAG device transfers the collected data of the trace tool to the PC where it can be evaluated by a special software.

Trace tools can be obtained by the operating system vendor, if one is available. For instance for Linux the trace tool is called LTT, for VxWorks WindView and for ThreadX TraceX [Moy13].

It is hardly possible to utilize a trace tool to profile RTOS services. At first, trace tools are designed to debug a user applications, which means that these tools are not intended to do profiling of RTOS services. Second, trace tools are supplied by the RTOS vendors, which means there is no compatibility to other RTOSs. Due to this it is hard to deliver comparable results. Third, depending on the implementation, the logging of events might cause overhead in the system services.

2.6.4 Available RTOS Implementations

Table 2.1 shows a number of selected RTOSs implementations and their operating modes. The RTOSs are selected because of their compatibility to safety standards and multicore platforms. All RTOSs are at least compatible to SIL-3. In Table 2.1 it can be seen that all selected RTOSs implement a SMP mode and all implement either AMP or BMP mode.

RTOS	**Provider**	**Operating Mode**		
		AMP	**SMP**	**BMP**
Neutrino	QNX	No	Yes	Yes
VxWorks	Wind River	Yes	Yes	No
Integrity	Green Hills	No	Yes	Yes
PikeOS	Sysgo	Yes	Yes	No
ThreadX	expressLogic	Yes	Yes	No

Table 2.1: Comparison of available real-time operating systems according to their operating modes. Adopted from [Hoe13]

2.7 Related Work

In recent years there have been only a few scientific publications discussing how to measure the performance of RTOSs. Especially there are a few publications in measuring RTOS system service latencies on a complex hardware or even a multicore processor.

2.7.1 Timing Analysis on Small Scale RTOS

The most similar approach compared to this work was published by Örnvall [Oer12]. Örnvall does a benchmarking of RTOS for the use in a radio base station. The aim of the work is to compare two RTOSs for small scale embedded applications on the same microcontroller. Therefore the author implements the Rhealstone benchmark [KP89]. The reason for choosing this benchmark method was because it is easy to understand, rather well known and the methods are meaningful for the work. Further the author implements the benchmarking tool in a way such that the porting between the RTOSs is as simple as possible. Additionally, it is shown that it is possible to write a portable benchmarking tool by using preprocessor macros. To verify the resulting times an oscilloscope and a logic analyzer are used. The comparison of both RTOS shows a noticeable difference in the runtime of system calls. The use of Rhealstone benchmark and the portable implementation are also intended by our work, but the difference is the use of multicore processors. In general the utilization of more powerful processors also implies a multi-level cache which makes it more difficult to benchmark such a system.

The authors of [Xu+08] does a runtime benchmarking of FreeRTOS and its hardware abstraction. The authors of [Xu+08] implement system service benchmarks in a similar way as [Oer12], but additionally the memory footprint is calculated and the hardware abstraction layer of FreeRTOS is benchmarked. Both, [Xu+08] and [Oer12], use an oscilloscope and a logic analyzer to verify their benchmarking results. For our work it is of special interest that the author of [Xu+08] discusses the design process of benchmarking methods for RTOS.

A survey and performance evaluation of RTOS for small microcontrollers is presented in [TTN09] by Tan and Su-Lim. They discuss the attributes, which are concerned for the selection of an RTOS. The survey discusses how a RTOS can be compared using benchmarking, RTOS features and API richness. The most interesting part is the benchmarking section where the authors say that the Rhealstone benchmark is outdated, but with a few modifications a subset of the benchmark can be utilized. As shown in [Oer12] and [TTN09] Tan and Su-Lim also use only one microcontroller in order to present comparable results. Tough this is a small scale microcontroller, which means that there are no caches and no difficult to predict processor features as it is the case in this work. It is also interesting that the benchmarking ends without a clear winner, but they show that every RTOS has different strengths and weaknesses.

A survey about performance metrics for real-time systems is presented in [GH10]. The authors state that benchmark programs are hardly portable because of the different system call methods implemented by the numerous RTOSs. Standards like POSIX 1003.4 should assist developers in building real-time systems with standard interfaces. Furthermore, standard benchmarks, as for instance: Rhealstone, Hartstone, DIN19242, MiBench and SNU-RT benchmark are surveyed. All these benchmarks are criticized to be outdated or not meaningful, due to this the authors of [GH10] recommends to form a work group to specify, promote and report benchmarks.

Walls stated in [Wal12] that numerous vendors publish measurements of their RTOS, but this data is sometimes hard to interpret because of the taken annotations made for the tests. Furthermore, Walls shows that it is a good idea to implement own measurements in order to see how good a RTOS fits the needs of the users application. He also demonstrated a way to measure relevant metrics of RTOS: memory footprint, interrupt latency, scheduling latency and kernel services. Interrupt latency is measured similar way as presented in [KP89] and preemption delay and scheduling latency are similar to task switch time.

2.7.2 Other Benchmark Implementations

The authors of [Kha+09] attempt to identify metrics for a RTOS in order to compare it with an other one. They select four metrics: task switching time, preemption delay, interrupt latency and semaphore shuffling time. The measured values are added up to form a RTOS-comparison-value similar to Rhealstone [KP89]. Critics, as for instance [TTN09], say that, interrupt latency does not depend on the RTOS, which means that this is not a good value to compare RTOS. Second, combining time measurements to a single value is also known to loose information about the particular performance of the services.

Ugurel and Bazlamacci in [GC11] stated that the most important factors for choosing a RTOS for a project are the memory footprint and the context switching time. Their measurement is done on a FPGA board and a Xilinx MicroBlaze CPU. Two RTOSs, Xilkernel and μC/OS-II, are benchmarked. The way of measuring the context switch time is different to most other papers. Two tasks are running for 30 seconds and switches contexts as often as possible to the other task. The result of the benchmark is a number of context switches per time. Converting those counts of context switches to a time per context switch gives an average value of the system service. The results shows a significant difference between the two benchmarked RTOSs.

There are only a few related works for benchmarking RTOS in a safety-critical process. One is [Kim+10], where the authors benchmark QNX for a nuclear power plant. The authors stated that a RTOS in a safety-critical environment needs safety relevant features as for example priority inheritance protocols and deadlock avoidance mechanisms. Furthermore, the authors benchmarked four criteria of QNX: Context switch time, message passing, synchronization and deadline violation. In the case of context switching, it is distinguished between inter-thread and interprocess context switching. Also message passing and synchronization distinguish between different kernel functions, as for example, mutex and semaphore or sending data and receiving data. The results shows that there are significant differences in runtime between different system calls. Unfortunately, it is omitted to demonstrate the operating principle of the deadline violation test.

PapaBench [Nem+06] is a free to use real-time benchmark. The benchmark is based on the paparazzi project, where an application for an unmanned areal vehicle is designed. It is directly integrated in the application and measures the runtime of the application tasks. The benchmark is intended to be valuable for statical timing analysis. The authors state: "It is important to experiment and evaluate every static WCET analysis" [Nem+06].

In [Mar+11] large scale RTOSs, RT-Patch Linux and Xenomai, are compared with an embedded RTOS, eCos. The authors benchmarked five metrics of the RTOSs: Interrupt latency, task switch time, preemption time, deadlock break time and network package processing. The interrupt latency scenario is similar to the Rhealstone [KP89] interrupt latency, but in this work the interrupt is triggered by software. The results show that task switching time on the embedded RTOS is smaller than on both large scale RTOSs. However, the package processing rate is higher on the large scale RTOSs. Unfortunately, the authors omit to state the used hardware, which might be helpful when interpreting the results.

2.7.3 Timing Analysis on Complex Hardware

A different way of timing analyzing is presented in [Cho+13]. The authors of [Cho+13] use an integrated static timing analysis, which means that the WCET calculation includes not only the application code but also the system calls used by the application. The calculated WCET bounds for the tasks are later compared with measured values and the overestimation is discussed. What is especially interesting in this work is that the same processor as we use for this work is used. Consequently, the authors of [Cho+13] have to deal with multilevel caches and other advanced processor features. Their solution was to turn off unpredictable processor features like speculative branch prediction and implement a so-called cache damage function in order to deal with the cache related preemption delay.

Another work about statical timing analysis is done by Yan and Zhang in [YZ08]. The authors describe a statical timing analysis on a modern dual-core CPU including a L2 cache, but without an RTOS. Most recent approaches just ignores the L2 cache by simply disabling it, so all requests are missed. When ignoring caches, the WCET bound is largely overestimated. The authors of [YZ08] demonstrate a more accurate and tighter WCET bound in contrast to ignoring the L2 cache. Further, a second interesting aspect for this work is discussed in [YZ08]: A private L2 cache might give worse execution times than a shared L2 cache when executing two or more threads of a process simultaneously. This is because the threads share instructions and data.

Tomiyama *et al.* illustrate the development of a multicore capable RTOS in [THT08]. The authors describe requirements, benefits and drawbacks of multicore capable RTOS, which have been found during the implementation process. The utilized RTOS, called μItron FDMP kernel, is designed to work on an Asymmetric Multi Processor (AMP) where every core has its own memory and RTOS instance. When using AMP one core can read and write to the memory of the other cores by accessing an interconnection bus. In the evaluation the code size and the performance of the FDMP kernel, is compared with the uniprocessor kernel. The code size of the multicore kernel is about 60% greater than the uniprocessor kernel. The performance evaluation is done by measuring the runtime of two frequently used system calls in an inter- and intra-processor use case. It can be seen that all system calls on the multiprocessor kernel are slower compared to the uniprocessor kernel.

In [DD13] another comparison of multi-processor capable RTOS is done. The RTOSs are evaluated regarding the support of multicore or multiprocessor types, scalability and implementation. The focus lies on Symmetric Multi-Processing (SMP) and the FreeRTOS xcore. Simple benchmarks demonstrate the high processing power of FreeRTOS xcore. The benchmarking results of the FreeRTOS xcore shows a speed up of 1.88 on a dual-core CPU compared to a single-core CPU.

The author of [Bog13] implemented a Rhealstone benchmark on a Xilinx Zynq-7000 Extensible Processing Platform which includes a dual core ARM Cortex A9 processor and a Field Programmable Gate Array (FPGA) section, which can be configured by the user. The processor is able to run FreeRTOS in AMP mode. Unfortunately, the author utilizes only on one of the two available cores for the benchmarks. The benchmarking is done by setting the Pmod2 port to HIGH at start and LOW at the end of the benchmark. The measurement is done with a logic analyzer with a 10ns resolution. The author provides a source code of the implemented benchmark for FreeRTOS which is similar to SafeRTOS.

2.7.4 Benchmarked Metrics of Related Works

Since most of the related works consider different RTOS metrics to be important, an attempt is made to create a comparison of the related works regarding their preferred metrics. It can be seen in Table 2.2 that most the authors prefer the same metrics as defined in Rhealstone [KP89]. Special attention is paid to the context switching time, preemption time and semaphore shuffling time, which indicates that these metrics are most important to be known.

	[Oer12]	[Xu+08]	[KP89]	[THT08]	[TTN09]	[LA14]	[Kha+09]	[Wal12]	[Kim+10]	[GC11]	[Bog13]	[Mar+11]	Σ
Context switch time	x	x	x		x		x	s	x	x	x	x	10
Preemption time	x		x		x	x	x	s			x	s	8
Semaphore shuffle time	x		x		x	x	x		x		x		7
Deadlock break time	x		x						x		x	x	5
Datagram throughput time	x		x		x				x		x		5
Frequently used system service latency		x		x	x			x					4
Memory acquire and release time					x					x			2
Memory footprint		x		x	x			x					4
Heap usage		x											1
Interrupt latency	x		x				x				x	s	5
Network throughput												x	1
Peripheral function latency		x											1

Table 2.2: The twelve metrics defined for our work provides a range of possible profiling procedures for RTOS. This table shows which metrics, including non system service metrics, are implemented or considered as important by related works. A ”x” means that the referred work considers the metric as important. ”s” indicates that the interpretation of the benchmarking metric is similar to ours. This might occur if the authors of a work changed the metric. The last column of the table contains the number of related works that consider a special metric as important.

Chapter 3

Concept and Design

This chapter is based on the technical background section in Chapter 2. At first, the requirements of the RTOS profiling suite are defined, in order to see the goal of the implementation. Furthermore, the hard- and software environment is described. At next, timing analysis issues, which causes jitter on a complex hardware, are discussed. The conception of the profiling suite and its metrics is made while considering the before described timing issues. At least the concept is refined to the design of our profiling suite.

3.1 Requirements

The requirements for the RTOS profiling suite define the goals of this work. The concept refers to this list of requirements in order to show how and where these terms are achieved.

Req.1 Awareness of performance issues of complex hardware. The usage of modern hardware for real-time systems makes it necessary to adapt the approved profiling procedures for more complex hardware. Also the profiling results need to be more detailed in order to avoid misinterpretation.

Req.2 Compatibility with singlecore processors. All profiling procedures needs to be compatible with modern singlecore processors those uses a cache and other performance improvement features.

Req.3 Compatibility with multicore processors. The presence of more than one processing element makes it necessary to redesign the singlecore profiling procedures. The procedures need to consider the concurrency of such systems in order to provide meaningful results.

Req.4 Implementation of profiling procedures for RTOS metrics. A set of profiling procedures for RTOS metrics should be chosen and implemented.

Req.5 Portability to other RTOSs. The RTOS profiling suit requires an interface to the RTOS API, which covers the system calls needed to implement the profiling procedures.

Req.6 Portability to other hardware. The needed hardware features require abstraction by an interface, in order to simplify the porting between two hardware platforms.

Req.7 Extendability with other profiling procedures. The implementation of the chosen procedures can never cover all needs of a specific project. Therefore, it is required to allow the user to design own procedures and test cases for the RTOS.

Req.8 Logging information to support the evaluation of the results by the user. While profiling a system call, the profiling suite is required to store as much as possible values and deliver them to the user. The further processing and interpretation of the data is left to the user of the RTOS profiling suite.

3.2 Environment

This section states the hardware and software environment of this work. It shows which hardware and software is chosen and the arising possibilities and restrictions for profiling RTOS metrics.

3.2.1 Singlecore Platform i.MX28

In our work we use the Freescale MCIMX28EVKJ evaluation board, which includes the i.MX287 [Sem12] applications processor. The processor has an ARM926J-S [ARM08] core running at up to 454 MHz clock frequency. It also includes a 16 kB I-Cache and a 32 kB D-Cache. Both caches are four way associative with a cache line size of 8 words. The replacement strategy of the cache is either FIFO or random. The processor also includes an on chip 128 kB on-chip SRAM and a MMU with a TLB. The evaluation board also includes a 128 MB DDR2 RAM with a page size of 2 kB [Sem12].

As recommended by the authors of [Cho+13], who use a similar processor for their work, we disabled the speculative branch prediction of the processors five stage pipeline in order to make measurements more deterministic.

3.2.2 Multicore Platform i.MX6Q

As second hardware platform we use the Sabre Lite Evaluation Board [Dev14], an IMX6Q processor with four ARM Cortex-A9 MPCores [ARM08]. Every Cortex-A9 core has its own L1 instruction and data cache with 32 KB each and a cache line size of 32 bytes. Different to the ARM926J-S the default caching policy is pseudo-round-robin (PRR), which is similar to FIFO, but cheaper to implement and unfortunately, harder to predict [Rei08]. The second caching policy is pseudo random replacement. The IMX6Q contains a unified I/D L2 cache with a size of 1 MB. The L2 cache is shared between all four cores and can be accessed via the AXI bus. All cores are available in a cache coherent cluster under the control of a Snoop Control Unit (SCU), which maintains the coherency of all L1 caches. The Cortex-A9 features an eight stage, superscalar, variable length, out-of-order pipeline with dynamic branch prediction. Same as it is done in case of the i.MX28, the speculative branch prediction is switched to the more predictable static branch prediction.

3.2.3 Real-time Operating Systems

QNX Neutrino

QNX Neutrino [QNX14a] is an RTOS that is highly configurable and compatible with advanced and multicore processors. Neutrino is fully compatible to POSIX, which makes it a good choice to port our profiling suite. This is, because POSIX is used by numerous other operating systems. POSIX compatibility also makes it easy to use a standard compiler as for instance GCC [GNU14a]. Another feature is the single and multicore compatibility of QNX Neutrino. It can run either in AMP, SMP or BMP mode. Neutrino is also a SIL 3 safety standard certified RTOS.

An important fact to notice is that the Neutrino kernel is delivered in binary form, because QNX defined the RTOS to be closed source. The closed source kernel causes the RTOS profiling to be a black box testing.

SafeRTOS

SafeRTOS, made by Wittenstein [Wit14], is a commercial and also SIL 3 safety-certified RTOS based on the community driven project FreeRTOS [Fre14]. SafeRTOS and FreeRTOS are designed to fit on small scale and also advanced processors, but not on multicore systems. Hence, both have less API functions compared to other RTOS, as for instance QNX Neutrino. SafeRTOS is also not POSIX compatible, it implements own API calls [Fre14]. The SafeRTOS kernel is delivered including all source codes, which gives insight in how the kernel works.

3.3 Predictability and Timing Considerations on Complex Hardware

There are only a few works that study WCET analysis on complex hardware such as multicore processors. The reason for the complexity is, that nearly all hardware features in modern processors are designed to improve the best case and the average case execution time. However, processor designers are also urged to keep the chip size small and the production costs low. This often results in a poor predictability of the whole system. An example for this is the caching strategy, as stated in Section 2.5.1.

Not only multicore platforms cause predictability issues. Singlecore architectures often include processor features as for instance a pipeline with branch prediction or a multi level cache. All performance issues of a singlecore platform can also occur on a multicore platform. In order to meet **Req.1** it is necessary to consider side effects of complex hardware and deal with them.

3.3.1 Cache Related Preemption Delay

Description

Cache Related Preemption Delay (CRPD) is the most important factor for profiling a system that uses a cache and an RTOS with preemptive scheduling. Every time when a task is interrupted by another task, the content of the cache might be corrupted. Figure

3.1 shows a typical example of such a behavior. If the task P1 in Figure 3.1 can run without preemption, the data in cache would be fetched only once. Due to the interruption of P2 and the operating system, the execution time of P1 increases. However, not only the tasks are affected by the CRPD, but also the cached operating system data might be replaced. That means, it is not optional for real-time operating system profiling procedures to pay attention to this fact.

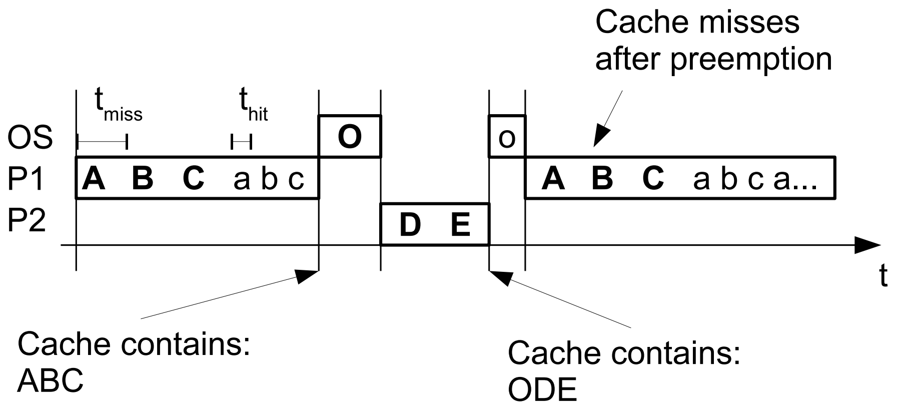

Figure 3.1: Cache Related Preemption Delay (CRPD), upper case letters means cache misses, lower case means cache hits. In this example the cache has three slots to store data. When task P1 is scheduled the first time, it fetches the three datasets A, B and C from the main memory. After some time the operating system (OS) preempts P1 and starts P2, therefore the operating system data O is loaded from the main memory and also P2 loads its data, D and E. Scheduling back to P1 takes less time than scheduling to P2, because O is still in the cache. After scheduling to P1, it is required to reload all data from the main memory which increases the runtime of the task.

Especially tasks working on a large memory block might replace nearly all cached data. A possible way to deal with that issue is to lock important peaces of code and data in the cache [Moy13]. However, in most cases the L1 cache will be too small to lock down all important peaces of code. A profiling tool for RTOS can deal with this problem in three ways:

1. Before running a profiling procedure, run a method or function that uses enough memory to invalidate data and instructions stored in the cache. This method is time consuming and complex, but the advantage is, that it can be used in combination with locking of cache lines. This technique is also called cache thrashing [Kin13].

2. Invalidate or flush the cache with hardware functions. This method is easy to implement and faster than method 1, but it can not be used in combination with cache line locking. Furthermore, hardware commands to clear the cache are only available in kernel mode, in most cases.

3. Disable the cache. As stated in [Moy13] this is done in order to receive higher predictability, but it will cause unrealistic and too pessimistic measurements.

Handling

Due to the pessimistic measurements we will focus on methods 1 and 2. This will be done by implementing cache thrashing functions similar to [Kin13]. However, the problem with methods 1 and 2 is, while calling the time measurement function or other code fragments before the to be profiled system call, it is possible that the data of the to be measured operating-system-code is also loaded into the cache. This depends on the page size of the memory. A RAM page of size 2KB, for instance, might contain parts of the profiling suite and also parts of the RTOS. In order to avoid such situations, timer and flash-invalidation functions, are implemented directly in the profiling suite. No included RTOS functions are used for these purposes. Additionally the code of the profiling suite is separated from the RTOS code using a linker script.

3.3.2 Branch Prediction of Pipeline

Description

For a non-pipelined architecture it is possible to add up the instructions to get the worst case runtime. A pipeline, in contrast, is able to improve the average runtime of code segments without branches. In the case of a branch, the pipeline can statically predict, if the branch will be taken or not. In order to further enhance the average runtime, some pipelines utilize a speculative branch prediction. However, speculative branch prediction might mispredict a branch result and fetch unnecessary instructions which might not be cached. This is hard to predict and should not be used in a hard real-time system [Cho+13].

Handling

On our processors, the speculative branch prediction is activated by default. In order to achieve a better predictability, the speculative branch prediction is replaced by a static branch prediction function.

3.3.3 Timing Anomalies and Domino Effects

Description

Timing anomalies are counterintuitive influences of the local execution time to the global execution time [Wil+08]. An example for this is the scheduling anomaly [Cul+10], where a shorter runtime of a single task can lead to a global longer execution time of the task set.

Domino effects are counterintuitive situations in caches. Missing information about the actual content of the cache can lead to wrong assumptions of the runtime. In Figure 3.2 an example of a FIFO cache can be seen. It is known that numerous caching strategies exhibit domino effects, as for instance: FIFO (iMX28), Pseudo-FIFO (iMX6Q), random (IMX28 and iMX6Q) and Pseudo Last Recently Used (PowerPC 755). It is also known that Last Recently Used (LRU) does not exhibit domino effects.

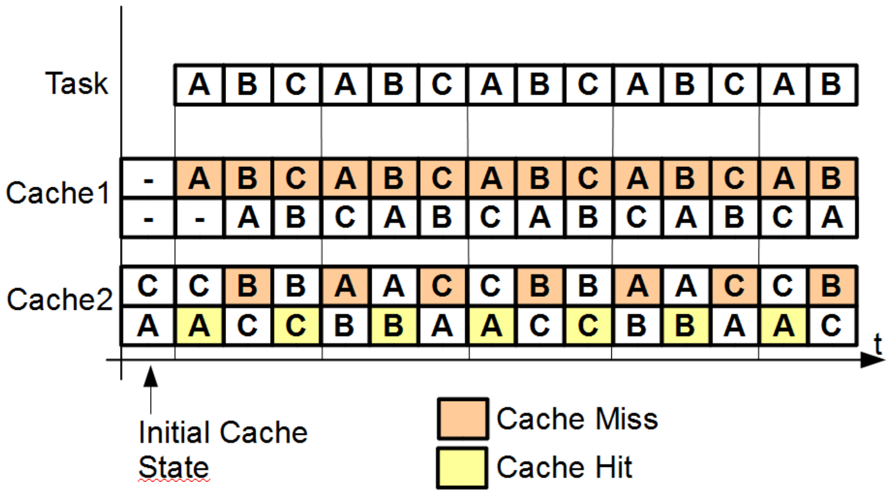
Task
A B C A B C A B C A B C A B
Cache1
- A B C A B C A B C A B C A B
- - A B C A B C A B C A B C A
Cache2
C C B B A A C C B B A A C C B
A A C C B B A A C C B B A A C
t
Initial Cache State
Cache Miss
Cache Hit

Figure 3.2: Domino effect of a 2-way-FIFO cache. The worst case execution time depends on the initial state of the cache. Intuition would say: due to the access sequence the cache will get N misses. Contrary to intuition, depending on the initial cache content the cache misses are reduced to N/2. Adopted from [Ber06]

Handling

For a profiling suite for RTOS it is hard to enforce the occurrence of such effects. However, when profiling RTOS or applications on an architecture that exhibits timing anomalies the profiling suite is required to keep in mind that these effects will increase the jitter of measurements. For further informations on timing anomalies and domino effects we refer to [Cul+10].

3.3.4 Translation Lookaside Buffer

Description

A TLB can be found in processors that support paging. It is used to speed up the translation from virtual to physical addresses calculation. It works like a cache for virtual addresses in the MMU, but as every other cache the size is limited. That means, if a virtual address is not in the TLB, it needs to be fetched from the main memory. With an increasing number of processes with separated address space, the performance loss, caused by TLB misses, increases.

Handling

While real-time systems deal with this issue in various ways, a soft real-time system may just ignore this fact. However, a hard real-time system needs to pay attention to the TLB. A possible way is to share the address space between all tasks. In literature this operating mode is called single process mode [Moy13]. Another, more drastic, way is to turn off the MMU and access physical addresses directly. A third option is to lock critical addresses in the TLB in order to guarantee TLB hits on these specific virtual address. On SafeRTOS, we use the single process mode to deal with the TLB cache. In contrast to the singlecore, a multicore hardware, as for instance the IMX6Q, every core has its own MMU and TLB. Due to this, migration of tasks between cores enforces the migrated task to reload all used virtual address. Therefore, migration is turned off and all profiling tasks are pinned to specific cores.

3.3.5 Migration

Description

Migration can only appear on a multicore architecture. It means that the processing element of a task can be switched during runtime. This is used most frequently in SMP and partial in BMP operating mode. The benefit of migrating tasks between cores is a better load balancing, but the consequence is that migration costs time. All cached data of a task gets removed and needs to be fetched again. This timing penalty increases with the separation of cores, which means a shared L2 cache is better for migration than a L2 cache per core. Hard real-time systems often abstain from migration, because the latency is hard to predict and therefore it is also hard to find a realistic upper bound.

Handling

As already mentioned in the last section about the TLB caches, the migration is turned off. This is also necessary to implement the profiling procedures in a way such that the scenario can be simulated meaningful.

3.3.6 Intra- and Inter-Processor Communication and Sharing

Most performance issues on multicore architectures arise as a result of sharing data and resources. Sharing data or communicating on a singlecore processor, or within one processor core, is simple and fast. However, if the communicating tasks reside on different processors or cores, it depends on the memory architecture and the memory bus how fast the messages can be passed to the waiting process. The same applies also for locking of critical sections and other inter-processor actions.

In a communication between two tasks the receiving task waits for the sender to send data. The job of the operating system is then to wake up and schedule the receiving task. How a multicore RTOS handles this depends on the operating mode. In a single RTOS instance mode, as for example SMP or BMP, the RTOS might reside on a different core than both of the communicating tasks. In contrast to that, an AMP operating mode, where one RTOS instance per core is running, inter-processor communication and synchronization is also an inter-RTOS communication and synchronization.

3.4 Selection and Adaption of Profiling Procedures for RTOS Metrics

After comprehensive research in Section 2.7 we choose five benchmarking metrics for our work, because this five metrics are implemented by the most related works.

1. Context Switch Time
2. Preemption Time
3. Semaphore Shuffling Time
4. Deadlock Break Time
5. Intertask Message Latency

In order to satisfy our requirements **Req.2** and **Req.3**, which means compatibility with single and multicore processors, it is necessary to adapt the chosen metrics. The definition of the metrics for singlecore processors can be seen in Section 2.4.2. In this section, the adaption and extension for complex singlecore and multicore processors is discussed.

3.4.1 Realization of Measurements with Cache Related Preemption Delay

The cache related preemption delay is always simulated in the same way during our profiling procedures. Before starting a procedure iteration, a method is called to clear the cache. It is up to the user to decide when and how often the cache is cleared. The cache clearing methods are not explicitly stated in the pseudo codes of the metrics.

In case of a BMP operating mode the instance of the RTOS might reside on another core than the profiling task. The cache clearing method is called by the profiling task and affects only the executing core. Since that, the user of the profiling suite needs to ensure that the operating system code will not be removed from the cache.

It should be noted, that there are two different methods to read the time in the pseudo codes of the metrics: *getLocalTime()* and *getGlobalTime()*. The local time means the execution time measured on the active core. Global time means a synchronized time over all cores. Further information on this will be provided in Section 3.5.3.

3.4.2 Context Switch Time

The context switch time is implemented similarly on singlecore and multicore systems. This metric is affected by numerous factors. On a singlecore system, this time is mainly influenced by the cache. If the cache is empty and all code and data has to be loaded from main memory the time will be longer. On a multicore system, simultaneous context switching on all cores can cause higher run times, if a single instance of an operating system maintain all cores. For instance this is the case in a BMP operating mode of the RTOS. An illustration of this situation can be seen in Figure 3.3.

Listing 3.1: Task P1 yields and P2 starts running

```
wait_ready();
while(true){
  t1 = getLocalTime();
  yield();
}
```

Listing 3.2: Task P2 starts and yields immediatly

```
wait_ready();
while(true){
  t2 = getLocalTime();
  yield();
}
```

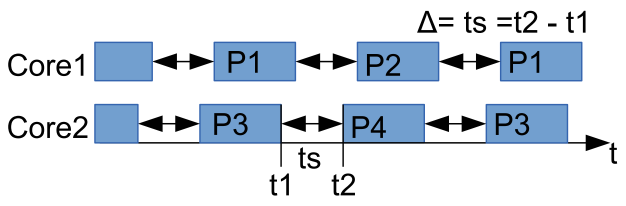

Figure 3.3: Context switch time on a multicore system is almost the same as on a single core system. The only extension is that the profiling procedures runs on every core simultaneously.

In Listing 3.1 and 3.2 the procedure for tasks P1 and P2 is depicted. The tasks P3 and P4 work the same way. The two methods *wait_ready()* represent a barrier, in order to wait until both tasks are ready. It should be noted that it is necessary to guarantee that P1 and P3 are scheduled before P2 and P4.

3.4.3 Preemption Time

Measurement of the preemption time on a multicore does not differ from the singlecore measurement. This is, because it does not make sense to process an interrupt on another core than the ISR. It would induce additional overhead to pass the data and control flow to another core.

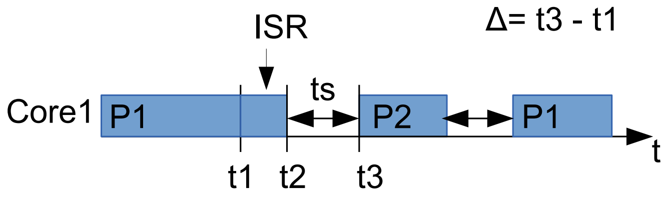

Figure 3.4: Preemption time is exactly the same on singlecore and multicore architectures. The time t_1 is measured in the ISR and t_3 is measured when P2 starts running.

Listing 3.3: Task P1 is interrupted by an interrupt. The corresponding ISR sets the higher prior task P2 runable

```
P1:
wait_ready();
fire_interrupt_manually(); // Or do useless work
    and wait for a timer interrupt

ISR:
t1 = getLocalTime();
wakeup(P2);

P2:
wait_ready();
sleep();
t3 = getLocalTime();
```

As it can be seen in Listing 3.3, at first in this scenario P1 is running and triggers an interrupt. This could be done by triggering a software interrupt or by waiting for a hardware interrupt as for instance a cyclic timer interrupt. After that the task P1 is interrupted and the ISR is called, which wakes P2 up.

3.4.4 Semaphore Shuffle Time

Interprocess communication is especially in a multicore environment an important factor. For an RTOS in AMP or BMP operating mode it is necessary to provide intra and inter-processor synchronization possibilities. Measuring the intra-processor synchronization can be done by the singlecore semaphore shuffling time profiling procedure. Profiling the interprocessor semaphore shuffling time needs to measure the same timestamps as the intra-processor variant, but the measurement points are distributed over two cores. That means, it is necessary to acquire global timing states in contrast to the both profiling procedures mentioned before. Also the assembly of the procedure is more complex, because of the additional communication between the tasks. A timing diagram of this profiling procedure can be seen in Figure 3.5.

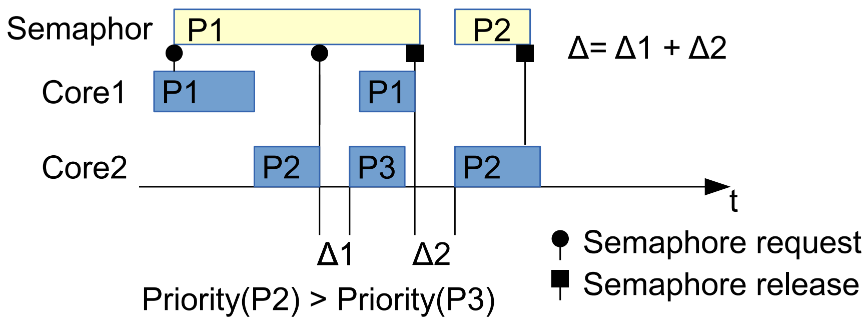

Figure 3.5: Semaphore shuffle time is implemented as an interprocessor synchronization. P1 locks the semaphore S1, which is also needed by P2. P3 is a helper process to measure times and wake up P1 when P2 has tried to acquire S1.

Listing 3.4: Task P1 acquires the semapore S1 and waits for the signal of P3 to release it.

```
wait_ready();
lock(S1);
wakeup(P2);
sleep();
t3 = getGlobalTime();
unlock(S1);
```

Listing 3.5: Task P2 tries to acquire the semapore S1 and fails. It has to wait for P1 to unlock it.

```
wait_ready();
sleep();
wakeup(P3);
lock(S1);
t1 = getGlobalTime();
unlock(S1);
```

Listing 3.6: Task P3 is a helper task to synchronize the tasks P1 and P2. It has lower priority than P2.

```
wait_ready();
sleep();
t2 = getGlobalTime();
wakeup(P1);
```

Listings 3.4, 3.5 and 3.6 show how the three processes work together. At first, all processes need to wait until all of them are ready. Then P1 acquires the semaphore and wakes up P2. P2 first wakes up P3 and then tries to acquire the semaphore, because P3 has lower priority it will not interrupt P2. When P2 waits for the semaphore, P3 starts running and notifies P1 to release the semaphore. P3 needs to acquire a global system time at this point. How this can be done is stated in Section 3.5.3.

3.4.5 Deadlock Break Time

Deadlock break time is similar to the semaphore shuffling time in singlecore and multicore variants. The main difference is the presence of a task, which prevents the lower priority task from releasing the semaphore. This profiling procedure requires the RTOS to have a priority inheritance protocol being activated. If there is no such protocol active, the test will deliver false results. This is, because P2 will run until it has finished, which means that the measured time will be too large. This behavior can be seen in Figure 3.6. If it is known that the RTOS does not support priority inheritance, this profiling procedure should be deactivated.

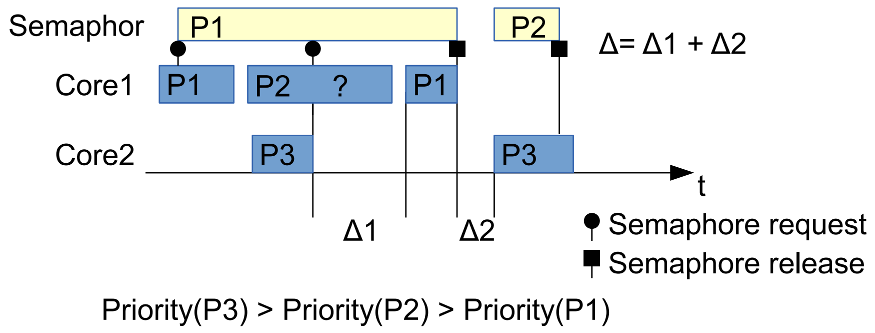

Figure 3.6: Measurement of deadlock break time for multicore architectures is implemented similar to the singlecore variant. The only modification is that P3 runs on a different core than P1 and P2.

Listing 3.7: Task P1 acquires the semapore S1 and gets preempted by the higher prior task P2.

```
wait_ready();
lock(S1);
wakeup(P2); //dispatch!
t3 = getGlobalTime();
unlock(S1);
```

Listing 3.8: Task P2 wakes up the highest priority task P3 on core 2 and utilizes the core as long as it gets preempted by a higher prior task.

```
wait_ready();
sleep();
wakeup(P3);
while(!signal_exit){
  do_something;
}
```

Listing 3.9: Task P3 is the highest prior task which wants to acquire the semaphor S1.

```
wait_ready();
sleep();
t1 = getLocalTime();
lock(S1);
t2 = getLocalTime();
unlock(S2);
emit(signal_exit);
```

In Listing 3.7 P1 locks the semaphore S1 and wakes up the higher prior P2. At this point it is necessary to make sure, that the operating system schedules to the higher prior task P2 and interrupts P1 reliably. The behavior of P2 is shown in Listing 3.8. P2 is required to simulate a high work load, in a way that the RTOS will not switch context back to P1. However, if the deadlock cannot be resolved or resolving takes too long, the task P2 needs to be signaled to stop working. In Listing 3.9 the procedure of P3 is shown. It should be noted, that P3 is responsible to emit a signal to P2 to stop working and that measurement of time is made using the global time.

3.4.6 Intertask Message Latency

Singlecore and multicore implementations of this profiling procedure are almost equal. The only difference is that the communicating tasks in a multicore system reside on different cores, as it can be seen in Figure 3.7. That means further, messages have to be passed between either only two cores or between two cores and two operating system instances. In any case, the measured times need to be taken on different cores, which makes it necessary to read the global time.

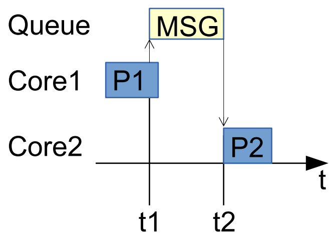

Figure 3.7: Intertask message passing latency in a multicore architecture is measured by passing the message between two cores.

Listing 3.10: Task P1 sends a message to P2

```
wait_ready();
wait_for_receiver();
t1 = getGlobalTime();
send(MSG);
```

Listing 3.11: Task P2 receives a message from P1

```
wait_ready();
MSG = receive();
t2 = getGlobalTime();
```

In Listing 3.10 and 3.11 a normal procedure of task P1 and P2 is shown. This profiling procedure requires task P1 to wait until P2 has called the receive function reliably. This is necessary, because when P1 sends the message to a queue with no receiver, it might take some time until P2 calls the receive method, which will cause an inaccurate measurement. The singlecore profiling procedure handles this issue by rising the priority of P2 to a higher level than the priority of P1.

3.5 Profiling Suite for RTOS Metrics

In this section the design of our profiling suite for RTOS metrics is described. The focus lies on the fulfillment of our requirements defined in Section 3.1. At first an overview of the software architecture and the control flow is provided. After that, it is described how runtimes are measured, how the suite is ported to other systems and how the suite can be extended with other profiling procedures.

The intent of developing this profiling suite is to provide a tool to measure the execution times of RTOS services in a special situation, as for instance a special hardware, a specific configuration of the hardware or the RTOS. It is not designed to say an RTOS is better or worse than another in general. Instead it is designed to show how good the tested RTOSs perform in the users project specific scenario. The profiling suite is not claimed to cover all possible RTOS metrics. If there are other necessary system calls to be measured, it is intended that the user implements the missing metrics.

3.5.1 Overview

In Figure 3.8 the architecture of our profiling suite is represented. Most RTOS vendors supply a C or a CPP version of their code. Due to this fact, the profiling suite is written in C language in order to support as much as possible RTOS. Another important fact of the implementation is, that the interfaces to hardware and operating system are defined by preprocessor macros as recommended in [Oer12]. The advantages for using macros will be stated in Section 3.5.4 and 3.5.3.

In Figure 3.8 an overview of the profiling suite is provided. The elements of this overview are described below. Figure 3.9 illustrates the sequence of initialization and a single iteration of a metric in a profiling procedure.

ProfilingTask is a task that controls the whole process of calling profiling procedures and delivering results to the user. However, before starting this task, it is necessary to initialize the hardware and the RTOS. When the task starts running, it initializes the *ProfilingAPI*. After that the task calls a function named *ProfilingSequence*. This function contains a list of profiling procedure calls handled by the *ProfilingAPI*. The intention of the *ProfilingSequence* function is to vary the order and parameters of profiling procedures. It should be noted that this task need to have all necessary permissions to run the profiling procedures in its context.

ProfilingAPI contains all necessary functions for the profiling task and the implemented profiling procedures. In the initialization function the *ProfilingManager* and other internal variables are initialized. Every profiling procedure is started by calling the according function in the *ProfilingAPI*. Also every profiling procedure uses the logging functions of the *ProfilingAPI*.

ProfilingLogger contains a global data storage for all profiling results. When a profiling procedure is started, the *ProfilingAPI* calls the *createLog* function. This resets all data arrays and counters. When a new timing event needs to be stored the profiling procedure calls one of the *logValue* functions also via the *ProfilingAPI*. At the end of a profiling procedure, the *printLog* function is called by the *ProfilingAPI*, in order to avoid extensive storage utilization.

ContextSwitchTime, PreemetionTime and OtherMetric are the implementations of the profiling procedures for RTOS metrics. These implementations are handled by the *ProfilingAPI* and uses the functions defined in RTOSMacros and HardwareMacros, respectively the implementation of the macros in the related port. The profiling procedures implementation is required to not return, until the last related task has finished and is removed from the list of tasks or set inactive.

RTOSMacros and RTOSPort defines the interface to the RTOS. The macros define the necessary functions in order to be able to implement the profiling suite. The RTOSPort copies these macros and defines the according functionality.

HardwareMacros and HardwarePort defines the interface to the hardware. The macros define functions for initializing the hardware, acquiring actual system times and other functionalities. The HardwarePort copies these macros and defines the according functionality.

3.5.2 Profiling Procedures for RTOS Metrics

The profiling procedures are implemented as standalone functions in exclusive c and h files. An example of such an implementation can be seen in Figure 3.8, namely *ContextSwitchTime*, *PreemtionTime* and *OtherMetric*. *OtherMetric* means, that at this point in code it is possible to place other profiling procedures, as for instance other singlecore and exclusive multicore profiling procedures.

For our implementation, and to fulfill **Req.4**, we implemented the five singlecore and four multicore profiling procedures as described in Section 3.4. It should be noted that there are nine different profiling procedures implemented, because of the difference between single- and multicore metrics. The choice, which metrics should be profiled, is made in the *ProfilingTask*s function *ProfilingSequence*.

3.5.3 Measurement

In [Str14] we did a comprehensive analysis of available measurement techniques regarding the required time to take a measurement and the deviation of that time. We stated in [Str14], that measuring the runtime of executing code gets more complicated the more advanced the hardware is. Therefore, time measurement is prone to error. As stated in Section 2.4.4, it is common to use an internal timer of the CPU or toggle a GPIO pin to measure times. That is a good approach as long as the CPU clock is slow enough to get meaningful results. The ARM926J-S CPU clock runs at 454 MHz, but the fastest timer on this CPU runs at a maximum frequency of 24 MHz. That means, the accuracy decreases with rising frequency gap between CPU and timer. The same holds when using GPIO pins for measurement, it depends on the GPIO controller how fast a pin can be toggled.

In contrast to the ARM926J-S CPU, the Cortex-A9-MPCore CPU contains a global timer, which is clocked with the half frequency of the CPU [ARM10]. Due to that, measurement accuracy is better than on ARM926J-S. When using a timer in general, it is required to ensure that the RTOS does not use the same timer.

Reading a timer value is not a cycle accurate process since reading the timestamp shows different runtimes every time it is requested. However, analysis of execution time

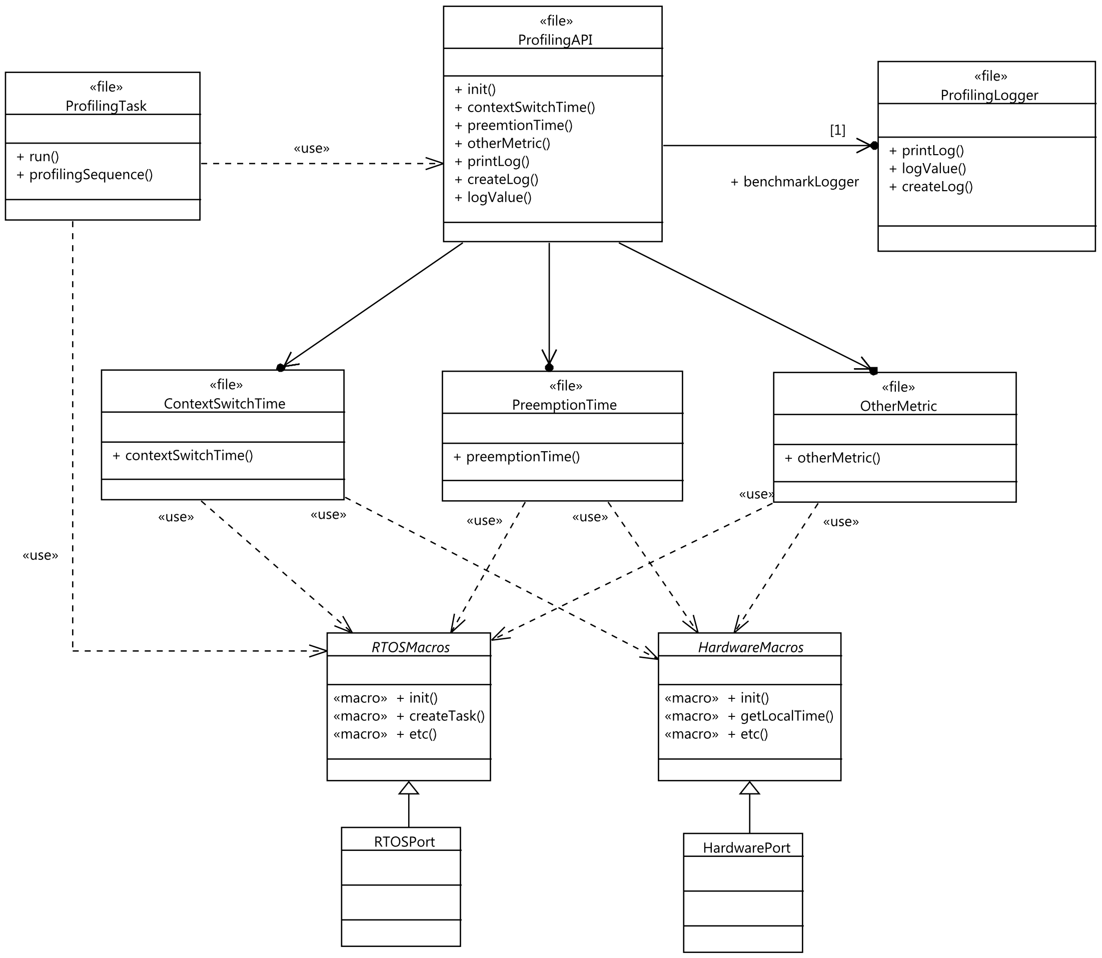

Figure 3.8: This image shows an outline of the architecture of our profiling suite. The suite is implemented in C language, which means there are no classes. In this diagram a box means a combination of a C language header file and source file. Information about the program flow is provided in Figure 3.9.

is needed by many applications, CPU vendors added a so-called Performance Monitoring Unit (PMU) to their products [ARM10]. A PMU provides counters to gather statistics on the operation of the processor and memory, which will further be called events. An event can be for example a cache miss or a cache hit. For further information on events the reader is referred to [ARM10].

The PMU contains also a cpu-cycle-counter which allows cycle accurate measurement of execution time. On a Cortex-A9, this counter can be within an almost constant amount of CPU cycles, because the PMU is located directly on the processing element. No memory bus or other memory transfer is necessary to get the data. However, this is problematic in some of our profiling procedures where the timestamp has to be gathered from another core.

In [Str14] we recommended evaluating all available measurement techniques on the actual system, in order to tell which technique is suited for a certain measurement. Fur-

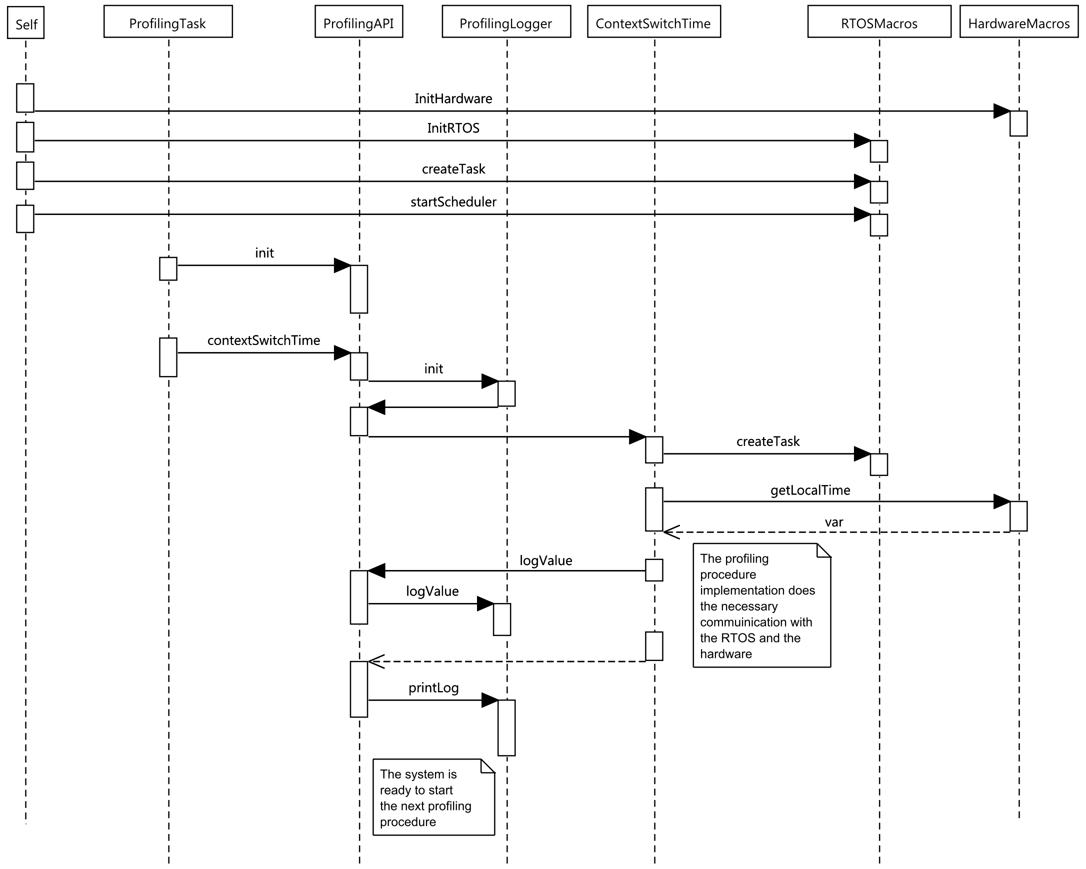

Figure 3.9: The sequence diagram of the profiling suite. This diagram shows the set up procedure and a single iteration of a profiling procedure. Self in this case means the main procedure which is called at processor startup.

thermore, we stated in [Str14], that for our environment, QNX Neutrino running on the SabreLite [Dev14] evaluation board, a well-suited way to do singlecore measurements is to utilize the PMU cycle counter. Multicore measurements should be done with the global timer, in order to reduce the deviations to a minimum.

3.5.4 Portability

The requirements **Req.5** and **Req.6** demand portability between both, hardware and software platforms. Therefore we use preprocessor directives defined by the ISO C99 standard [JTC99]. That means, the profiling suite requires a developing environment that is compatible to standard C and is compiled with the GCC [GNU14a].

The profiling suite provides two files RTOSMacro.h and HardwareMacro.h. These files contain a list of preprocessor function definitions, which are used by the profiling suite. The preprocessor functions are links to RTOS functionalities and hardware commands. When porting the profiling suite to a new hardware or a new RTOS the user needs to provide two header files which link to the correct RTOS and hardware functionalities. Such implementations can be seen in the following examples:

RTOSMacro.h:

$\#define\ PORT\_YIELD() \quad yield()$

PORT_YIELD() will be used by the profiling suit and is replaced by the RTOS function *yield* by the preprocessor.

HardwareMacro.h:

$\#define\ PORT\_GetLocalTime() \quad get\_cyclecount()$

PORT_GetLocalTime() will be used in the profiling suite and by the preprocessor replaced by the inline function *get_cyclecount().*

If it is necessary to do more than one function call in order to implement the required functionality, it is possible to write a user defined function and set the link to this function.

3.5.5 Extendability

It is not possible to cover all necessary RTOS metrics to ensure all users of the profiling suite are satisfied. Therefore, an extensible design is made in order to provide the possibility for the user to extend the profiling suite with own metrics.

In order to satisfy the requirement **Req.7**, every metric is implemented in a unique file and installed in the profiling suite by adding it to the *ProfilingAPI* and the call it in the *ProfilingTask*. The user of the profiling suite can extend it by following the steps described here:

1. Create a header and a source file for the new metric.
2. Implement a main function for the metric, which takes the configuration as parameters.
3. Add the header file as an include to the *ProfilingAPI*. Implement a function with the same name in the API that forwards a call to the implementation of the metric. If not done already in the metrics implementation, before and after the metric, the *ProfilingLogger* has to be notified.
4. Call the new Metric with the desired parameters in the *profilingSequence* function of the *ProfilingTask*.

3.5.6 Logging and Output

The *ProfilingLogger* is the central element for logging data. It consists of four global data arrays and four separated counters. The size of the data arrays is static and has to be known beforehand. This is done, in order to avoid dynamic memory allocation or list operations. A new value can be stored in constant time.

When creating a new log, all counters are reset and the arrays are filled with zeros. Further, the user provides a number of timestamps for the active profiling procedure. After that, the *ProfilingLogger* is ready to run and accepts new values delivered by the function *logValue<N>(long value)*, where N stands for the numbers one to four. Furthermore, it will monitor the incoming values for the correct sequence. The profiling procedures are responsible for counting up the counters and delivering values at the correct time. This is done in order to provide better flexibility for custom profiling procedures.

When all values to capture are stored in the data array and the profiling procedure ends, the *ProfilingAPI* calls the *printLog()* function. This function generates a human readable output in form of a string and sends it to an output device. The output device is defined in the RTOSMacro.h file. It could be every possible output device that supports serial streaming as for instance a UART module. The string can be formatted either in a statistical evaluation format for debugging, or alternative a Matlab [Mat14] compatible format for further evaluation on the target PC. This is done in order to fulfill **Req.8**.

3.5.7 Evaluation

Req.8 states, the user of the profiling suite should be able to evaluate the captured values by using the raw data. For this work, the raw data is delivered in a Matlab script compatible format and is evaluated using Matlab functions. However, the raw data is, dependent on the used measurement technique, biased by the time to read a timestamp and the related deviation [Str14]. As recommended in [Str14], the bias and the deviation are estimated for every measuring point in the profiling suite by measuring the time to take a timestamp. The behavior of the measuring technique is approximated by a Gaussian bell curve using the mean value and the standard deviation.

If the deviation of the measurement technique is smaller than a certain threshold, no approximation via a Gaussian bell curve is needed. The threshold can be defined as needed, but if the deviation is smaller than one CPU cycle, as it is in case of the PMU, such an approximation would reduce the accuracy and is therefore not recommended [Str14].

For our evaluation we subtract the average runtime of the measurement technique from the measurement point value. If the measurement technique exhibits deviations, we fit a Gaussian bell curve with a size of three times sigma to the measurement points value. According to [Bar07], three times sigma covers more than 99% of the values. When drawing the overall curve, every measuring points Gaussian bell curve is added up for every sample. After that, the overall curve is normalized in order to provide a probability density function. All calculations, such as the expected value and the standard deviation are calculated, using the formulas for probability densities found in [Sta08] and [Bar07].

Chapter 4

Implementation

This chapter describes the implementation of the profiling suite as designed in the last chapter. The configuration of SafeRTOS and QNX Neutrino are described and the structure of the profiling procedures is stated. Not all profiling procedures are described in detail, only special implementation notes are stated here. For further information on the implementation the reader is referred to the source code of the profiling suite.

4.1 Overview

Basically, the implementation is done in standard C and uses as much default configurations as possible. This means default configurations of RTOS and the hardware are changed as little as possible. The default configurations are seen as the users desired configuration. By avoiding of numerous changes in the configurations, it is easier for the user of the RTOS profiling suite to add project specific configurations. There are only a few configurations that have to be done in order to use the included profiling procedures. Furthermore, the profiling suites implementation does not contain extensive error handling methods, because different RTOS have different ways to handle errors.

4.1.1 File Structure

Basically all methods of the RTOS profiling suite are in a folder called *profilingsuite*. As shown in Figure 4.1, this folder resides in the root folder of the project. The root folder contains at least two files, which should be implemented by the user, *main.c* and *profiler_config.h*. The file *main.c* contains the startup procedure and the setup of the *profiling_task*. *profiler_config.h* tells the system which hardware and RTOS port should be used.

The subfolder *profilingsuite* contains the *profilingAPI*, *profilingLogger* and the *profilingTask* implementations. This three implementations are the heart of the profiling suite. In addition to that, there are two more folders: *port* and *procedures*. The *port* folder contains two important files *port_hardware.h* and *port_rtos.h*. These files use the definitions in *profiler_config.h* to decide which RTOS and hardware porting should be used. The profiling *procedures* and the *profilingTask* uses these header files to implement their functionality.

```
Main Folder of the Project
├── profiler_config.h
├── main.c
└── profilingsuite
    ├── profilingAPI.c
    ├── profilingAPI.h
    ├── profilingLogger.c
    ├── profilingLogger.h
    ├── profilingTask.c
    ├── profilingTask.h
    ├── port
    │   ├── port_hardware.h
    │   └── port_rtos.h
    ├── hw_imx28
    │   └── ...
    ├── hw_imx6q
    │   ├── port_hardware_mx6q.c
    │   └── port_hardware_mx6q.h
    ├── rtos_qnx
    │   ├── port_rtos_qnx.c
    │   └── port_rtos_qnx.h
    ├── rtos_safertos
    │   └── ...
    └── procedures
        ├── contextswitchtime.c
        ├── contextswitchtime.h
        ├── ...
        └── multicore
            ├── mccontexswitchtime.c
            ├── mccontextswitchtime.h
            └── ...
```

Figure 4.1: Overview of the file structure of the profiling suite

In the subfolders of *port* the hardware and RTOS macros are defined and the necessary functions are programmed. When porting a new hardware or RTOS, one of the present header files should be copied and edited.

The *procedures* folder contains the implementations of all profiling procedures for RTOS metrics. A metric consists of a header file with a single function prototype and a C file with the implementation of this function and also the additional procedure implementations.

4.1.2 Basic Configuration of the Profiling Suite

Configuration of the RTOS profiling suite is straight-forward, provided that porting to hardware and RTOS has already been done. There are three files where changes has to be made. The first one is the *profiler_config.h* in the root directory. In this file, the path to the hardware that should be used and RTOS port header files are defined. This information is used by *port_hardware.h* and *port_rtos.h* to include the correct header files in the procedure implementations. The second file to configure is *profilingTask.c*. In this C file, there exists a function called *profilingSequence()* that implements, as the name tells, the sequence to call the procedures. The user has to choose which and how often a procedure should be called by commenting or uncommenting a profiling procedure call. The maximum number, how often a metric can be tested, is defined in the file *profilingLogger.h*. All predefined profiling procedures runs as often as possible, which is limited by the number in *profilingLogger.h*.

4.1.3 RTOS and Hardware Configuration

Porting a new hardware or a new RTOS to the profiling suite is done by copying a folder of a present hardware or RTOS port and edit it. All necessary macro definitions need to be redefined by the new functions, as for instance priority levels, desired cores, measuring methods and numerous more. Which functions should be set and how is explained in the following sections.

Another also important step is the setup of the project for the target platform. At first the file containing the main function has to be edited, in order to start up the profiling task at a desired priority level. Therefore, project specific hardware and RTOS configurations have to be defined in this file. The profiling suites hardware and RTOS ports only contain the configurations necessary for their execution. For instance no clock gating or other processor features are manipulated inside the ports.

The profiling suite is a portable peace of code, which means, there are implementations of different hardware and RTOS inside the project. Those implementations may cause build errors. Therefore, it is required that unnecessary ports are excluded from the build. That can be done for instance by using makefiles or by instructing the Integrated Development Environment (IDE) to leave out these folders.

4.2 Profiling Procedures

This section gives an introduction how the profiling procedures are setup and how they work. Here, we provide details about the profiling procedure as illustrated in Figure 3.9.

Further, implementation details of special profiling procedures are stated in order to give the reader a better overview of the system.

4.2.1 Startup

All profiling procedures are started the same way, but before that can happen, the *profilingTask* needs to be set up. The *profilingTask* sets its own permissions, priority and deactivates migration. After that, the *profilingSequence* function is called. From there, the profiling procedures are started via the supposed API functions. The API initializes or reinitializes the *profilingLogger*, such that all counters and data arrays are filled with zeros. In the next step, the desired profiling procedure is called, using the function prototyped in the procedures header file. Within that function, the tasks used for the procedure are configured and set up in a three-way configuration. All that happens in the context of the *profilingTask*.

4.2.2 Three-Way Procedure Task Setup

Every present profiling procedure uses multiple tasks to profile the respective metric. The context of the *profilingTask* is only used to start the profiling procedures tasks. Creating, configuring and starting a task is different on numerous RTOS. POSIX compatible RTOS as for instance QNX Neutrino starts a task by using *pthread_create* and configures the tasks properties in its own context. Or, if the creating task has a higher priority than the default priority of the created task, it is possible to do the configuration of the new task in the context of the creating task, in this case the context of the *profilingTask*.

SafeRTOS and FreeRTOS do not implement POSIX or other standard API. The task creation, configuration, starting and also deleting are handled in the context of the creating task via configuration structures. This method is different from POSIX and needs special attention.

Therefore, the task configuration in our present profiling procedures is done in multiple ways. To be more precise in three different ways: Configuration via struct, before task has been created and after task creation. It is up to the user of the profiling suite to decide which way of configuration is best suited for the present RTOS.

4.2.3 Locking

After a task has been created and configured, it is necessary to wait until all tasks of a profiling procedure are ready. In addition to that, it is also important that all tasks start in the correct order. This is achieved by using barriers and locks. A barrier is a locking mechanism where a number of tasks has to get to a point in code and wait there. After all tasks have arrived they become runnable the same time, but not necessarily in the correct order. Hence, a second locking mechanism has to be applied to guaranty the correct order of execution.

This locking mechanism is implemented using binary semaphores instead of mutexes. That is, because not every RTOS implements a mutex method, as for instance SafeRTOS. The starting order of tasks is different in nearly every profiling procedure. When implementing a new profiling procedure, the user has to handle the locking. It is also necessary for the user to mention that locking mechanisms may corrupt measurements, because the time to leave a locking mechanism might unintentionally be measured.

4.2.4 Running

A ready to run task of a profiling procedure repeats testing and measuring N times. The task is responsible for managing the counter of the active measurement and storing the timestamp to the correct place in the profiling logger. This high responsibility leads to a higher risk of making an error. However, this is also an advantage, because it shows errors made within the construction of the profiling procedure. Furthermore, it gives more freedom for the design of profiling procedures.

4.2.5 Finalizing and Clean Up

After all measurements have been finished, it might be necessary to do some clean ups, as for instance deleting tasks. This can be done either in the context of a task or after the control reaches the *profilingTask*. The *profilingTask* waits on a barrier until all tasks finished their execution. At this point it is possible to delete tasks from the task queue. After that the control returns to the calling *profilingAPI*, where the function *printLog* is called. The function *printLog*, as the name tells, prints the values stored in the log, in order to reduce the required amount of memory used by the *profilingLogger*. After that, the profiling suite is ready for the next procedure.

4.2.6 Preemption Time

The profiling procedure preemption time is one of the most complex procedures to be implemented. The intention of the procedure is to have a heavy work load, but a low priority task that is interrupted by a higher priority task that becomes runnable. There are two possible strategies to implement this scenario. The first is to start up the low priority task and trigger an interrupt manually. The ISR sets the higher prior task runnable and the scheduler switches to this task. However, this strategy might be problematic, because the RTOS might not allow triggering an interrupt by software. A possible workaround for this issue is to toggle a GPIO pin to trigger an interrupt via another GPIO pin. As said in [Str14] toggling a GPIO pin also takes time, which will influence the measurement.

The second strategy to implement this scenario is to configure a cyclic timer to trigger an interrupt every time interval. The benefit of this method is that the high workload task can work without any need to be disturbed. Furthermore, it is possible to configure a timer in a way such that it resets itself when it has reached a given limit. That makes it possible to find out how long it takes from the moment when the interrupt occurs until the hander task gets in control, by reading the actual timer value.

This strategy can also be used to simulate a wakeup scenario of a high priority task. This means, if the high prior task is sleeping until its next periodic start, the RTOS wakes the task up within the RTOS own systick timer interrupt. The wake up can be recreated by attaching to the systick interrupt of the RTOS. The present implementation uses the systick interrupt in both ports, SafeRTOS and QNX Neutrino.

4.2.7 Message Passing

Message passing, as well as the preemption time procedure, has special requirements for implementation, because message passing is implemented differently on numerous RTOS. SafeRTOS implements the message passing by using kernel maintained queues [Wit12]. A new message queue is created and the receiver attaches to it, in order to wait for a message to receive. The sender creates the message and pushes it on the queue.

QNX Neutrino in contrast to that provides numerous ways to implement message passing in an application. Native message passing methods are known to be fast and reliable [QNX14a]. However, we decided to use the standard POSIX message passing [QNX14c] method for our implementation. We choose this, because the portability of the profiling suite is better when preferring standard POSIX methods instead of native methods.

4.3 Abstractions

As already stated, the abstraction of the RTOS and hardware is done by preprocessor macros. Two abstraction classes are made, one for the RTOS and one for the hardware. Both contain a list of preprocessor macros that has to be defined, in order to implement the profiling procedures.

The preprocessor of the compiler translates those macros to code, it can be seen as a rule, which command should be used where in the code. Compilers as for instance the GCC [GNU14a] are able to run only the preprocessor and provides the translated C file to see what the preprocessor does. This is helpful when debugging a new designed profiling procedure.

4.3.1 RTOS Abstraction

The RTOS macros are defined in a header file located in a subfolder of the *port* folder. Such a folder is called for instance *rtos_qnx*. In this folder are at least two files, one header file and one C file for additional function implementations.

Inside the header file, the first define-statements are configurations which will be made at the startup of a task. Listing 4.1 shows the configurations for the QNX Neutrino RTOS. It can be seen that the main task (*profilingTask*) has a higher priority than all other processes. Furthermore, it can be seen that the *profilingTask* resides on core three and the other tasks on core one and two.

Listing 4.1: Configurations of the profiling procedure tasks.

```
#define PORT_MAINTASK_CORE 3
#define PORT_MAINTASK_PRIORITY 30

#define PORT_TASK_PRIORITY_DEFAULT 20
#define PORT_TASK_PRIORITY_LOW 20
#define PORT_TASK_PRIORITY_MEDIUM 21
#define PORT_TASK_PRIORITY_HIGH 22
#define PORT_TASK_PRIORITY_IST_TASK 250

#define PORT_TASK_CORE_DEFAULT 2
#define PORT_TASK_CORE_1 1
#define PORT_TASK_CORE_2 2
#define PORT_TASK_CORE_ISR 0
```

After this definitions the translation macros for the RTOS methods are defined. Listing 4.2 shows the full list of preprocessor macros for QNX Neutrino and all available profiling procedures. It can be seen how POSIX elements such as pthread, barriers, semaphores and others are abstracted. The lines 20 to 30, of Listing 4.2 shows how the three-way configuration is done. In this case only the configuration after task creation is defined. All other possible configuration methods are left blank.

It should also be noticed, that in case of a POSIX compatible operating system, there is no need for memory buffers. In contrast to that those buffers are required in case of SafeRTOS. The problem is, that it is not possible to leave the buffer definition blank, the compiler would report an error. Therefore, those buffers are set to dummy variables as for instance an integer, which can be seen in lines 37 and 45. Unfortunately, the compiler recognizes that these variables are unused and will report warnings. This is one of the drawbacks when using preprocessor macros for portability.

Listing 4.2: Selected examples of preprocessor macros used for the QNX Neutrino RTOS.

```
1  #define PORT_Task_Pattern(TaskFunctionName)      void *TaskFunctionName( void *ptr )
2
3  #define PORT_Barrier_var pthread_barrier_t
4  #define PORT_BarrierInit(Barrier,Count) pthread_barrier_init(&Barrier, NULL, Count)
5  #define PORT_BarrierDestroy(Barrier) pthread_barrier_destroy(&Barrier)
6  #define PORT_BarrierWait(Barrier) pthread_barrier_wait(&Barrier)
7
8  // initializing of main task
9  #define PORT_MainTask_PinToCore(Core) pinThreadToCore(Core)
10 #define PORT_MainTask_SetPermissions() setPermissionsIO()
11 #define PORT_MainTask_SetPriority(Prio) setThreadPriority(Prio)
12
13 // general cleanup for the main task after a procedure
14 #define PORT_MainTask_CleanUp() return NULL
15
16
17 // initializing of procedure specific tasks before creation
18 #define PORT_Task_var pthread_t
19 #define PORT_Task_params pthread_attr_t
20 #define PORT_PreCreate_PinTaskToCore(Task, Core)
21 #define PORT_PreCreate_SetPermissions(Task)
22 #define PORT_PreCreate_SetPriority(Task, Prio)
23 #define PORT_PreCreate_SetConfigurationStruct(Task)
24 #define ↙
       PORT_PreCreate_InitWithConfigurationStruct(Task,Task_Params,TaskNumber,Core,Priority)
25 #define PORT_TaskCreate(TaskHandle, Attributes, ConfigParameter, Parameter)      ↙
       pthread_create(&TaskHandle, NULL, Attributes, Parameter);
26
27 // initializing of procedure specific tasks before after creation
28 #define PORT_TaskCreated_PinTaskToCore(Core) pinThreadToCore(Core)
29 #define PORT_TaskCreated_SetPermissions() setPermissionsIO()
30 #define PORT_TaskCreated_SetPriority(Prio) setThreadPriority(Prio)
31
32 // general cleanup after a procedure task
33 #define PORT_Task_CleanUp()      return NULL
34
35 // Semaphore
36 #define PORT_Semaphore_Var sem_t
37 #define PORT_Semaphore_Buffer(SemaphoreBuffer) int Semaphorbuffer
38 #define PORT_Semaphore_Init(Sem,Value,Buffer) sem_init(&Sem,0,Value)
39 #define PORT_Semaphore_Wait(Sem) sem_wait(&Sem)
40 #define PORT_Semaphore_Post(Sem) sem_post(&Sem)
41 #define PORT_Semaphore_Destroy(Sem) sem_destroy(&Sem)
42
43 // Mutex
44 #define PORT_Mutex_Var   pthread_mutex_t
45 #define PORT_Mutex_Buffer(MutexBuffer) int MutexBuffer
46 #define PORT_Mutex_Init(Mutex,MutexBuffer)        pthread_mutex_init(&Mutex, NULL)
47 #define PORT_Mutex_Lock(Mutex)   pthread_mutex_lock(&Mutex)
48 #define PORT_Mutex_UnLock(Mutex)          pthread_mutex_unlock(&Mutex)
49 #define PORT_Mutex_Destroy(Mutex) pthread_mutex_destroy(&Mutex)
50
51 // Yield
52 #define PORT_Scheduler_Yield()   sched_yield()
```

4.3.2 Hardware Abstraction

The hardware also needs to be abstracted by macros, in order to be portable between hardware platforms. The hardware abstraction is located, same as the RTOS macros, in a subfolder in the *port* folder. This header file is intended to provide functionality for getting timestamps, simulating CRPD and definitions of variable types. It has to be distinguished to the startup process in the *main.c* file. The startup process will also contain hardware configurations, but that will not be abstracted, because it will not be used directly inside the profiling suite.

Listing 4.3: Hardware abstraction header file of iMX6Q.

```

#define PORT_Time_Type unsigned int

#define PORT_CRPD() //SimulateCRPD()

#define MEASUREMENT_TYPE 1

#if MEASUREMENT_TYPE == 1
// Normal measurement
#define PORT_GetLocalTime()      get_cyclecount()
#define PORT_GetGlobalTime()     getGlobalTime32()
#define PORT_GetInterruptTime()  getEPIT1Time32()

#define PORT_InitLocalTime()     init_pmu()
#define PORT_InitGlobalTime()    configureGlobalTimer()
#define PORT_InitInterruptTime() configureInterruptTimer()

#define PORT_ResetLocalTime()    init_pmu()
#define PORT_ResetGlobalTime()   resetGlobalTimer()

#elif MEASUREMENT_TYPE == 2
//Overhead of timer functions
#define PORT_GetLocalTime()      measure_cyclecount()
#define PORT_GetGlobalTime()     measure_globaltime32()
#define PORT_GetInterruptTime()  measure_EPIT1Time32()

#define PORT_InitLocalTime()     init_pmu()
#define PORT_InitGlobalTime()    configureGlobalTimer()
#define PORT_InitInterruptTime() configureInterruptTimer()

#define PORT_ResetLocalTime()    init_pmu()
#define PORT_ResetGlobalTime()   resetGlobalTimer()
#endif
```

4.3.3 Evaluation of Measurement Techniques

In Listing 4.3 it can be seen that there are two different implementations for the measurement. This is because, as said in [Str14], it is necessary to measure the deviations of the used measurement techniques. For the iMX6Q board we use the PMU for singlecore only measurements and the global timer of the ARM MPCore cluster for synchronous multicore measurements. It is conspicuous, that there is a third measurement called *getEPIT1Time32()*. This is the method used to read the systick timer EPIT1 after an interrupt occurred.

In [Str14] we outlined aspects that have to be considered in order to deliver deviation aware measurements: We suggested that every measurement technique on every measuring point has to be evaluated independently. The resulting average runtimes and deviations are approximated by a Gaussian bell curve function that is further been subtracted from the measured value. After normalization that results in a probability density function of the profiled RTOS metric.

4.4 QNX Neutrino on i.MX6Q

4.4.1 Development Environment

The QNX Neutrino RTOS [QNX14a] in version 6.6.0 is installed on the Sabre Lite evaluation board [Dev14], which can be seen in Figure 4.2. In this version of the RTOS, only two processor architectures are supported: x86 and armle-v7 [QNX14d]. This means, the RTOS is no more compatible with the old CPU variant armv5 of the iMX28.

For the development we use the Momentics IDE in version 5.0 with a Board Support Package (BSP) for our hardware platform as recommended by [QNX14a]. In the Momentics IDE the BSP is imported as a project and compiled. The compilation results in a RTOS image that will later be booted on the evaluation board.

After the configuration and compilation of the BSP, the image is downloaded on the Sabre Lite evaluation board using the bootloader UBoot [Den14]. The preinstalled version of the UBoot, which is UBoot 2009.08, is used to boot the image. UBoot is configured to start up the network and download the image via Trivial File Transfer Protocol (TFTP) from the development PC. Furthermore, the Momentics IDE is configured that after building of an image or compilation of a program, it copies the executables to a folder that is accessible via TFTP.

Having all executables accessible via network is helpful to easily transfer new compiled source codes to the target platform. When QNX Neutrino is booted the network is configured with a static IP address. Every time a new version of the profiling suite is available, a shell script is started on QNX Neutrino to download the new version from the TFTP server. Furthermore, the script also starts the execution of the profiling suite task and prints the results to a PuTTYtel console on the development PC.

Sabre Lite Board

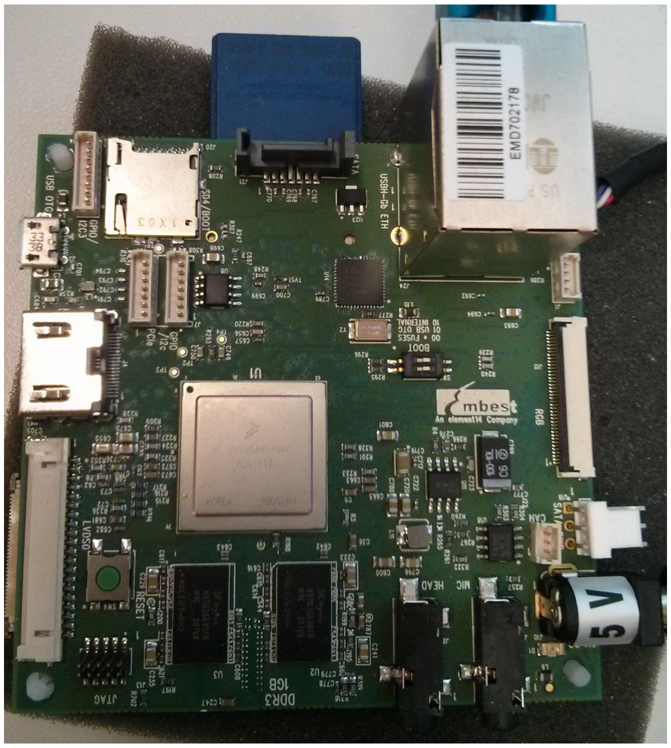

Figure 4.2: Evaluation kit for the Freescale i.MX6Q processor

4.4.2 Bord Support Package Configuration

The default configuration of the bootable RTOS image contains numerous unnecessary drivers and processes for RTOS profiling. Hence, this image is configured to be minimalistic using the build script inside the BSP. The build file also contains the ability to bind processes to a specific core. In order to provide a largely uninterrupted runtime for the profiling suite, all drivers and other processes are bound to core 0.

The BSP contains source codes for numerous drivers and also a so called startup code that is executed before the kernel. This startup code is used to configure all hardware that requires kernel mode. After the kernel is booted there is no way to reenter the kernel mode. In QNX Neutrino, the maximum permission a task can get is the IO mode, which allows the user to interact with components off the core, as for instance GPIO and timers. Features as for instance phase locked loops, clock gating, pipeline branch prediction mode or the performance monitoring unit has to be configured before the kernel starts.

For this work, we modified the function *board_cpu_startnext(unsigned cpu)* inside the BSP, in order to deactivate branch prediction and activate user mode access to the PMU. Listing 4.4 shows the executed code for every core. The configuration is made according to the documentation in [ARM10].

Listing 4.4: Startup code executed for every core.

```
/* enable user-mode access to the performance counter */
asm ("MCR p15, 0, %0, C9, C14, 0\n\t" :: "r"(1));

/* disable branch prediction */
asm("mrc        p15, 0, r0, c1, c0, 0    @ Read SCTRL\n\t"::"r"(0));
asm("bic        r0, r0, #0x00000800      @ Z = 0\n\t"::"r"(0));
asm("mcr        p15, 0, r0, c1, c0, 0    @ Write SCTRL\n\t"::"r"(0));
```

4.4.3 Performance Monitoring Unit

After the PMU is activated, it has to be configured and reset before every profiling procedure. This is, because the PMU has got a 32 bit cycle counter and the core runs at 792 MHz, after a few seconds an overflow would occur. Initializing and resetting the PMU are done by exactly the same function that does the following steps:

- Enable all counters
- Reset all counters
- Reset all overflows
- Disable the frequency divider

When the PMU has been configured the profiling procedure starts and whenever a measurement has to be taken, the PMUs Cycle CouNT (CCNT) register is read. This is done by a single assembler statement, as it can be seen in Listing 4.5. In Listing 4.5 it can also be seen that the prototyping and implementation are done static and inline, including an attribute called always_inline. That means, the compiler, in this case the GCC v4.7.3 [GNU14a], is enforced to implement the function inline [GNU14b]. By not doing so, the measurement would lose precision due to the additional function call overhead [Str14].

Listing 4.5: Code for reading the CCNT register of the PMU.

```
static inline uint32_t get_cyclecount (void) __attribute__((always_inline));
static inline uint32_t get_cyclecount (void)
{
  uint32_t value;
  // Read CCNT Register
  asm volatile ("MRC p15, 0, %0, c9, c13, 0\t\n": "=r"(value));
  return value;
}
```

4.4.4 Timer

Aside from the PMU, the global 64 bit timer is the second fastest measurement method on the i.MX6Q. The global timer has the advantage to be synchronous over all cores, which makes it suitable for multicore measurements. On the i.MX6Q platform, the global timer is located off the processor core, inside the Snoop Control Unit (SCU) [Str14]. Therefore it is possible to access the timer registers using memory mapped IO. This means, in order to access this registers it is sufficient to have IO permissions, no kernel mode is needed. The configuration of the global timer is done the following way:

- Request IO Permissions
- Memory map the timer base address, including read and write access and without caching
- Set the higher and lower counting values to zero
- Configure the timer to be free running: only enable the timer, all interrupts and features are turned off

On the startup of a profiling procedure, the timer is reseted. This is done, in order to prevent overruns of the lower 32 bit counting register, because the global timer runs at the halve frequency of the CPU, which is in this case 396 MHz. By resetting the timer before every profiling procedure, it is possible to read only the lower 32 bit value without a possible overrun. This results in an overall faster time to get a timestamp.

Reading the actual timer value is done in the same way as for the PMU. The read method is implemented static and forced to be inlined. The only difference is the used assembler command, which can be seen in Listing 4.6. Furthermore, it can be seen that, compared to the PMU read command, this command needs to read a value called *addr*. This is the address where the memory mapping mapped the physical address of the timer registers is stored. Reading this *addr* value might cause cache misses that influences the measurement. Therefore, as recommended in [Str14], especially this measurement technique needs evaluation.

Listing 4.6: Assembler command for reading the lower 32 bit value of the global timer.

```
__asm__ __volatile__("ldr %0, [%1]\n\t"\
        : "=&r"(value)
        : "r" (addr)
        : "memory");
```

When implementing a timer based measurement technique on another hardware, it is possible that the RTOS uses the desired timer for its own purposes. The user of the profiling suite should investigate which timer is already used by the RTOS and which one is free for use. Another possible mistake, when implementing timer based measurement techniques, is the raising of clocking frequencies of the timer, in order to obtain better accuracy. Changing the clock frequency might result in a changed system behavior that results in unrealistic measurements. The focus lies on the profiling of an RTOS for a project specific configuration that should be used in a real world application.

4.4.5 Task Configuration

QNX Neutrino implements, same as Linux and numerous other operating systems, the POSIX API interface. The configuration can either be done by the creating task or by the created task within its runtime. We decided to use the configuration interface when the task is already running. So the task configures its own parameters.

As already mentioned, all tasks are required to have the permissions to access IO modules. This is necessary in order to configure and operate time measurement methods. To be more precise, due to the userspace configuration of the PMU on startup, it is not necessarily needed to raise privileges for every task. However, it is still possible to switch to another measurement method that may need these privileges, so we decided to give all tasks IO permissions.

In addition to the IO permissions, all tasks need to know their scheduling priority and policy. The task configures its priority depending on its role in the actual profiling procedure. For the implemented profiling procedures it is necessary to configure the scheduling policy to FIFO without time slicing. This is, because it makes it easier to implement this specific profiling procedures for RTOS services. The FIFO policy makes it also possible to implement software based Cache Related Preemtion Delay (CRPD) simulations. Listing 4.7 shows the used methods to configure the task priority and the scheduling policy.

Listing 4.7: Pthread commands to set the scheduling policy and the priority for a given task.

```
pthread_getschedparam(pthread_self(), &policy, &param);
policy = SCHED_FIFO;
param.sched_priority = priority;
pthread_setschedparam(pthread_self(), policy, &param);
```

Another important step is to pin the task to a specific core. Therefore, a method, provided within the QNX 6.6.0 documentation [QNX14b], is used. The method principally sets the tasks runmask and the inheritance mask. This means, the runmask defines the core affinity and the inheritance mask is the runmask that is inherited by child tasks. Binding the thread to a core is only effective in multicore mode. When switching to singlecore mode this runmasks are ignored by the operating system.

The configuration methods presented in this section are using numerous kernel calls, mallocs and other hard to predict methods. However, that does not affect the profiling at all, because the tasks wait until all of them reached the common barrier. After that barrier, it is not recommended to use methods that are hard to predict.

4.4.6 CRPD Simulation

The Cache Related Preemption Delay (CRPD), as described in Section 3.3.1, is an additional delay caused by cache misses. This special cache misses occur, because the running task is preempted by another task. Switching to and back from the preempting task causes parts of the instruction and data cache to be invalidated.

There are two possible ways to simulate CRPD, by hardware and software based methods. Hardware cache invalidation requires, depending on the used processor, kernel permissions to be executed. QNX Neutrino does not provide methods to raise the threads permissions to such a high level. Therefore, it is hard to implement the hardware based cache invalidation.

The remaining alternatives are the software based methods, which can be implemented easier in the userspace. The implementation for this work is designed to fill a data array with N elements, in order to fill the data cache with useless data. Furthermore, a code with M lines of assembler commands is executed in order to invalidate the instruction cache. The number N and M depends on the size of the cache. In this case the processor has a 32kB I-Cache and a 32kB D-Cache.

A similar approach is used in [Kin13]. The author describes a task called cache trasher that is used to invalidate the L2 cache, in order to show the benefits of cache partitioning. The difference to our implementation is that our CRPD simulation is done in the context of one of our profiling tasks. This is, because it is easier to implement the strict order of execution and the exact moment of the CRPD simulation can be assigned without disturbing the profiling procedure.

4.5 SafeRTOS on i.MX287

4.5.1 Development Platform

SafeRTOS runs on an i.MX287 processor on the Freescale iMX28 evaluation kit revision D. The evaluation board can be seen in Figure 4.3. SafeRTOS is a safety certified derivate of FreeRTOS [Wit14]. Furthermore, SafeRTOS in version 4.6 is not able to run on multicore processors, without any third party hypervisor technologies.

For the development we use the recommended Sourcery CodeBench Integrated Development Environment (IDE) in version 2011.07-41. This IDE delivers compilers, the so called Sourcery CodeBench C Compiler and Sourcery CodeBench Assembler, which are used for compilation of the RTOS and also the profiling suite.

We are using the example project for the iMX28 evaluation kit provided by Wittenstein. This project is used for the integration of our profiling suite with as little as possible changes. It contains a linker script that handles the linking procedure of the Sourcery CodeBench C Linker. The result is an executable file that can be transfered to the hardware using the Segger J-Link JTAG probe version 8.0.

By using the JTAG interface, it is possible to do a line by line debugging, which is helpful when designing new profiling procedures. Therefore, all code resides in the same project, the profiling suite, the RTOS kernel code and the board support package. On startup, when the control flow reaches the main function, at first the hardware is initialized. Afterwards the scheduler is initialized and the *profilingTask* is created. Finally, the scheduler is started with the *profilingTask* as the only runnable task on the system.

i.MX28 EVK

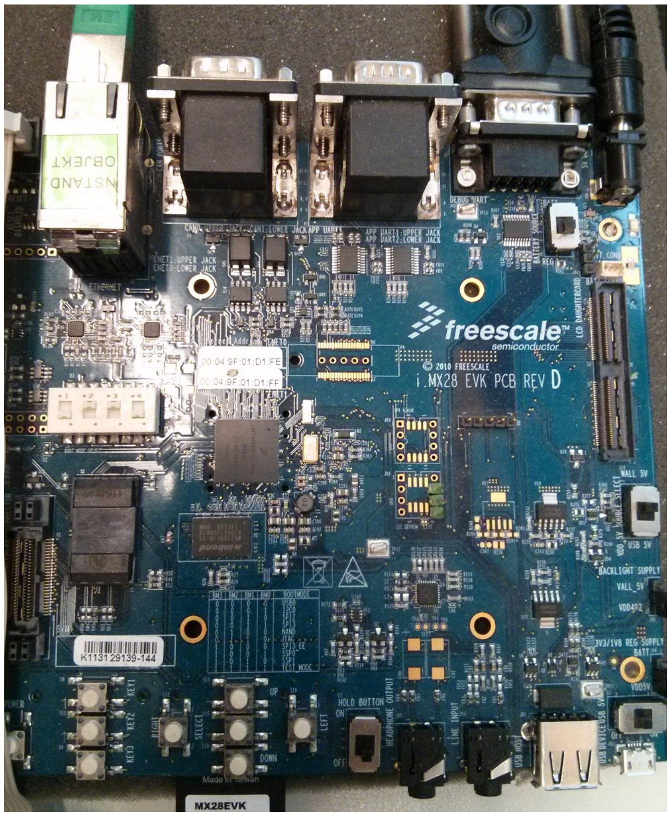

Figure 4.3: Evaluation kit for the Freescale i.MX28 processor

4.5.2 Timer

For the time measurements on the iMX287 we use timer 1. This is a timer that is not used by SafeRTOS. It is a 32 bit timer configured to be free running. The timer is connected to the main peripheral bus system that runs at a frequency of 24 MHz. In order to get a safe result that is not possible to overrun, the timer is reseted before the start of a profiling procedure.

The methods to configure and read from the timer 0 are already implemented by the operating system in order to generate a system tick. We adopted these functions to configure and read the timer 1. We used the same configuration as in QNX Neutrino. The timer is free running and its 32 bit value is read during the profiling procedures loops. The deviation of reading a timestamp from the timer is recorded by calling the measurement method twice and subtracting the values.

4.5.3 MyBarrier

All implementations of our profiling procedures for RTOS services require a synchronized start. The order of the tasks startup is mandatory. Therefore, synchronization mechanisms are needed to provide such a predictable startup. The barrier, is one of the central elements used in every profiling procedure. Unfortunately SafeRTOS in version 4.6 does not provide such a functionality.

Therefore, we implemented our own barrier method for SafeRTOS, called MyBarrier. The barrier itself is a construct where a counter is incremented until a maximum number (N) of waiting tasks is reached. When N-1 tasks are waiting on the barrier the last task wakes up the other tasks and all tasks are scheduled according to their priority. The barrier has to be aware of different priorities of the waiting tasks in a real-time system, waking up a task with a higher priority than the waking task leads to an interruption of the waking process. Therefore, the waking task has to raise its priority to a higher level than all other waiting tasks. The principal process can be seen in Listing 4.8.

Listing 4.8: Pseudocode version of the waitbarrier method of MyBarrier.

```
lock.accuire();
waiting_counter++;
if(waiting_counter < MAX){
        lock.release();
        suspend();
}
else{
        setPriority(self, max(priorities));
        resume_all_tasks();
        setPriority(self, oldpriority);
        lock.release();
        yield();
}
```

4.5.4 Task Configuration

In SafeRTOS the recommended way to configure tasks is by setting attributes in a struct. The pointer to this struct is passed to the kernel that copies the attributes to the task control block. This strategy of configuring tasks is applied before the task is created, which is possible due to the three-way configuration of the RTOS profiling suite.

The struct to configure a task in SafeRTOS contains the following attributes that has to be configured [Wit12]:

- **Functionpointer:** A pointer to a function that implements the task.
- **Name:** A character array that contains a name for the task. This is mainly used for debugging.
- **Task Control Block (TCB):** A pointer to a global variable that should be used as TCB for this task.
- **Stack:** Also a global array that will be used as stack for the task.
- **Stack size:** The size of the stack.
- **Parameters:** Additional parameters, not used in this work.

- **Priority:** The priority of the task. Minimum is one, maximum is six. The profiling-Task runs at priority level one, the other tasks runs at priority level two by default. Note: the priority can be changed during runtime.
- **Privileged mode:** The task can be either privileged or unprivileged. In our implementation, all tasks are set to be privileged, in order to access CPU configuration registers.
- **Region definitions:** Definitions for the Memory Management Unit (MMU) regions. This region definitions are set to the default values.

4.5.5 CRPD Simulation

In QNX Neutrino it is necessary to simulate the Cache Related Preemption Delay (CRPD) using software methods to invalidate caches, because the required permissions can not be obtained. In case of SafeRTOS it is possible to raise the privileges of a task to kernel mode. This enables the tasks to manipulate the cache configurations whenever they want.

The implementation of the CRPD simulation is done by at first deactivating both, the instruction and the data cache of the i.MX28. By disabling those two caches, all stored data inside is dropped. After that, the caches are re-enabled and the profiling procedure is started. That means, all necessary lines of code and data has to be fetched from the main memory. This simulates the worst case scenario of CRPD, where the activity of the preempting task has invalidated all data from the caches.

There is one drawback of this method, it can not be used in combination with cache line locking. All data in the cache would be removed no matter if the lines are locked or not.

Chapter 5

Results

This chapter presents the measurement results of our profiling procedures for RTOS metrics. There are two separated measurements done per RTOS, one for the normal runtimes and the other for the measurements including CRPD simulations. The results presented in this chapter are measured on two different real-time operating systems, SafeRTOS and QNX Neutrino. QNX Neutrino is capable of running on a multicore CPU in either SMP or BMP mode. Additionally, it is also capable of running on a multicore CPU in singlecore mode. Therefore, the singlecore profiling procedures for RTOS metrics are measured on SafeRTOS, QNX Neutrino in BMP mode and also on QNX Neutrino in singlecore mode.

As a note for all measurements made in this chapter: In order to be statistically independent we decided to do 1000 measurements of a profiling procedure in series. Therefore, also the chance to be interrupted by any interrupt on the system is higher. Another point for such a high number of measurements, is that the probability that locking errors get visible is higher.

5.1 SafeRTOS on i.MX28

In this section, the results of the four possible profiling procedures on SafeRTOS are presented. The results are discussed in two separated ways, with and without CRPD simulation. This helps to view the results from two perspectives and also shows the impact of the used CRPD simulation strategy.

5.1.1 Statistics

Table 5.1 shows the runtimes of the profiling procedures without CRPD simulation activated. It can be seen that minimum observed and mean values are close together. In contrast to that mean value and maximum observed value are wider apart to each other. This is, because longer runtimes can not be observed often, which is positive for the RTOS implementation. Hence, long runtimes in these measurements are considered outliers.

Table 5.2 presents the same profiling procedures as Table 5.1, but with activated CRPD simulation. It can be seen that the observed minimum, average and maximum values are significantly larger compared to the measurements without CRPD simulation. However, the standard deviations and the difference between minimum and maximum observed values shows a reduction. This means, there are fewer outliers when CRPD is activated.

	Min	**Max**	**Mean**	**Std. Dev.**
Context Switch Time	5.9583	7.2916	5.9906	0.059597
Message Pass Time	24.2083	30.7916	24.9059	0.34413
Semaphore Shuffle Time	32.5416	44.9583	33.1433	0.50584
Preemption Time	13.2917	21.4583	13.3653	0.27158

Table 5.1: Statistical evaluation of the four possible profiling procedures on SafeRTOS without CRPD simulation

That indicates a meaningful CRPD simulation, because the average observed runtime is close to the worst observed runtime.

	Min	**Max**	**Mean**	**Std. Dev.**
Context Switch Time	16.1338	17.7883	16.6877	0.14994
Message Pass Time	53.0119	55.4556	53.8553	0.34607
Semaphore Shuffle Time	69.3386	72.5034	70.8451	0.31825
Preemption Time	37.3254	38.8385	37.8991	0.14807

Table 5.2: Statistical evaluation of the four possible profiling procedures on SafeRTOS with CRPD simulation

5.1.2 Precision

We stated in [Str14] that deviations caused by the measurement technique should be considered when evaluating results. Table 5.3 shows the average runtimes and the corresponding standard deviation of the timing function when CRPD simulation is turned off. It can be seen that the standard deviation is close to zero in every case. This means, only the average runtime has to be subtracted from the measurement.

When comparing Table 5.3 with 5.4, it can be seen that the CRPD simulation also affects the runtimes of the measurement technique. Therefore, it is necessary to evaluate the CRPD simulation values using the recommended techniques in [Str14]. The difference in evaluation can be seen in Figures 5.3, 5.4, 5.7 and 5.8.

Normal Measurement

	Mean	**Std. Dev.**
Context Switch Time	0.3334	0.0027951
Message Pass Time	0.3334	0.0027951
Semaphore Shuffle Time	0.66673	0.0027951
Preemption Time	0.29167	0

Table 5.3: Deviations of the used measurement techniques on SafeRTOS without CRPD simulation enabled

CRPD Simulation

	Mean	**Std. Dev.**
Context Switch Time	0.41396	0.081295
Message Pass Time	0.49542	0.025348
Semaphore Shuffle Time	0.78731	0.096909
Preemption Time	0.35554	0.064672

Table 5.4: Deviations of the used measurement techniques on SafeRTOS with CRPD simulation enabled

5.1.3 Figures

Normal Measurement **CRPD Simulation**

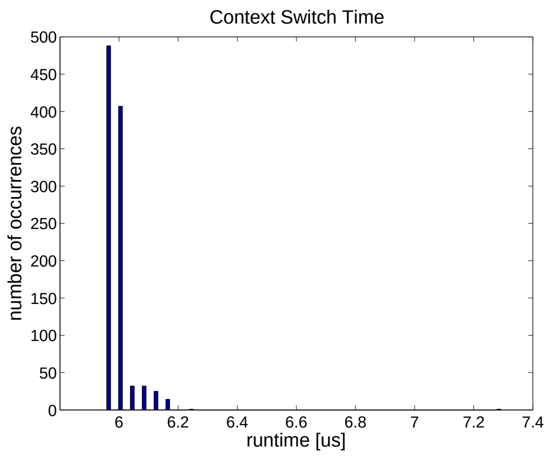

Figure 5.1: SafeRTOS without CRPD: Context switch time

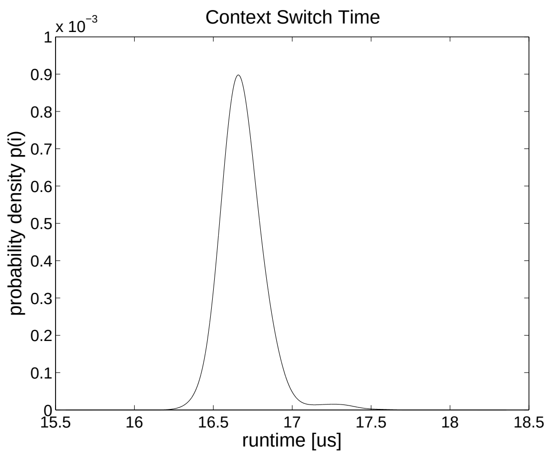

Figure 5.3: SafeRTOS with CRPD: Context switch time

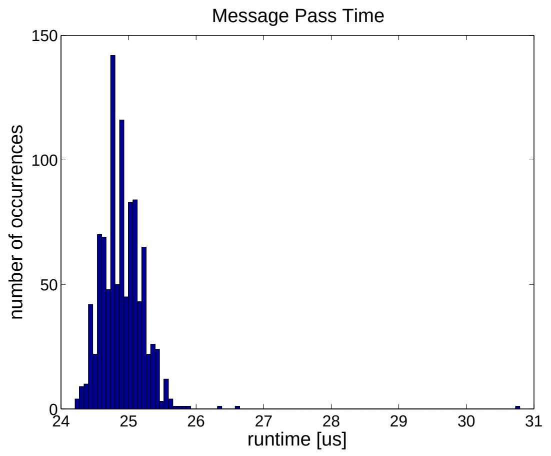

Figure 5.2: SafeRTOS without CRPD: Message pass time

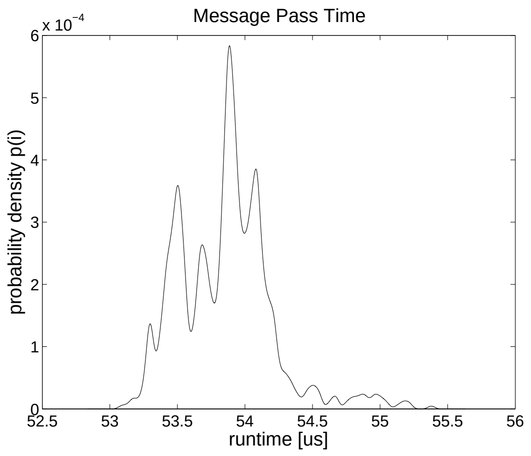

Figure 5.4: SafeRTOS with CRPD: Message pass time

Normal Measurement **CRPD Simulation**

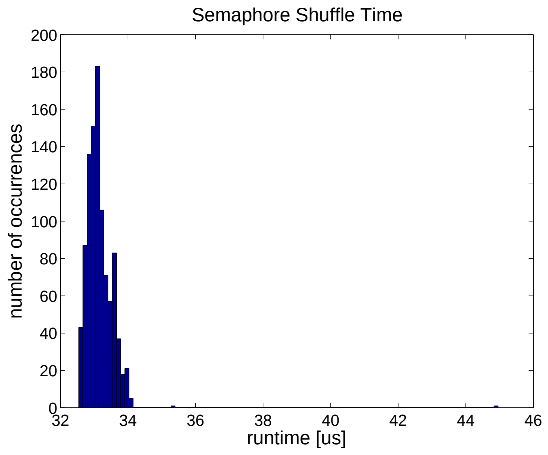

Figure 5.5: SafeRTOS without CRPD: Semaphore shuffle time

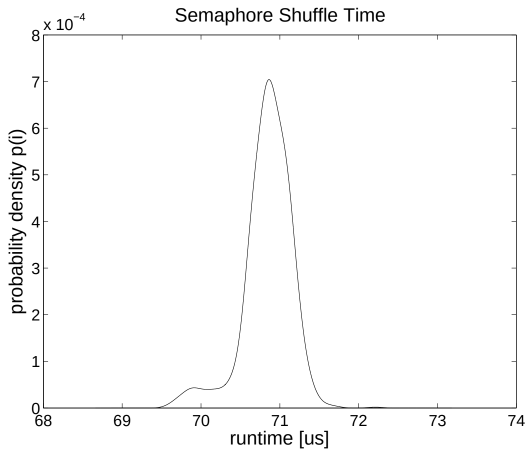

Figure 5.7: SafeRTOS with CRPD: Semaphore shuffle time

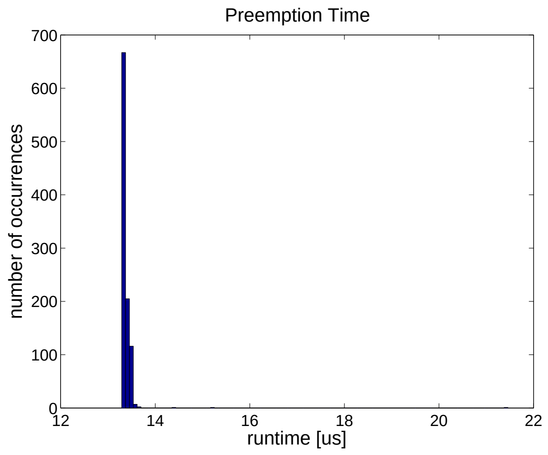

Figure 5.6: SafeRTOS without CRPD: Preemption time

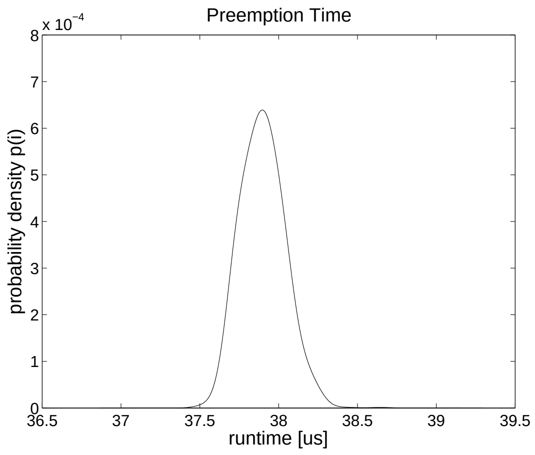

Figure 5.8: SafeRTOS with CRPD: Preemption time

5.1.4 Discussion

Figures 5.1, 5.2, 5.5 and 5.6 presents the normal measurements and Figures 5.3, 5.4, 5.7 and 5.8 shows the measurements with CRPD simulation. By comparing those images, it can be seen that every normal measurement exhibits outliers. On the other hand with activated CRPD simulation, the number of outliers is reduced. This is, because the CPRD simulation deletes reliably every cached value from the first level cache and therefore it simulates the expected worst case state of the cache. On the beginning of every run of a profiling procedure, all required code and data has to be fetched from the main memory.

5.2 QNX Neutrino in Singlecore mode on i.MX6Q

QNX Neutrino runs by default in BMP mode on a multicore CPU. However, it is also possible to run the singlecore kernel on a multicore CPU. Thus, all other cores than core null are deactivated and the system shows the same behavior as a regular singlecore system.

5.2.1 Statistics

Table 5.5 shows the timing behavior of the implemented singlecore profiling procedures in the singlecore mode of QNX Neutrino. When comparing the values of Table 5.5 with Table 5.1, it can be seen that the differences between minimum and maximum runtime are greater in case of QNX Neutrino. It can also be seen in Table 5.5 that the deadlock breaktime takes more time to execute than the semaphore shuffle time. This behavior is expected, because resolving the deadlock is more complex than passing a semaphore to another process.

	Min	**Max**	**Mean**	**Std. Dev.**
Context Switch Time	0.68813	3.5038	0.69137	0.089556
Deadlock Break Time	3.9103	10.5631	4.1206	0.30464
Message Pass Time	2.0215	13.8775	2.1821	0.43196
Semaphore Shuffle Time	2.6275	4.9811	2.6886	0.10695
Preemption Time	0.68725	4.9519	0.75968	0.15195

Table 5.5: Statistical evaluation of the five singlecore profiling procedures on QNX Neutrino without CRPD simulation

In Table 5.6 the measurements with activated CRPD simulation are presented. As expected, the CPRD simulation increases all runtimes and deviations of all profiling procedures. Especially, the preemption time procedure shows increased runtimes. This behavior indicates that developers need to take care about possible caching issues when handling interrupts by waking up a thread.

	Min	**Max**	**Mean**	**Std. Dev.**
Context Switch Time	1.5139	4.6515	1.9595	0.26936
Deadlock Break Time	8.4596	17.4495	9.3095	0.53257
Message Pass Time	4.9495	19.5997	5.7478	0.63221
Semaphore Shuffle Time	5.2184	9.6275	5.8866	0.44394
Preemption Time	5.1135	13.0667	5.641	0.59398

Table 5.6: Statistical evaluation of the five singlecore profiling procedures on QNX Neutrino with CRPD simulation

5.2.2 Precision

For singlecore-only profiling procedures the PMU is used as measurement device. It can be seen in Tables 5.7 and 5.8 that the standard deviation is close to zero and therefore no Gaussian bell curve fitting is necessary [Str14]. The only exception is the preemption time profiling procedure, where the cyclic timer of the system is read after an interrupt has occurred and when the interrupt service task has been woken. The time to read the system timer exhibits a small but not negligible deviation, which can be seen in Figures 5.17 and 5.18.

Normal Measurement

	Mean	**Std. Dev.**
Context Switch Time	0.0050505	0
Deadlock Break Time	0.010114	0.00012658
Message Pass Time	0.0050505	0
Semaphore Shuffle Time	0.010101	0
Preemption Time	0.29406	0.0062306

Table 5.7: Deviations of the used measurement techniques on QNX Neutrino in singlecore mode without CRPD simulation enabled

CRPD Simulation

	Mean	**Std. Dev.**
Context Switch Time	0.0050505	0
Deadlock Break Time	0.010101	0
Message Pass Time	0.0050947	0.00046297
Semaphore Shuffle Time	0.010101	0
Preemption Time	0.29626	0.0098733

Table 5.8: Deviations of the used measurement techniques on QNX Neutrino in singlecore mode with CRPD simulation enabled

5.2.3 Figures

In this section, the figures are presented such that the normal and the CRPD simulation measurement can be compared directly. This enables the reader to draw conclusions of the impact of the CRPD simulation to a given RTOS metric.

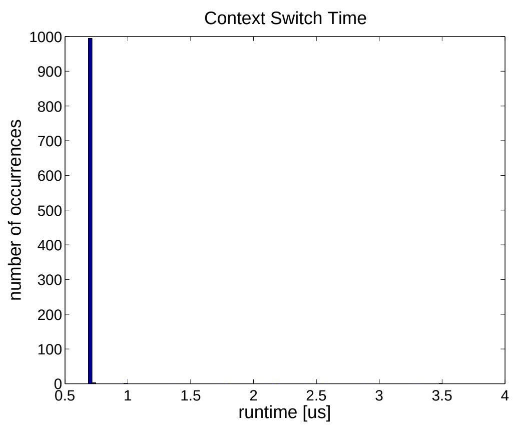

Figure 5.9: Context Switch Time on QNX Neutrino in singlecore mode without CRPD simulation.

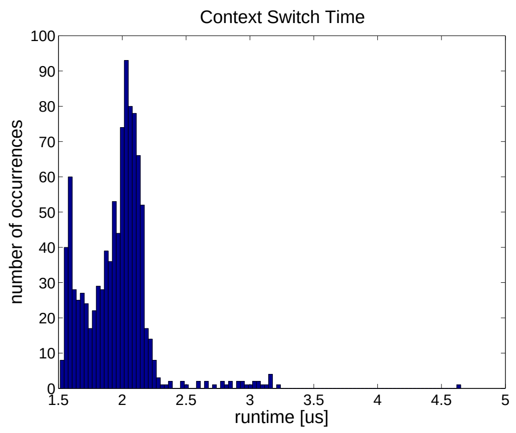

Figure 5.10: Context Switch Time on QNX Neutrino in singlecore mode with CRPD simulation.

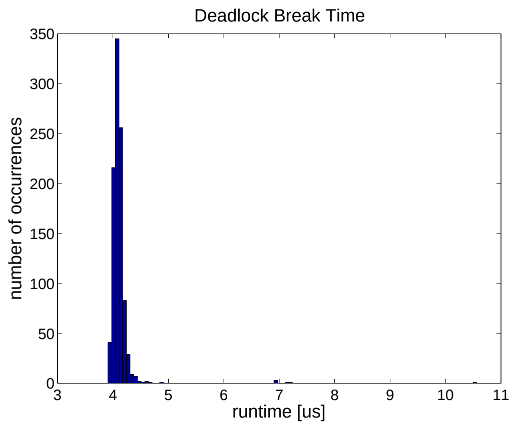

Figure 5.11: Deadlock Break Time on QNX Neutrino in singlecore mode without CRPD simulation.

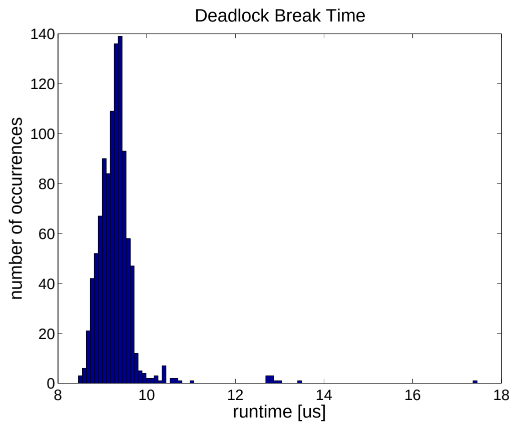

Figure 5.12: Deadlock Break Time on QNX Neutrino in singlecore mode with CRPD simulation.

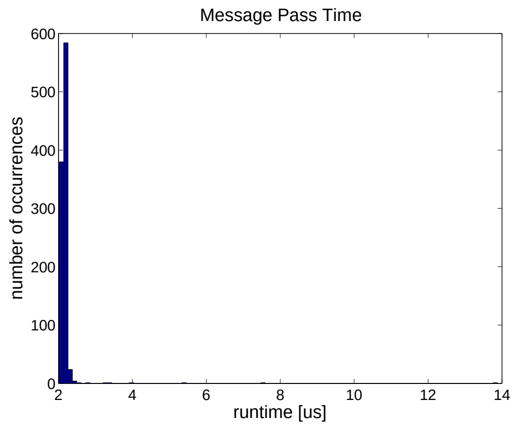

Figure 5.13: Message Pass Time on QNX Neutrino in singlecore mode without CRPD simulation.

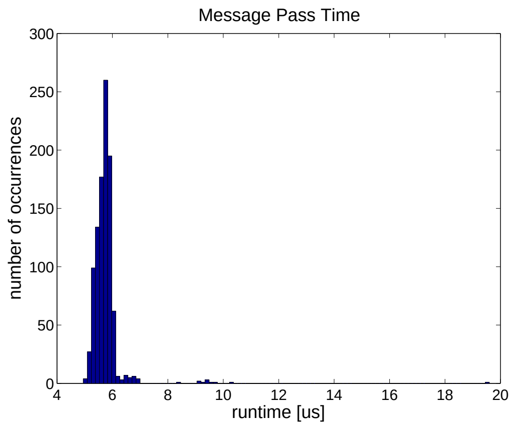

Figure 5.14: Message Pass Time on QNX Neutrino in singlecore mode with CRPD simulation.

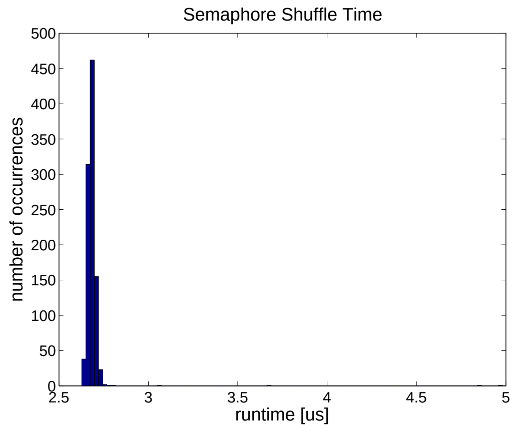

Figure 5.15: Semaphore Shuffle Time on QNX Neutrino in singlecore mode without CRPD simulation.

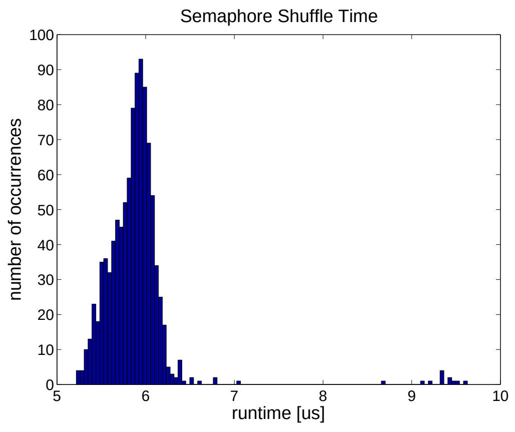

Figure 5.16: Semaphore Shuffle Time on QNX Neutrino in singlecore mode with CRPD simulation.

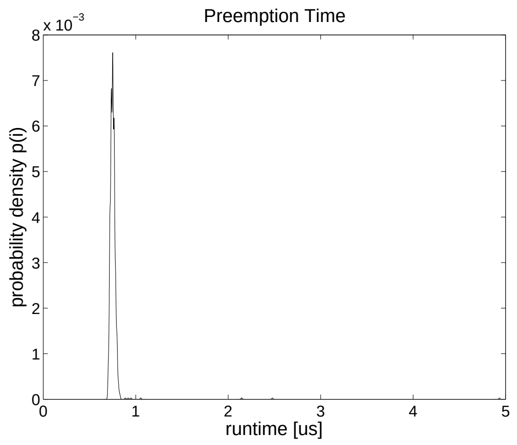

Figure 5.17: Preemption Time on QNX Neutrino in singlecore mode without CRPD simulation.

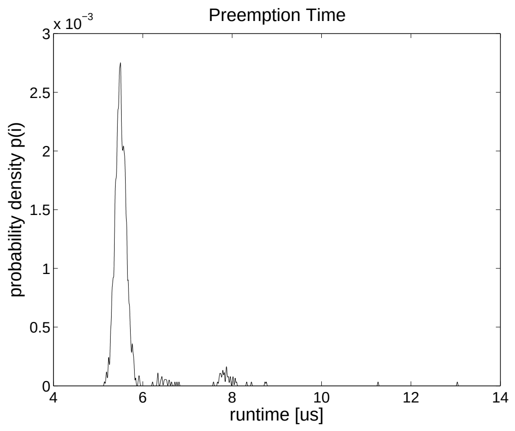

Figure 5.18: Preemption Time on QNX Neutrino in singlecore mode with CRPD simulation.

5.2.4 Discussion

Comparing Figures 5.9, 5.11, 5.13, 5.15 and 5.17 with Figures 5.10, 5.12, 5.14, 5.16 and 5.18, it can be seen that all normal and all CRPD measurements exhibits outliers. The presence of outliers indicate interruptions or caching issues during the measurements. In our configuration of QNX Neutrino, every task that does not belong to the profiling suite runs at priority level 24 or below. The profiling procedure tasks are configured to run at least at priority level 30. That means, only interrupts and the associated interrupt service routines are able to preempt one of our profiling procedure tasks.

By default, in QNX all interrupts are serviced on core zero. In BMP mode it is possible to run the profiling tasks on another core than core zero in order to avoid these interruptions. However, that is not possible in singlecore mode. Every outlier increases the maximal observed execution time of a function. Developers of singlecore systems have to deal with such possible outliers by either disabling every functionality that might interrupt the profiling tasks or interpret these results it as a realistic scenario where the task is interrupted by an unpredictable occurring interrupt.

5.3 QNX Neutrino in BMP mode on i.MX6Q

This section presents the results of the profiling procedures for single- and multicore profiling procedures. As well, as it can be seen in the last section, the figures of the results are arranged, such that the reader can compare the normal measurements with the measurements with CRPD simulation activated. The BMP mode in QNX Neutrino is in fact the SMP kernel, but with all running system tasks pinned to core zero. That ensures a largely undisturbed run of the profiling procedures.

5.3.1 Statistics

Table 5.9 shows the results of the measurements without CRPD simulation. By comparing the values of Table 5.9 with Table 5.5, it can be seen that all singlecore profiling procedures show greater minimum and average runtimes on the multicore configuration of QNX Neutrino. It is noticeable, that the maximum runtime of the context switch time is smaller than in singlecore mode of QNX Neutrino. That indicates, that in there are fewer outliers in multicore mode.

Table 5.9 presents the first results of the multicore profiling procedures. It can be seen that all multicore variants exhibits longer runtimes compared to the singlecore variants. Except the message passing procedure in the singlecore and the multicore variant shows the same behavior. This happens, because the implementation uses a message passing server in order to be POSIX compatible. This implementation of message passing makes it necessary to redirect the communication through the message passing server, which is running at core 0. Therefore, it does not make a difference if both communicating tasks are running on the same or on different cores.

	Min	**Max**	**Mean**	**Std. Dev.**
Context Switch Time	0.91919	1.2475	0.95742	0.032952
Deadlock Break Time	6.0631	19.572	6.4522	0.51687
Message Pass Time	3.4255	16.5013	3.6421	0.41408
Semaphore Shuffle Time	3.4874	16.7159	3.5853	0.45496
Preemption Time	2.4311	7.5823	2.6589	0.3584
Multicore Context Switch Time	0.93434	26.6414	2.4178	0.61892
Multicore Deadlock Break Time	11.7697	22.4638	12.6751	0.6944
Multicore Message Pass Time	3.2692	8.4173	3.7057	0.2926
Multicore Semaphore Shuffle Time	6.8032	19.7607	7.4351	0.51694

Table 5.9: Statistical evaluation of the nine single- and multicore profiling procedures on QNX Neutrino without CRPD simulation

The results of the profiling procedures with CRPD simulation are summed up in Table 5.10. As expected, by comparing Table 5.10 with Table 5.9 it can be seen that the CRPD simulation increases almost all runtimes of the profiling procedures. In the last section, it turns out that CRPD simulation has serious effect on the preemption time. The preemption time procedure takes more time to execute at all, but also the CRPD simulation has a significant effect.

	Min	Max	Mean	Std. Dev.
Context Switch Time	2.2374	6.5985	2.4651	0.31031
Deadlock Break Time	11.2424	21.8965	11.9868	0.66253
Message Pass Time	7.2033	21.9318	7.6591	0.51604
Semaphore Shuffle Time	6.8636	17.9482	7.2363	0.54942
Preemption Time	9.5709	15.5458	10.1978	0.66037
Multicore Context Switch Time	4.1692	20.4495	5.014	1.0385
Multicore Deadlock Break Time	16.5419	34.3772	17.5024	0.77552
Multicore Message Pass Time	8.276	18.5263	9.2457	0.71359
Multicore Semaphore Shuffle Time	10.4656	22.9752	13.3712	0.53426

Table 5.10: Statistical evaluation of the nine single- and multicore profiling procedures on QNX Neutrino with CRPD simulation

5.3.2 Precision

The Tables 5.11 and 5.12 show the average runtime and deviations of the measurement techniques used for singlecore-only and multicore measurements. The singlecore-only and the multicore context switch time measurements are made by the PMU, which shows an insignificant deviation. Preemption time and other multicore measurements are made by using the global timer of the iMX6Q. These measurement techniques exhibits deviations that needs to be compensated by a Gaussian bell fitting.

Normal Measurement

	Mean	Std. Dev.
Context Switch Time	0.0050505	0
Deadlock Break Time	0.010101	0
Message Pass Time	0.0050505	0
Semaphore Shuffle Time	0.010101	0
Preemption Time	0.29633	0.0049829
Multicore Context Switch Time	0.010101	0
Multicore Deadlock Break Time	0.14963	0.021828
Multicore Message Pass Time	0.046893	0.0065863
Multicore Semaphore Shuffle Time	0.10947	0.017313

Table 5.11: Deviations of the used measurement techniques on QNX Neutrino in BMP mode without CRPD simulation enabled

CRPD Simulation

	Mean	Std. Dev.
Context Switch Time	0.0050505	0
Deadlock Break Time	0.010101	0
Message Pass Time	0.0050505	0
Semaphore Shuffle Time	0.010101	0
Preemption Time	0.29768	0.0084374
Multicore Context Switch Time	0.010101	0
Multicore Deadlock Break Time	0.15788	0.033578
Multicore Message Pass Time	0.058425	0.01057
Multicore Semaphore Shuffle Time	0.13566	0.024313

Table 5.12: Deviations of the used measurement techniques on QNX Neutrino in BMP mode with CRPD simulation enabled

5.3.3 Figures

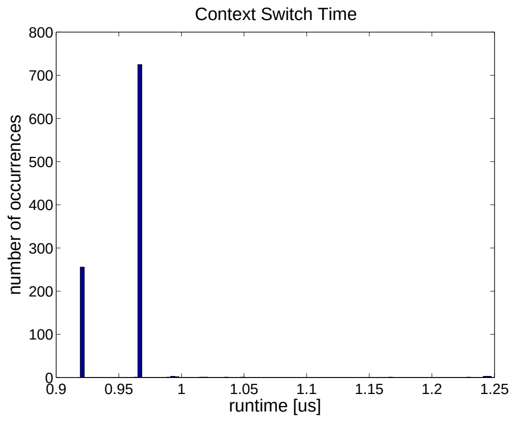

Figure 5.19: Context Switch Time on QNX Neutrino in BMP mode without CRPD simulation.

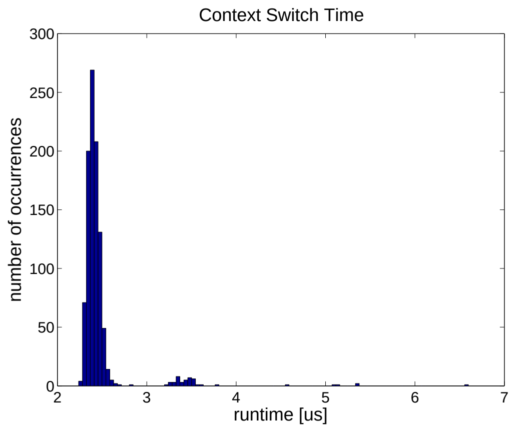

Figure 5.20: Context Switch Time on QNX Neutrino in BMP mode with CRPD simulation.

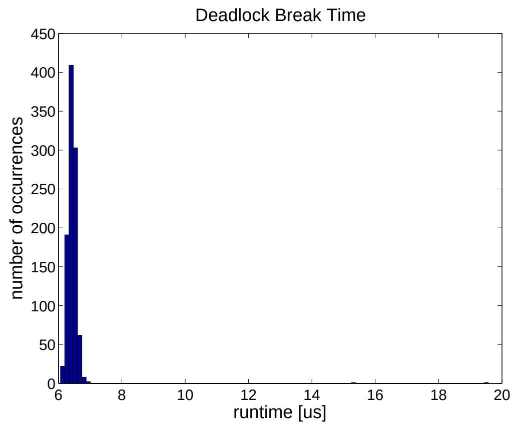

Figure 5.21: Deadlock Break Time on QNX Neutrino in BMP mode without CRPD simulation.

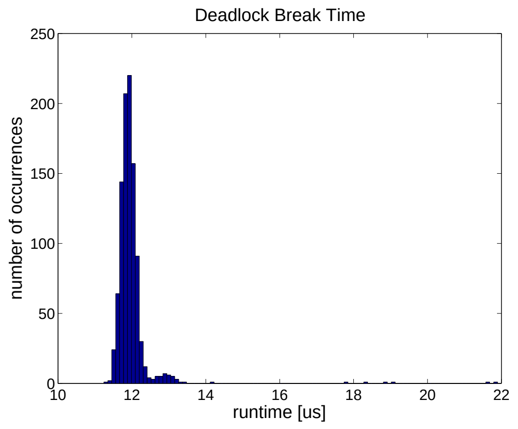

Figure 5.22: Deadlock Break Time on QNX Neutrino in BMP mode with CRPD simulation.

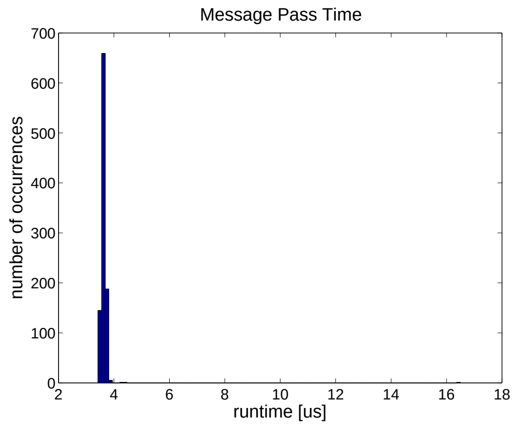

Figure 5.23: Message Pass Time on QNX Neutrino in BMP mode without CRPD simulation.

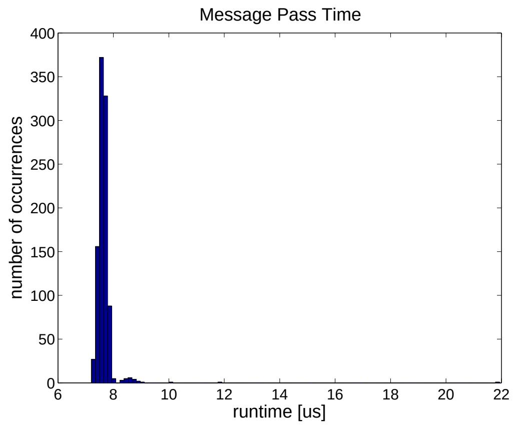

Figure 5.24: Message Pass Time on QNX Neutrino in BMP mode with CRPD simulation.

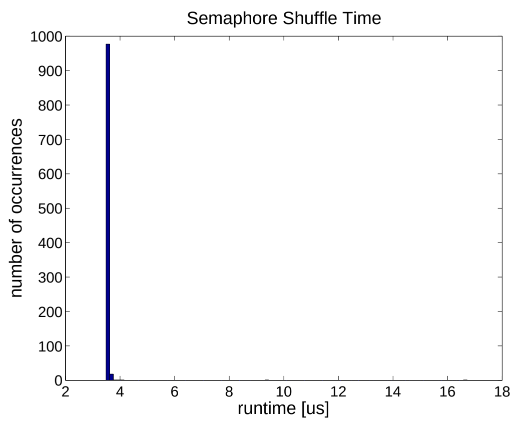

Figure 5.25: Semaphore Shuffle Time on QNX Neutrino in BMP mode without CRPD simulation.

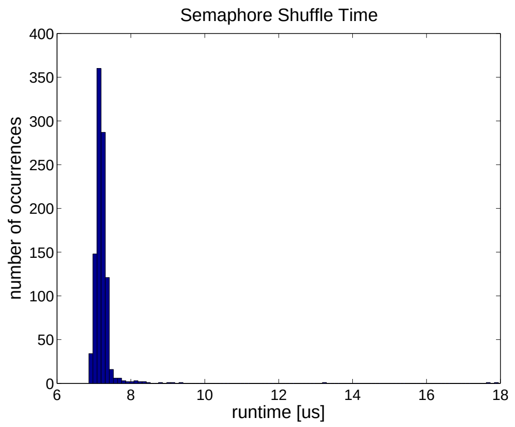

Figure 5.26: Semaphore Shuffle Time on QNX Neutrino in BMP mode with CRPD simulation.

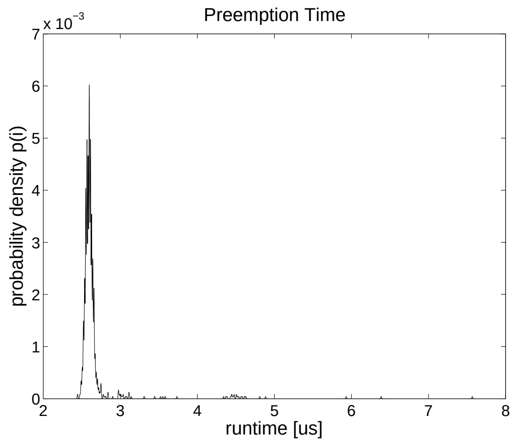

Figure 5.27: Preemption Time on QNX Neutrino in BMP mode without CRPD simulation.

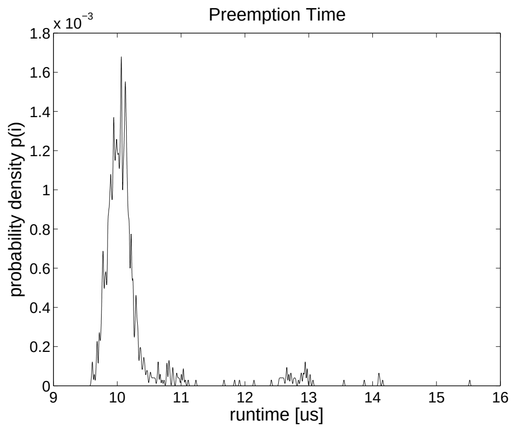

Figure 5.28: Preemption Time on QNX Neutrino in BMP mode with CRPD simulation.

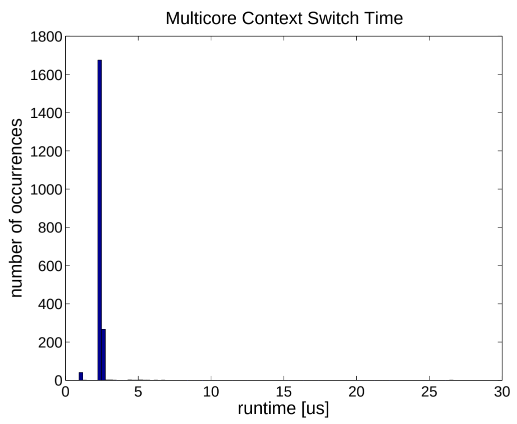

Figure 5.29: Multicore Context Switch Time on QNX Neutrino in BMP mode without CRPD simulation.

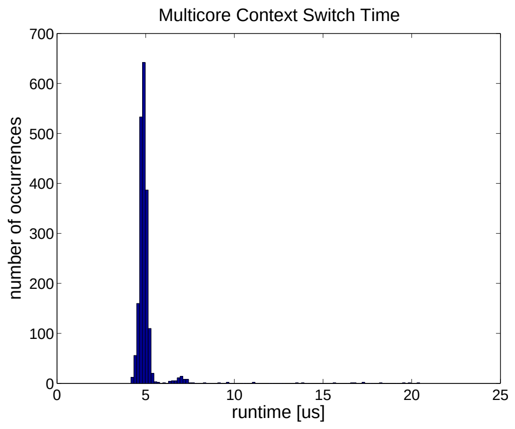

Figure 5.30: Multicore Context Switch Time on QNX Neutrino in BMP mode with CRPD simulation.

Multicore Context Switch Time in normal mode shows outliers below the main bar, as it can be seen in Figure 5.29. This is a result of the non synchronous start of both context switching task pairs. If one pair is ready on the first core, it does not mean that the other pair is ready on the second core. Until both pairs have started context switching the early pair behaves like the singlecore variant of this profiling procedure. In Figure 5.30 it can be seen that these negative outliers vanished. This is, because of the software cache invalidation strategy used in QNX Neutrino. The long runtime of the cache invalidation makes it possible that both task pairs are ready at the same time.

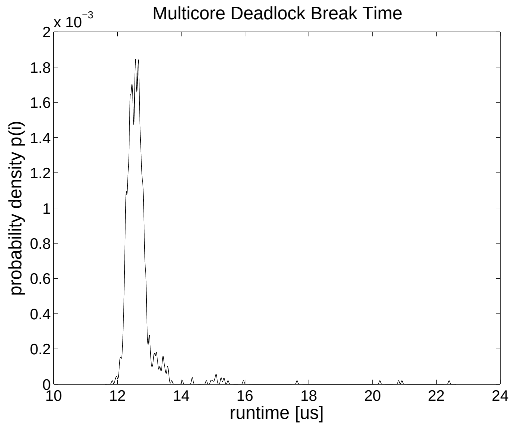

Figure 5.31: Multicore Deadlock Break Time on QNX Neutrino in BMP mode without CRPD simulation.

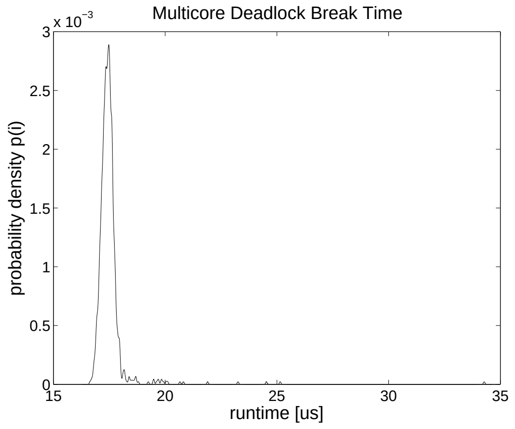

Figure 5.32: Multicore Deadlock Break Time on QNX Neutrino in BMP mode with CRPD simulation.

Multicore Deadlock Break Time compared with its singlecore variant shows a significant increase of runtime to nearly twice the time in average. However, the CRPD influences both, the singlecore and the multicore variant equally. Both increase the average by 4 μs. That means, that the CRPD simulation influences the same parts of the code in both variants.

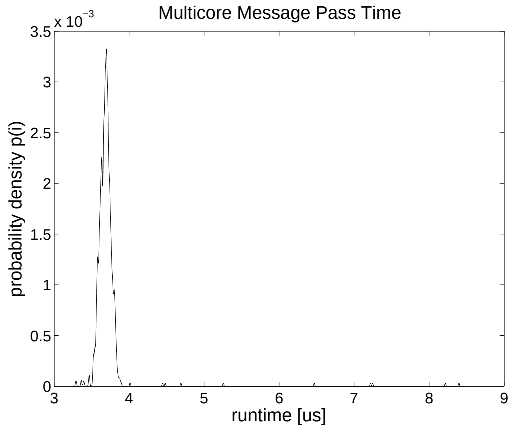

Figure 5.33: Multicore Message Pass Time on QNX Neutrino in BMP mode without CRPD simulation.

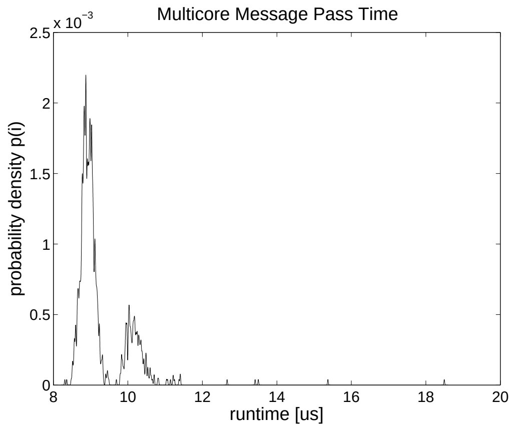

Figure 5.34: Multicore Message Pass Time on QNX Neutrino in BMP mode with CRPD simulation.

Multicore Message Pass Time shows a special behavior, because of the used message passing server for POSIX compatibility. Both, the singlecore and the multicore variant of this profiling procedure shows similar behavior.

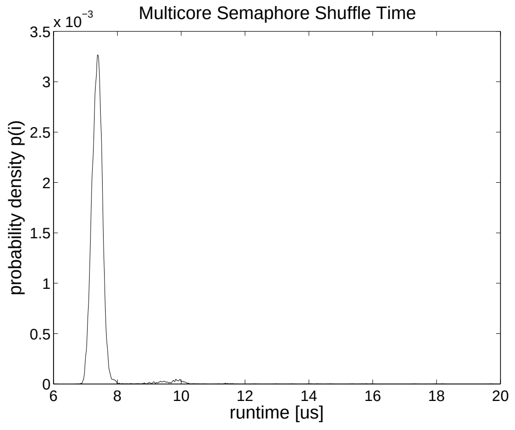

Figure 5.35: Multicore Semaphore Shuffle Time on QNX Neutrino in BMP mode without CRPD simulation.

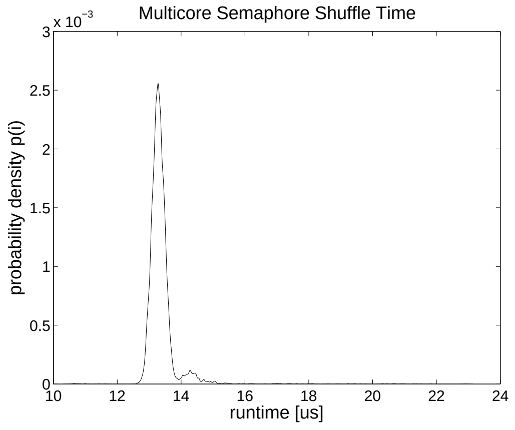

Figure 5.36: Multicore Semaphore Shuffle Time on QNX Neutrino in BMP mode with CRPD simulation.

Multicore Semaphore Shuffle Time shows a similar behavior compared to the singlecore variant. In Figure 5.35 it can be seen that the most measurements are around 7 μs, which is, as expected, faster than the deadlock break time. The CRPD has a observable, but not significant, influence on the measurement.

5.3.4 Discussion

When looking at the Figures 5.19, 5.21, 5.23, 5.25, 5.27, 5.29, 5.31, 5.33 and 5.35 it can be seen that all normal measurements exhibits outliers. In some cases the outliers of the normal measurements are nearly equal to the outliers seen in case of the related measurements with CRPD simulation. This can also be seen, when comparing Table 5.9

with Table 5.10. The CRPD simulation enforces the runtimes to be worse at all compared to the normal measurements and also stimulates the occurrence of outliers.

Figures 5.20, 5.22, 5.24, 5.26, 5.28, 5.30, 5.32, 5.34 show the effect of CRPD simulation on the singlecore and multicore profiling procedures. Considering the singlecore profiling procedures, the effect is similar between the singlecore and BMP mode of QNX Neutrino. However, it is a new finding how the CRPD simulation influences the multicore profiling procedures in contrast to the singlecore profiling procedures.

5.4 Overall Results

Table 5.13 shows two subtables containing the average runtimes and the standard deviations of the profiling procedures. Every cell containing a "-", means that this measurement is not available on the given platform, as for instance multicore measurements in singlecore operating modes or singlecore platforms.

Figure 5.37 presents a comparison between four profiling procedure executions. The comparison shows the differences of the average runtime between single and multicore mode of the RTOS and single and multicore type of the profiling procedure. It can be seen that the singlecore mode of QNX Neutrino is faster in every case than Neutrino in BMP operating mode. This behavior is expected, because when configuring Neutrino in singlecore mode, the kernel can use faster methods for locking and context switching. However, due to the closed source kernel, it is hard to find out exactly why this behavior occurs. Additionally, it can be seen that in BMP mode of the RTOS, the singlecore profiling procedures are faster than the multicore procedures, except of the message pass time, which is because of the utilized POSIX message passing server. That shows the influence of multicore processors and kernel modes to the performance of the RTOS services.

Furthermore, it can be seen in Table 5.13 that the Cache Related Preemption Delay (CRPD) simulation has a significant impact on the average runtimes of all measurements. Especially, the preemption time profiling procedure shows the importance of CRPD simulation on advanced CPU systems. When considering the standard deviations, it is conspicuous that the multicore profiling procedures exhibits large deviations when the CPRD simulation is turned on. However, the opposite behavior can be seen in case of SafeRTOS. The standard deviations of the semaphore shuffle time and the preemption time are smaller when the CPRD is turned on. That means that the CPRD simulation leads to a slower but more stable behavior in this special cases.

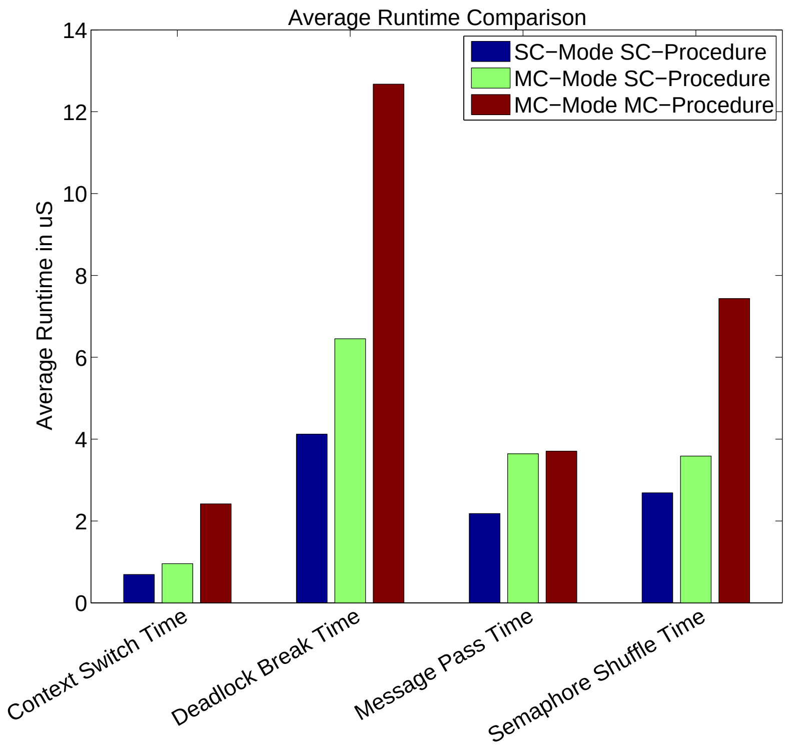

Figure 5.37: Comparison of singlecore and multicore average runtimes. SC and MC mode means the single and multicore mode of the QNX Neutrino RTOS and SC and MC procedure means the type of profiling procedure, as for instance singlecore and multicore context switch time.

Average Values of All Metrics

	SafeRTOS	CRPD	QNX SC	CRPD	QNX BMP	CRPD
Context Switch Time	5.9906	16.6877	0.69137	1.9595	0.95742	2.4651
Deadlock Break Time	-	-	4.1206	9.3095	6.4522	11.9868
Message Pass Time	24.9059	53.8553	2.1821	5.7478	3.6421	7.6591
Semaphore Shuffle Time	33.1433	70.8451	2.6886	5.8866	3.5853	7.2363
Preemption Time	13.3653	37.8991	0.75968	5.641	2.6589	10.1978
Multicore Context Switch Time	-	-	-	-	2.4178	5.014
Multicore Deadlock Break Time	-	-	-	-	12.6751	17.5024
Multicore Message Pass Time	-	-	-	-	3.7057	9.2457
Multicore Semaphore Shuffle Time	-	-	-	-	7.4351	13.3712

Standard Deviation of All Metrics

	SafeRTOS	CRPD	QNX SC	CRPD	QNX BMP	CRPD
Context Switch Time	0.059597	0.14994	0.089556	0.26936	0.032952	0.31031
Deadlock Break Time	-	-	0.30464	0.53257	0.51687	0.66253
Message Pass Time	0.34413	0.34607	0.43196	0.63221	0.41408	0.51604
Semaphore Shuffle Time	0.50584	0.31825	0.10695	0.44394	0.45496	0.54942
Preemption Time	0.27158	0.14807	0.15195	0.59398	0.3584	0.66037
Multicore Context Switch Time	-	-	-	-	0.61892	1.0385
Multicore Deadlock Break Time	-	-	-	-	0.6944	0.77552
Multicore Message Pass Time	-	-	-	-	0.2926	0.71359
Multicore Semaphore Shuffle Time	-	-	-	-	0.51694	0.53426

Table 5.13: Overview of average runtimes and standard deviations of all metrics. The first table shows the average runtimes of every metric. The second table shows the standard deviations of the metrics. Both tables shows the values grouped by operating system and CRPD. The first column contains the normal measurements done with the corresponding RTOS. The second column contains the measurements with CRPD simulation turned on. A cell containing a "-" means, that this measurement is not available on the given platform.

5.5 Problems

During the evaluation of the results, we found several problems of the implementation and design of the profiling procedures. The most error prone part of the work is the locking mechanism that is used to guarantee the correct order of the execution. It is necessary to be aware of the fact that RTOSs might start the created tasks in an unexpected order. Therefore, we used mutexes, semapores and barriers to guarantee the correct order.

The developer of a profiling procedure has to take care about every possible race condition during the execution of the procedure. Especially the multicore procedures exhibit more possible race conditions due to the real parallel execution. In order to avoid making such errors we defined multiple execution phases for the profiling procedure tasks.

1. **Initialization:** Initializing all elements used by the task.
2. **Loop Begin:** Every procedure is called multiple times. In this phase all initializations for the next iteration are made, as for instance locks are acquired in order to prevent tasks from early starts.
3. **Start:** At this point, a barrier is placed in order to guarantee that all tasks start at the same time.
4. **Run:** The execution of the procedure.
5. **End:** All tasks have to wait together again in order to not disturb any of the running measurements.
6. **Loop End:** After all tasks have arrived at this point, the counters are incremented and the loop starts again.

It is clear that this phases make use of multiple locking mechanisms. Numerous locking mechanisms use kernel traps to enter a critical region. This kernel activity might influence the measurement by adding up a certain offset to the values. Therefore, the designer of a profiling procedure should think about which kind of locking mechanism is suitable for the new procedure.

There are various signs for a wrong usage of locking mechanisms or a hidden race condition. All our profiling procedures have one thing in common: the measurements condense to single accommodation around the average value, if the number of measurements is high enough. If there are more than one accommodation, this is considerably sign for an error in the usage of the locking mechanisms. Another possible issue are cyclic appearing outliers. Cyclic appearance indicates either the wrong usage of locking mechanism or a problem with caches. If the locking mechanism is correct, the caching issue might be caused by an operating system service.

Chapter 6

Conclusion

Problem Statement

When we started this work, we wanted to know if it is possible to write a portable and meaningful profiling suite for RTOS services on single and multicore CPUs. Therefore, we wanted to know which are meaningful metrics for RTOS and which one can be adopted or extended to measure multicore RTOS services. Most related works focus on small scale and singlecore hardware. However, advanced hardware raises additional predictability problems and makes it harder to measure the runtimes of RTOS services.

Design and Implementation

In order to solve these problems, we did comprehensive research about existing related works about profiling and benchmarking of RTOS services. The result was a list of potential important RTOS metrics. This list contains five RTOS metrics that are used most frequently by other works. All five metrics are part of the Rhealstone benchmark found in [KP89].

However, before we could start the implementation of the chosen RTOS metrics, we evaluated the predictability issues of advanced processors. Therefore, we researched related works for documented predictability issues and how to handle them. Most issues can be avoided by turning off special processor features or switching to better predictable versions of the feature. We found that there is one problem on preemptive systems that can not be switched off. It is called the Cache Related Preemption Delay (CRPD). The CRPD occurs when a task is preempted by another task that invalidates at least parts of the cache. In order to address this issue, we implemented a hardware based CRPD simulation on the iMX28 and a software based CRPD simulation on the iMX6Q platform. We designed our profiling suite in a way that all profiling procedures contain an optional CPRD simulation.

The design of the profiling suite proposes also multicore capable profiling procedures. We evaluated which of the five chosen RTOS metrics are suitable for multicore adoption. The result of the evaluation was that four of the five metrics are suitable to be adopted. With that, the design of the profiling suite could be finished.

We implemented the profiling suite in a way to be portable between hardware and software platforms. Therefore, we made use of preprocessor macros to replace the functions and variables in the implementation of our profiling procedures. All function calls and variables are abstracted by preprocessor macros.

In order to provide a flexible evaluation of the measurements, we decided to deliver all measurement values in raw form to the user. Additionally, the runtimes of the measurement tool are provide in order to evaluate the precision of the measured values. Hence, the user of the profiling suite has full control about the evaluation.

Results

During the implementation, we recognized that the usage of preprocessor macros has its drawbacks, but after all it is a suitable and useful way to implement portable profiling code for RTOS services in C. The drawbacks are mainly the complexity and the problematic error handling. This is, because every RTOS has got a different way of handling errors. A benefit is that it is easy to switch on or off features as for instance the CRPD simulation.

The CRPD simulation turns out to be a useful tool for profiling at all. When the CRPD simulation is activated, every profiling procedure takes more time. How much time is consumed, depends on multiple factors as for instance the implementation of the CRPD simulation, the position of the CRPD macro inside the code and the profiling procedure itself.

The results show that the CRPD simulation promotes the presence of outliers. However, any measurement of an outlier is no guarantee that the worst case runtime has been found. The measurements, especially the CRPD measurements, show significant differences between the singlecore and multicore kernel of QNX Neutrino. As expected, the singlecore kernel exhibits less overhead compared to the multicore kernel. SafeRTOS on the iMX28 platform shows only a few outliers in the normal measurements and also dense accommodations in the CRPD measurements. This is a result of the hardware based CRPD simulation method.

Another interesting result is the different behavior of the singlecore and the multicore profiling procedures. We found, that all multicore procedures take more time than the singlecore equivalent on the same kernel. Except of the multicore variant of the message pass time procedure.

We made all multicore measurements using the global timer of the Cortex-A9 MPCore cluster. The time to read this timer exhibits deviations, that are handled by us using our methods described in [Str14].

During the implementation we found two different kinds of problems, the measurement accuracy and the locking mechanisms. The measurement techniques we tried to use at first were way to imprecise and caused problems on synchronous multicore measurements. Therefore we put effort in the research of measurement techniques that results in the work [Str14]. The other problem with the locking mechanisms was, that there are numerous race conditions that can easily be overseen and also that excessive usage of locking mechanisms might influence the operation of the respective profiling procedure. We saw that behavior for instance in case of the multicore semaphore shuffle time, where a semaphore behavior is profiled and others are also used for the locking mechanism. After a redesign of the affected procedures we obtained improved runtimes and higher stability. However, especially in case of multicore profiling procedures it is hard to say how much the present locking mechanism influences the operation of the profiling procedure.

Chapter 7

Further Work

The two most interesting results of this work are the multicore profiling procedures for RTOS services and the effects of the CRPD simulation. Therefore, this is the point where further work should start. At first it would be interesting to find more useful multicore services of RTOS that can be profiled. Possible ideas would be for instance, a multicore mutex shuffle time, a multicore message passing scenario to more than one receiver or multicore barrier wake up time. All these procedures are easily possible to be implemented using our profiling suite, the remaining work is to find more useful scenarios.

The CRPD simulation turned out to be a useful tool when trying to promote long runtimes and outliers. The hardware based CRPD simulation, shows good results on all tests. However, the software based cache invalidation method could be better and more reliable. An idea would be to utilize tasks to preempt the running profiling procedure and invalidate caches. An example for such a behavior can be seen in [Str14] within the application profiling measurements where the task runtime is measured. When using tasks to invalidate caches makes it necessary to reconsider locking mechanisms for the procedures.

In order to see which method performs best, it would also be interesting to compare cache invalidation methods with each other, as for instance comparing hardware with software based methods. Also multicore environments makes it hard to invalidate all L1 caches of every core. An idea would be to implement a cache invalidation method on all cores involved in the profiling procedure.

In order to check how good a CRPD simulation works, it would be good to count the cache misses during a profiling procedure. The PMU could be used for such a functionality. It includes a counter that can count all L1 cache misses during a given time period. This functionality would also help the user of the profiling suite to identify caching problem within the RTOS service. In this work we focused only on L1 cache invalidation, but advanced processors as for instance the iMX6Q has also a second level cache. It would be interesting to invalidate also these L2 cache before starting a profiling procedure.

Beside the CPRD simulation, it is possible to port the profiling suite to different hardware and RTOS. An idea is to port it to Linux with a real-time kernel and to FreeRTOS, because these two operating systems are closely related to the two already ported RTOS. Furthermore, it would be interesting to port the profiling suite to the Intel x86 architecture.

Another interesting application of the profiling suite would be to use it to verify the results of a static timing analysis tool. It would be possible to use a tool as for instance

Chronos, which is used in [Cho+13]. This tool is compatible with one of our processors, the i.MX28.

Bibliography

[ARM08] ARM Limited. *ARM926EJ-S - Technical Reference Manual.* r0p5. ARM Limited. Cambridge, England, United Kingdom, 2008.

[ARM10] ARM Limited. *Cortex-A9 - Technical Reference Manual.* r2p2. ARM Limited. Cambridge, England, United Kingdom, 2010.

[Bar07] H.-J. Bartsch. *Taschenbuch Mathematischer Formeln.* 17th edition. Carl Hanser Verlag, 2007. ISBN: 9783446408951.

[Ber06] C. Berg. "PLRU Cache Domino Effects". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06).* Edited by F. Mueller. Volume 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. ISBN: 9783939897033. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2006.672`. URL: `http://drops.dagstuhl.de/opus/volltexte/2006/672`.

[Bog13] T. J. Boger. "Rhealstone Benchmarking of FreeRTOS and the Xilinx Zynq Extensible Processing Platform". Master Thesis. Temple University, 2013. URL: `http://digital.library.temple.edu/cdm/ref/collection/p245801coll10/id/216539`.

[Cho+13] L. K. Chong et al. "Integrated Timing Analysis of Application and Operating Systems Code". In: *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th.* 2013, pages 128–139. DOI: `10.1109/RTSS.2013.21`.

[Cul+10] C. Cullmann et al. "Predictability Considerations in the Design of Multi-Core Embedded Systems". In: *Ingénieurs de l'Automobile* 807 (2010), pages 36–42. ISSN: 0020-1200.

[DD13] S.-D. David and A.-T. David. "Evaluation of MPSoC Operating Systems". In: *International Journal of Engineering Practical Research* 2.2 (2013). URL: `www.seipub.org/ijepr/Download.aspx?ID=2186`.

[Den14] W. Denk. *U-Boot the Universal Boot Loader.* `http://www.denx.de/wiki/U-Boot`. Accessed: 2014-10-22. 2014.

[Dev14] B. Devices. *BoundaryDevices Website.* `http://boundarydevices.com/products/sabre-lite-imx6-sbc/`. [Online; accessed 31-March-2014]. 2014.

[EEM14] EEMBC. *EEMBC product page.* `http://www.eembc.org/products/`. [Online; accessed 14-March-2014]. 2014.

[Fre14] FreeRTOS. *FreeRTOS API Specification.* `http://www.freertos.org/a00106.html`. [Online; accessed 12-March-2014]. 2014.

[GC11] U. G. and B. C.F. “Context switching time and memory footprint comparison of Xilkernel and uC/OS-II on MicroBlaze”. In: *Electrical and Electronics Engineering (ELECO), 2011 7th International Conference on.* IEEE, 2011, pages II–62–II–65.

[GNU14a] GNU. *GCC v4.7.3 Documentation.* `https://gcc.gnu.org/onlinedocs/gcc-4.7.3/gcc/`. Accessed: 2014-10-15. 2014.

[GNU14b] GNU. *GCC v4.7.3 Inline.* `https://gcc.gnu.org/onlinedocs/gcc-4.7.3/gcc/Inline.html`. Accessed: 2014-10-03. 2014.

[GH10] R. Gumzej and W. A. Halang. “Performance Metrics for Real-time Systems”. English. In: *Real-time Systems’ Quality of Service.* Springer London, 2010, pages 9–16. ISBN: 978-1-84882-847-6. DOI: `10.1007/978-1-84882-848-3_2`. URL: `http://dx.doi.org/10.1007/978-1-84882-848-3_2`.

[Gut+01] M. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on.* IEEE, 2001, pages 3–14. DOI: `10.1109/WWC.2001.990739`.

[Hoe13] Hoeller, Andrea. *Evaluation of Multicore Technology in Safety-Critical Controller for Hydro-Electrical Power Plants.* Technical report. Graz University of Technology, 2013.

[IEE96] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language].* 1996, page 784. ISBN: 1-55937-573-6. URL: `http://standards.ieee.org/reading/ieee/std_public/description/posix/9945-1-1996_desc.html`.

[JJN08] S. Jarp, R. Jurga, and A. Nowak. “Perfmon2: a leap forward in performance monitoring”. In: *Journal of Physics: Conference Series.* Volume 119. 4. IOP Publishing. 2008. DOI: `10.1088/1742-6596/119/4/042017`.

[JTC99] JTC 1/SC 22/WG 14. *ISO/IEC 9899:1999: Programming languages – C.* Technical report. International Organization for Standards, 1999.

[KP89] R. P. Kar and K. Porter. “Rhealstone-a real-time benchmarking proposal”. In: *Dr Dobbs Journal* 14.2 (1989), page 14.

[Kha+09] Z. Khan et al. “Identification and Analysis of Performance Metrics for Real Time Operating System”. In: *Emerging Technologies, 2009. ICET 2009. International Conference on.* Islamabad: IEEE, 2009, pages 183–187. ISBN: 9781424456307. DOI: `10.1109/ICET.2009.5353177`. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5353177`.

[Kim+10] J.-Y. Kim et al. “A Commercial-Off-the-Shelf(COTS) dedication of a QNX real time operating system (RTOS)”. In: *Reliability, Safety and Hazard (ICRESH), 2010 2nd International Conference on.* Mumbai: IEEE, 2010, pages 123–126. ISBN: 9781424456307. DOI: `10.1109/ICRESH.2010.5779528`.

[Kin13] T. King. “Cache Partitioning Enhances Multicore Performance”. In: *COTS Journal of Military Electronics and Computing* (2013).

[LO11] P. Laplante and S. Ovaska. *Real-Time Systems Design and Analysis: Tools for the Practitioner*. Wiley, 2011. ISBN: 9780470768648. URL: `http://books.google.de/books?id=Ez6-aSfbqtsC`.

[LA14] S. C. Lee and N. B. Z. Ali. “Test Methodology for Real-Time Operating System”. In: *Bulletin of Networking, Computing, Systems, and Software* 3.1 (2014), pages 10–12. ISSN: 2186-5140.

[LY03] Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003. ISBN: 1578201241.

[Mar+11] M. Marieska et al. “On performance of kernel based and embedded Real-Time Operating System: Benchmarking and analysis”. In: *Advanced Computer Science and Information System (ICACSIS), 2011 International Conference on*. Jakarta: IEEE, 2011, pages 401–406. ISBN: 9789791421119.

[Mat14] Mathworks. *Matlab 2013a*. `http://www.mathworks.de/support/sysreq/sv-r2013a/`. Accessed: 2014-10-17. 2014.

[Moy13] B. Moyer. *Real World Multicore Embedded Systems*. 1st edition. Newnes, May 8, 2013. ISBN: 9780124160187.

[Nem+06] F. Nemer et al. “PapaBench: a Free Real-Time Benchmark”. In: OpenAccess Series in Informatics (OASIcs) 4 (2006). Edited by F. Mueller. ISSN: 2190-6807. DOI: `http://dx.doi.org/10.4230/OASIcs.WCET.2006.678`. URL: `http://drops.dagstuhl.de/opus/volltexte/2006/678`.

[Oer12] O. Oernvall. “Benchmarking Real-time Operating Systems for use in Radio Base Station applications”. 72. Master Thesis. Göteborg: Department of Computer Science and Engineering, Computing Science (Chalmers), Chalmers University of Technology, 2012. URL: `http://publications.lib.chalmers.se/records/fulltext/158029.pdf`.

[QNX14a] QNX. *Neutrino RTOS*. `http://www.qnx.com/products/neutrino-rtos/index.html`. Accessed: 2014-09-24. 2014.

[QNX14b] QNX. *Neutrino v6.6.0 Documentation*. `http://www.qnx.com/developers/docs/660/index.jsp`. Accessed: 2014-09-24. 2014.

[QNX14c] QNX. *QNX® Neutrino® RTOS, C Library Reference*. 6.6.0. Accessed: 2014-11-05. QNX Software Systems Limited. 1001 Farrar Road Ottawa, Ontario K2K 0B3 Canada, 2014.

[QNX14d] QNX. *QNX SDP 6.6.0 BSPs*. 6.6.0. Accessed: 2014-11-05. QNX Software Systems Limited. 1001 Farrar Road Ottawa, Ontario K2K 0B3 Canada, 2014.

[Rei08] J. Reineke. *Caches in WCET Analysis: Predictability, Competitiveness, Sensitivity*. epubli, 2008. ISBN: 9783941071698.

[ST02] K. Sakamura and H. Takada. *μITRON v4.0 Specification*. `http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf`. [Online; accessed 12-March-2014]. 2002.

[Sem12] F. Semiconductor. *i.MX28 Applications Processors for Consumer Products - Silicon Version 1.2*. 3rd edition. Freescale Semiconductor. Austin, Texas, USA, 2012.

[Sem14] F. Semiconductor. *i.MX6Dual/Quad Automotive and Infotainment Applications Processors*. 3rd edition. Freescale Semiconductor. Austin, Texas, USA, 2014.

[Sta08] E. Stadlober. *Wahrscheinlichkeitstheorie für Informatiker*. University Lecture. Münzgrabenstraße 11/III A-8010 Graz, 2008.

[Str14] F. Strasser. "Evaluation of Runtime Measurement Techniques for RTOS and Application Profiling on Multicore Platforms". Seminar Project. Technical University Graz, Institute for Technical Informatics, 2014.

[TTN09] S.-L. Tan and B Tran Nguyen. "Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers". In: *Micro, IEEE* PP.99 (2009), pages 1–1. ISSN: 0272-1732.

[Tan07] A. S. Tanenbaum. *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780136006633.

[THT08] H. Tomiyama, S. Honda, and H. Takada. "Real-time operating systems for multicore embedded systems". In: *SoC Design Conference, 2008. ISOCC '08. International*. Volume 01. 2008, pages I–62–I–67. DOI: `10.1109/SOCDC.2008.4815573`.

[THW10] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments". In: *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pages 207–216. DOI: `10.1109/ICPPW.2010.38`.

[Wal12] C. Walls. *What is your rtos' footprint*. `http://fplreflib.findlay.co.uk/articles/44275/P29-30.pdf`. [Online; accessed 19-March-2014]. 2012.

[Wei90] N. Weiderman. "Hartstone: Synthetic Benchmark Requirements for Hard Real-time Applications". In: *Proceedings of the Working Group on Ada Performance Issues 1990*. Baltimore, Maryland, USA: ACM, 1990, pages 126–136. ISBN: 089791354X. DOI: `10.1145/322807.322853`. URL: `http://doi.acm.org/10.1145/322807.322853`.

[Wil+08] R. Wilhelm et al. "The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: `10.1145/1347375.1347389`. URL: `http://doi.acm.org/10.1145/1347375.1347389`.

[Wit12] Wittenstein. *SAFERTOS USER MANUAL FOR THE GCC I.MX28 PRODUCT VARIANT*. 34-172-MAN-003-016-i1.0. Wittenstein HighIntegritySystems. Brown's Court, Long Ashton Business Park, Yanley Lane, Long Ashton, Bristol, BS41 9LB, UK, 2012.

[Wit14] Wittenstein. *SafeRTOS v4.6*. `http://www.highintegritysystems.com/safertos/`. Accessed: 2014-10-17. 2014.

[Xu+08] T. Xu et al. "Performance benchmarking of FreeRTOS and its Hardware Abstraction". Master Thesis. Technische Universität Eindhoven Department of Mathematics, Computer Science, and Department of Electrical Engineering, 2008. URL: `http://alexandria.tue.nl/extra2/afstversl/wsk-i/xu2008.pdf`.

[YZ08] J. Yan and W. Zhang. “WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches”. In: *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS ’08. IEEE.* St Louis, MO: IEEE, 2008, pages 80–89. DOI: `10.1109/RTAS.2008.6`.