DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

DLR

Bachelorarbeit

im Studiengang Informationstechnik

an der Dualen Hochschule Baden-Württemberg Mannheim

Implementierung und Evaluierung eines weakly-hard aware Schedulers in RTEMS

von

Carlo Brokering
Matrikel Nr.: 7551721
s212698@student.dhbw-mannheim.de

Kurs:	TINF21IT1
Bearbeitungszeitraum:	04.06.2024 bis 27.08.2024
Ausbildungsbetrieb:	Deutsches Zentrum für Luft- und Raumfahrt
Abteilung:	Institut für Softwaretechnologie
Betreuer:	Gabriel Moyano und Mark Geiger

Abstrakt

Deutsch — Diese Arbeit beschäftigt sich mit der Implementierung und Evaluierung eines weakly-hard aware Schedulers für das Real Time Operating System RTEMS mittels der bereitgestellten Scheduler API. Der verwendete *job-level fixed priority* Scheduling-Algorithmus erlaubt es, m verpasste Deadlines in einem Fenster von K Ausführungen zu tolerieren. Abhängig von der Anzahl der verpassten Deadlines werden den Jobs dynamisch Prioritäten zugewiesen. Zur Evaluierung des Schedulers wurden verschiedene Experimente sowohl in der simulierten Umgebung QEMU als auch auf der Hardwareplattform Xilinx Zynq ZedBoard durchgeführt. Während dieser Experimente wurden Tracedaten des Schedulers aufgezeichnet. Der Fokus der Evaluierung liegt auf der korrekten Funktionsweise des Schedulers, seinem Overhead und seiner Eignung für den Einsatz in realen Systemen. Zur Validierung dieser Aspekte wurden die Tracedaten visualisiert und analysiert. Die Ergebnisse zeigen, dass der Scheduler korrekt arbeitet und einen geringen Overhead im Mikrosekundenbereich aufweist.

Englisch — This thesis addresses the implementation and evaluation of a weakly-hard aware scheduler for the Real Time Operating System RTEMS using its scheduler API. The used *job-level fixed priority* scheduling algorithm allows to tolerate m missed deadlines in a window of K executions. Depending on the number of missed deadlines, the jobs are dynamically prioritised. To evaluate the scheduler, various experiments were carried out both in the simulated QEMU environment and on the Xilinx Zynq ZedBoard hardware platform. Trace data of the scheduler was recorded during these experiments. The focus of the evaluation is on the correct functioning of the scheduler, its overhead and its suitability for use in real systems. To validate these aspects, the trace data was visualised and analysed. The results show that the scheduler works correctly and has a low overhead in the microsecond range.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

API:	Application Programming Interface
CPU:	Central Processing Unit
CTF:	Common Trace Format
EDF:	Earliest Deadline First
GCC:	GNU Compiler Collection
GDB:	GNU Debugger
ISR:	Interrupt Service Routine
OS:	Operating System
QEMU:	Quick Emulator
RBTree:	Red-Black Tree
RMS:	Rate Monotonic Scheduling
RTEMS:	Real-Time Executive for Multiprocessor Systems
RTOS:	Real-Time Operating System
SoC:	System on a Chip
TCB:	Thread Control Block
WH:	Weakly Hard
WHA:	weakly-hard aware

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

Implementierung und Evaluierung eines weakly-hard aware Schedulers in RTEMS

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Braunschweig, den 27.08.2024

Ort, Datum	Unterschrift

1. Einleitung

1.1. Motivation

Der Einsatz kommerzieller Hardware in der Raumfahrt ist Gegenstand zahlreicher Forschungsarbeiten. Diese bietet im Vergleich zu konventioneller Hardware mehr Leistung bei geringeren Kosten. Da Anwendungen in Raumfahrtsystemen anspruchsvolle Regelalgorithmen mit Real-Time Anforderungen verwenden, werden Real-Time Operating Systems (RTOS) wie RTEMS eingesetzt. Dabei handelt es sich um ein Open-Source-RTOS, das verschiedene Prozessorarchitekturen unterstützt.

Üblicherweise sind solche Systeme als Hard Real-Time Systeme ausgelegt, bei denen ausgeführte Tasks jeweils eine Deadline haben, bis zu der ihre Ausführung abgeschlossen sein muss. Das Hard Real-Time Modell erlaubt dabei keine verpassten Deadlines. In vielen realen Anwendungsfällen die als Hard Real-Time Systeme modelliert sind, wären verpasste Deadlines bis zu einem gewissen Grad tolerierbar.

Dies wird durch das Weakly Hard Real-Time Modell von Bernat [1] ermöglicht, das für einen Task das Verpassen von m Deadlines in einem Fenster von K Ausführungen toleriert. Basierend auf dem Weakly Hard Modell hat Moyano [2] einen *global job-level fixed priority* Scheduling-Algorithmus vorgestellt. Dieser weist den Jobs, d.h. den ausgeführten Instanzen eines Tasks, je nach Anzahl der verpassten Deadlines vordefinierte Prioritäten zu und unterstützt Multiprozessorsysteme.

Um die Funktionalität und den praktischen Nutzen dieses Weakly Hard Schedulers beurteilen zu können, muss er implementiert werden. Da eine Implementierung für Multiprozessorsysteme sehr aufwendig ist, wird er in dieser Arbeit zunächst als Einprozessor-Scheduler entwickelt. Dies kann bereits ausreichend Aufschluss über die Arbeitsweise geben und bietet eine Grundlage für zukünftige Forschungen.

Für die Implementierung bietet sich RTEMS an, da der Quellcode für den Kernel frei verfügbar ist. Darüber hinaus definiert RTEMS eine Scheduler-API, an die die benötigten Scheduler-Operationen angebunden werden können.

1.2. Ziel der Arbeit

Im Rahmen dieser Arbeit soll die Scheduler API von RTEMS 6 verwendet werden, um einen Weakly-Hard-Aware (WHA) Scheduler nach dem von Moyano vorgestellten Algorithmus zu implementieren. Dieser muss in der Lage sein, periodische Jobs auszuführen und deren Prioritäten zu aktualisieren, abhängig davon, ob ein Job seine Deadline verpasst hat oder nicht.

Dazu muss der Scheduler in der Lage sein, die Ausführung der periodischen Jobs zu überwachen und zu verwalten. Es muss sichergestellt sein, dass die Arbeitsweise des Schedulers korrekt ist. Außerdem ist es wichtig, dass der Scheduler einen möglichst geringen Overhead hat, da dies in direktem Zusammenhang mit seiner Leistungsfähigkeit steht.

Um die Erfüllung der Anforderungen beurteilen zu können, sollen Tests durchgeführt und ausgewertet werden. Zusätzlich soll die Flexibilität der RTEMS Scheduler API in Bezug auf die Implementierung des Weakly-Hard-Aware Schedulers evaluiert werden.

1.3. Vorgehen

Zunächst wird ein Verständnis für die Implementierung der Scheduler API und die Funktionsweise des RTEMS Kernels erarbeitet. Dies beinhaltet die für die Implementierung notwendigen Strukturen und Operationen sowie die Mechanismen zur Verwaltung periodischer Jobs.

Anschließend wird der WHA Scheduler basierend auf den Anforderungen des *job-level fixed priority* Algorithmus implementiert. Zusätzlich wird ein WHA Job Manager implementiert, der zur Definition der Weakly Hard Parameter m und K und zur Verwaltung der periodischen Jobs benötigt wird. Für einen schnellen Entwicklungsprozess werden RTEMS Testanwendungen zuerst in QEMU ausgeführt und mit GDB debugged.

Anschließend werden Hardwaretests auf dem Xilinx Zynq ZedBoard durchgeführt, um zu demonstrieren, dass der Scheduler auch auf Hardware funktioniert und um seine Funktionsweise zu evaluieren. Um die korrekte Implementierung des Schedulers sicherzustellen und seine Performance zu messen, werden Testapplikationen ausgeführt und die Ausführung des WHA Schedulers mittels Tracing aufgezeichnet. Die gewonnenen Daten werden abschließend visualisiert und ausgewertet.

2. Grundlagen

2.1. Real-Time Systeme

In Real-Time Systemen hängt die Korrektheit des Systems nicht nur vom logischen Ergebnis der Berechnungen ab, sondern auch von der Zeit, in der das Ergebnis erzeugt wird [3]. Die zeitliche Komponente dieser Systeme steht dabei in engem Zusammenhang mit ihrer Umgebung. Viele dieser Systeme stehen in ständiger Interaktion mit der physischen Welt, wobei die Erfassung der Umgebung z.B. durch Sensoren oder andere Eingaben erfolgt. Deren Verarbeitung muss innerhalb einer definierten Zeitspanne erfolgen, um rechtzeitig auf Änderungen in der Umgebung reagieren zu können [4].

Ein *Real-Time Task* in einem Real-Time System ist eine Aufgabe, die durch eine Deadline charakterisiert ist, d.h. die maximale Zeit, innerhalb der der Task seine Ausführung abschließen muss. Die Zeit, die ein Task tatsächlich zur Ausführung benötigt, wird als Response Time bezeichnet [4].

Generell sollte natürlich die Response Time eines Tasks so gering wie möglich sein, wichtig ist jedoch, dass sie kürzer als die Deadline ist. Um garantieren zu, dass ein Taskset die zeitlichen Anforderungen eines Real-Time Systems einhält, wird die *Response Time Analyse* verwendet [5]. Diese wurde für verschiedene Scheduling-Algorithmen in Arbeiten wie [6] und [7] untersucht. Ein wichtiger Parameter ist dabei die *Worst Case Execution Time* (WCET) der Tasks, also die jeweils maximale Response Time der Tasks, die unter anderem zur Bestimmung der Ausführbarkeit eines Tasksatzes verwendet wird.

Wenn die Zeitvorgaben nicht eingehalten werden können, kann es zu kritischen Systemfehlern führen oder die Qualität der berechneten Daten sinkt. Aus den Konsequenzen, die aus dem Verpassen einer Deadline resultieren, lassen sich Real-Time Tasks generell in Kategorien unterteilen, die in Abbildung 1 verdeutlicht sind:

- **Hard Real-Time** – Die Deadlines müssen unbedingt eingehalten werden, da eine Nichteinhaltung zu schwerwiegenden oder irreversiblen Konsequenzen führen kann.
- **Firm Real-Time** – Die nach einer verpassten Deadline gelieferten Ergebnisse sind für das System ohne Nutzen, aber führen zu keinem Schaden.
- **Soft Real-Time** – Die Ergebnisse können auch nach einer verpassten Deadline noch verwendet werden, führen aber zu einer Verringerung der Systemleistung [4].

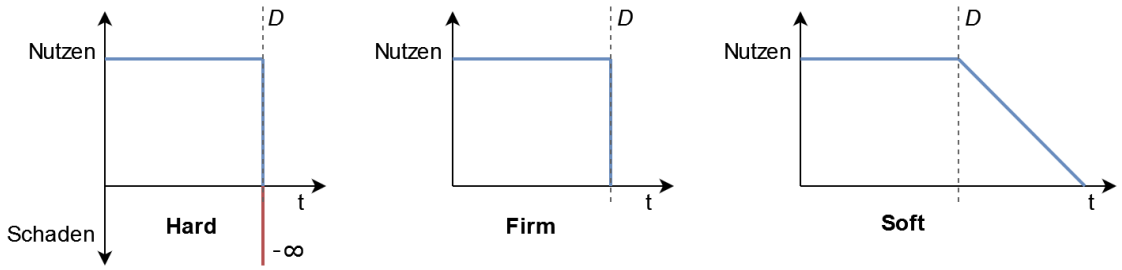


Abbildung 1: Hard, Firm und Soft Real-Time

Wie von Burns [8] beschrieben, können die Auswirkungen von verpassten Deadlines in Hard-Real-Time Systemen weiter differenziert werden. Dies hängt im Wesentlichen davon ab, welche Auswirkungen eine verpasste Deadline auf das System hat. Außerdem wird die Möglichkeit hybrider Systeme beschrieben, die gemischte Charakteristika aufweisen.

2.1.1. Real-Time Operating Systems

Viele Real-Time Operating Systems (RTOS) basieren auf Kerneln anderer, nicht real-time-fähiger Betriebssysteme. Diese bieten wichtige Elemente wie Multitasking, prioritätsbasiertes Scheduling, Prozesskommunikation, -synchronisation sowie Ein- und Ausgabefunktionen. Beispielsweise stammt ein Teil des in dieser Arbeit verwendeten Betriebssystems RTEMS aus dem FreeBSD-Code [9].

Damit ein Kernel für die korrekte Ausführung von Echtzeitsystemen geeignet ist, muss er neben den üblichen Konzepten für Betriebssysteme unter anderem die folgenden Eigenschaften haben:

- **Pünktlichkeit** – Die Ergebnisse von Real-Time Anwendungen müssen zeitlich korrekt sein, was eine genaue Bewertung der Einhaltung von Deadlines erfordert. Dazu muss das Betriebssystem bestimmte Kernel-Mechanismen für das Zeitmanagement und die Behandlung der Zeitbeschränkungen von Tasks bereitstellen.
- **Vorhersehbarkeit** – Um für ein Real-Time System die Einhaltung aller kritischen Zeitvorgaben im Voraus zu garantieren, ist eine Real-Time Analyse notwendig. Dazu muss die Ausführung von Systemoperationen analysierbar und deren Konsequenz vorhersehbar sein. Dadurch wird für sicherheitskritische Anwendungen gewährleistet, dass alle zeitlichen Vorgaben erfüllt sind, bevor das System in Betrieb genommen wird [4].

- **Effizienz** – Häufig werden Real-Time Systeme auf Embedded Hardware ausgeführt. Damit ein RTOS-Kernel auch auf solchen Systemen leistungsfähig ist, sollte er möglichst effizient arbeiten. Dazu gehört, dass er klein und speichereffizient ist, beim Scheduling den Kontextwechsel schnell durchführt und schnell auf externe Interrupts reagiert.

Die meisten RTOS sind als harte Echtzeitsysteme konzipiert. Da diese die höchsten zeitlichen Anforderungen an das Betriebssystem stellen, können Hard Real-Time Systeme in der Regel auch Firm und Soft Real-Time Tasks ausführen.

2.1.2. Weakly Hard Real-Time

In Hard Real-Time Systemen wird angenommen, dass keine Deadlines verpasst werden dürfen. Allerdings könnten viele Hard Real Time Tasks das Verpassen von Deadlines tolerieren, sofern es in einem kontrollierten Maß passiert.

Unter dieser Annahme haben Bernat et. al. [1] eine *Weakly Hard Constraint* $\binom{n}{m}$ beschrieben, mit der ein Task in einem Fenster von m Ausführungen n Deadlines einhalten muss. Umgekehrt bedeutet $\overline{\binom{n}{m}}$, dass maximal n Deadlines in m Ausführungen verpasst werden dürfen.

In einem früheren Ansatz von Hamdaoui et. al. [10] wurde die Notation (m, K)-firm eingeführt, nach der m Deadlines in K aufeinanderfolgenden Ausführungen eingehalten werden sollen. Darauf basierend haben sie einen Scheduling Algorithmus für ein Kommunikationssystem vorgestellt, der die Priorität von Tasks erhöht, wenn es wahrscheinlich ist, dass m Deadlines verpasst werden.

Diese (m, K) Notation kann, wie in [1] beschrieben direkt auf $\binom{n}{m}$ übertragen werden, weshalb in dieser Arbeit die *Weakly Hard Constraint* eines Tasks τ_i mit

$$\binom{m_i}{K_i} \qquad \text{und} \qquad \overline{\binom{m_i}{K_i}}$$

notiert wird.

Die Weakly Hard Constraint wurde von Choi et. al. [11] verwendet, um einen Scheduling Algorithmus zu entwickeln, der Prioritäten auf Job-Level zuweist. Dabei beschreibt ein Job die Ausführung eines periodischen Tasks. Es wurde gezeigt, dass die Zuweisung von Prioritäten zu einzelnen Jobs die Effizienz und Flexibilität des Schedulings verbessert.

Eine Response Time Analyse zu einem *global job-level fixed priority* Scheduling-Algorithmus wurde von Moyano durchgeführt [2]. Dafür wurde die Weakly Hard Constraint verwendet, um Parameter für ein globales Scheduling zu berechen, was in Abschnitt 4.1. näher beschrieben wird. Dies wird als Grundlage für diese Arbeit verwendet.

2.2. Scheduling Algorithmen

Jedes multitaskingfähige Betriebssystem benötigt einen Scheduler, der die Ausführung der Tasks steuert. Die Hauptziele eines Schedulers sind die Maximierung des Durchsatzes sowie die Minimierung von Wartezeiten und Latenzen. Diese Anforderungen stehen jedoch oft in Konflikt zueinander, sodass der verwendete Scheduling-Algorithmus Kompromisse eingehen muss, um eine ausgeglichene Lösung bereitzustellen. Ein Real-Time-Scheduler muss zusätzlich sicherstellen, dass alle Prozesse ihre Deadlines einhalten.

Es gibt eine Vielzahl von Scheduling-Algorithmen und Variationen dieser Algorithmen. Obwohl das Scheduling für Ein- und Mehrprozessorsysteme grundsätzlich auf den gleichen Ansätzen basiert, erfordern Mehrprozessorsysteme zusätzlich eine Synchronisation der Ressourcen. In dieser Arbeit soll jedoch nur ein Algorithmus für Einprozessorsysteme implementiert werden, so dass die Ressourcensynchronisation nicht weiter behandelt wird.

In den folgenden Abschnitten werden die in RTOS häufig verwendeten Scheduling Algorithmen *Earliest Deadline First* (EDF) und *Rate Monotonic Scheduling* beschrieben. Beide bieten einfache und effiziente Methode zur Priorisierung von Real-Time Tasks und wurden bereits in verschiedenen Arbeiten wie [6], [12], [13] untersucht. Ihre RTEMS Implementierung wurde analysiert und als Codebasis für die Implementierung des Weakly-Hard Scherdulers verwendet.

2.2.1. Earliest Deadline First

Bei Earliest Deadline First handelt es sich um einen dynamischen Scheduling Algorithmus, der zu jedem Zeitpunkt den Task mit der frühesten Deadline ausgeführt. Dazu werden alle Tasks nach ihrer Deadline sortiert, wobei die Tasks mit der nächstliegenden Deadline eine höhere Priorität erhalten. Liu und Layland haben dies 1973 erstmals beschrieben [14]. Sie zeigten, dass die EDF-Strategie eine optimale Methode für das Scheduling von periodischen Tasks in Echtzeitsystemen ist, solange die Gesamtlast des Systems nicht 100 % übersteigt.

2.2.2. Rate Monotonic Scheduling

Das Rate-Monotonic-Scheduling (RMS) ist ein Real-Time Scheduling-Algorithmus zur Verwaltung periodischer Tasks. Bei diesem Ansatz wird jedem Task eine feste Priorität zugewiesen, wobei die Priorität umso höher ist, je kürzer die Periode des Tasks ist. Die ausgeführte Instanz des periodischen Tasks wird als Job bezeichnet.

Ein Taskset τ mit n Tasks kann mit dem Rate Monotonic Manager garantiert ausgeführt werden, wenn die Gesamtauslastung

$$U(n) = \sum_{i=0}^{n} \frac{C_i}{T_i} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right)$$

ist, wobei C_i die Ausführungszeit eines Jobs und T_i die Periodenlänge eines Tasks ist. Mit zunehmender Anzahl von Aufgaben nähert sich die maximal mögliche Gesamtauslastung $\ln(2) \approx 69\%$ [15].

In der Praxis können jedoch viele Tasksets mit einer höheren Gesamtauslastung ausgeführt werden. Der durchschnittliche Grenzwert für die Prozessorauslastung eines zufällig generierten Tasksets liegt bei etwa 88% [14].

2.3. Xilinx Zynq ZedBoard

Der im Rahmen dieser Arbeit entwickelte RTEMS Scheduler soll auf Hardware getestet werden. Hierfür bietet sich das ZedBoard an, ein Entwicklungsboard mit dem System-on-a-Chip (SoC) Xilinx Zynq 7000 (Abbildung 2). Dabei handelt es sich um ein heterogenes SoC, das ein ARM Cortex A9 Prozessorsystem, ein FPGA und verschiedene konfigurierbare Schnittstellen in einem Chip vereint. Das Board verfügt über 512 MB RAM, 256 MB Quad-SPI Flash und bietet zahlreiche Schnittstellen des Xilinx Zynq. Unter den verschiedenen On-Board-Schnittstellen des ZedBoards befindet sich auch ein USB-JTAG Adapter, mit dem das SoC programmiert und debugged werden kann.

Dieses kommerziell erhältliche und leistungsstarke SoC wird unter anderem für den Einsatz in der Raumfahrt erforscht [16]. Beispielsweise kann das enthaltene FPGA als Hardwarebeschleuniger in der Bildverarbeitung eingesetzt werden [17]. Verglichen mit Hardware die üblicherweise in der Raumfahrt eingesetzt wird hat dieses SoC deutlich mehr Leistung, ist aber unter dem Einfluss von Strahlung fehleranfälliger [18].

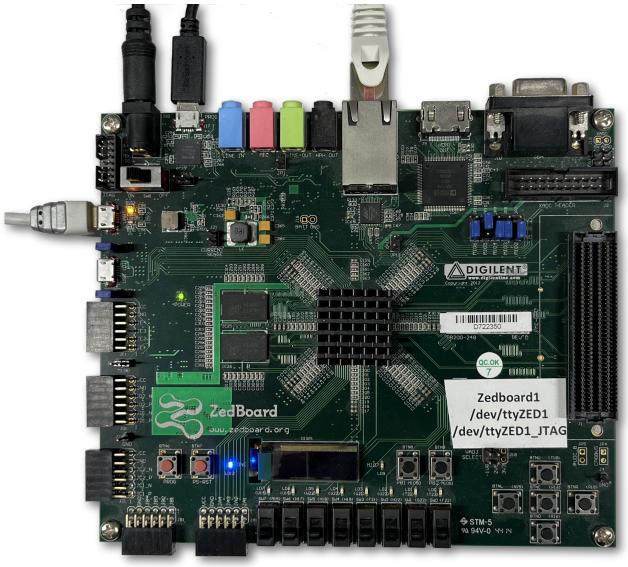


Abbildung 2: Xilinx Zynq ZedBoard

Um RTEMS-Anwendungen auf dem ZedBoard betreiben zu können, müssen diese zunächst in den Flash-Speicher geladen werden. Dazu wurde das ZedBoard mit dem Bootloader U-Boot[1] programmiert, der die kompilierte RTEMS-Anwendung von einem SFTP-Server in den Flash-Speicher lädt und ausführt.

2.4. Entwicklungsumgebung

Die Entwicklung von RTEMS-Programmen erfolgt auf einem Host-Computer, auf dem alle Tools, Bibliotheken und eine speziell für RTEMS eingerichtete Entwicklungsumgebung installiert sind. Auf diesem Host werden der RTEMS-Kernel und die Anwendungen für die jeweilige Zielplattform erstellt und getestet.

Der Quellcode des RTEMS-Kernels wird für die Entwicklung des Schedulers benötigt und sollte in der Entwicklungsumgebung zur Verfügung stehen. So kann die Funktionsweise des Kernels einschließlich des Schedulers durch Debugging nachvollzogen werden. Andererseits kann die Entwicklungsumgebung so automatische Vervollständigungen im Code vorschlagen.

2.4.1. RTEMS Installation

Die Installation auf dem Host umfasst mehrere Komponenten: den RTEMS Quellcode, die RTEMS Tool Suite und ein Board Support Package (BSP). Mit dem RTEMS Source

[1] https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842223/U-boot

Builder (RSB) kann man ein Paket bestehend aus Tool Suite und BSP vom Quellcode erstellen.

Die Tool Suite ist eine Sammlung von Softwaretools, die Compiler, Debugger und andere Tools enthält, um die Entwicklung und das Debugging von Anwendungen für RTEMS zu unterstützen. In diesem Projekt wurden insbesondere der GNU Cross-Compiler `arm-rtems-gcc` und der Debugger `arm-rtems-gdb` verwendet.

Ein BSP ist eine Sammlung von Softwarekomponenten, mit denen RTEMS-Anwendungen für eine bestimmte Hardware erstellt werden können. Darin sind Treiber, Bootloader und Konfigurationsdateien enthalten, die es dem Betriebssystem ermöglichen, das Gerät zu starten und auf die Schnittstellen zuzugreifen. In der Entwicklungsphase des Schedulers wird QEMU zum Debuggen verwendet und final wird auf dem Xilinx Zynq ZedBoard getestet. Entsprechend werden die BSPs `arm/xilinx_zynq_a9_qemu` und `arm/xilinx_zynq_zedboard` verwendet.

2.4.2. QEMU

RTEMS Anwendungen werden immer für die BSP-spezifische Zielplattform cross-kompiliert und verwenden im Binärcode den hardware-spezifischen Befehlssatz. Daher ist es nicht möglich, den Programmcode direkt auf dem Host-Computer auszuführen.

Der Quick Emulator (QEMU) ist eine Open Source Virtualisierungssoftware, die die Prozessorbefehle verschiedener Architekturen für die Ausführung auf einem Host übersetzt. Damit können für das BSP `arm/xilinx_zynq_a9_qemu` kompilierte Anwendungen simuliert und auf dem Host ausgeführt werden. QEMU bietet eine Anbindung an verschiedene Schnittstellen der Anwendung. Dazu gehören der Standard-I/O sowie ein GDB-Server zum Debuggen der Anwendung.

Durch den Einsatz von QEMU als Testumgebung kann der Entwicklungsprozess erheblich beschleunigt werden. Die benötigte Zeit zum Kompilieren, Deployen und Ausführen in der virtuellen Umgebung ist wesentlich geringer als auf der Hardware. Andererseits kann für das Debugging auf Entwicklungsboards wie dem ZedBoard die JTAG-Schnittstelle verwendet werden, der Zugriff darauf erfordert jedoch ein aufwändigeres Setup.

2.4.3. Debugging

Die Entwicklung und das Debugging von Anwendungen in einer IDE wie Visual Studio Code (VS Code) ist eine von vielen Entwicklern bevorzugte Methode. Dabei ist es prak-

tisch, die integrierten Debugging-Funktionen zur Steuerung und Analyse der in QEMU virtualisierten Anwendung zu nutzen.

```
qemu-system-arm \
  -s -S -no-reboot \
  -serial null -serial mon:stdio \
  -net none -nographic  \
  -M xilinx-zynq-a9 -m 512M \
  -kernel path/to/program.exe
```

Listing 1: Ausführung einer *Xilinx Zynq A9* Anwendung mit QEMU

Dazu wird die Anwendung wie in Listing 1 im Hintergrund in QEMU mit einem Debugging Server, der entsprechenden Hardware-Plattform `xilinx-zynq-a9` gestartet. Die IDE wird so konfiguriert, dass sie sich durch `arm_rtem_gdb` mit dem Server verbindet. In Visual Studio Code können Task- und Launch-Konfigurationen erstellt werden, um den gesamten Build- und Startprozess zu automatisieren [19]. Ein Terminal in VS Code ist direkt mit der Standard-I/O von QEMU verbunden, über das die Ausgabe des RTEMS-Programms angezeigt wird.

3. RTEMS Kernel

Real-Time Executive for Microprocessor Systems (RTEMS) ist ein quelloffenes Echtzeitbetriebssystem für eingebettete Systeme, das unter anderem in der Raumfahrt verwendet wird. Es unterstützt das standardisierte Application Programming Interface (API) POSIX und läuft auf 18 verschiedenen Prozessorarchitekturen, darunter ARM, PowerPC, Intel, RISC-V und SPARC.

Die RTEMS Classic API [20] ist die zentrale Programmierschnittstelle von RTEMS für die Entwicklung von Echtzeitanwendungen. Sie umfasst Schnittstellen für Taskmanagement, Interprozesskommunikation, Synchronisation und Zeitmanagement. Diese ermöglichen das Erstellen, Steuern und Scheduling von Tasks, die Kommunikation zwischen Thread über Message Queues, Events und Signale sowie die Synchronisation über Barrieren, Semaphoren und Mutexe.

Die Begriffe Task und Thread werden in RTEMS oft synonym verwendet, da sie dasselbe Konzept verwenden. Der Begriff Task wird häufig im Zusammenhang mit einer Aufgabe oder einem Objekt verwendet, das von der Anwendung ausgeführt wird. Der Begriff Thread wird dagegen meist im Zusammenhang mit dem Scheduler oder dem Kern verwendet.

3.0.1. Task Manager

Eine RTEMS-Anwendung startet normalerweise mit einem *Init*-Task, der das System initialisiert und dann weitere Tasks startet. Der Task Manager [21] der Classic API bietet dazu verschiedene Funktionen, um Tasks zu erstellen und zu verwalten. Ein neuer Task wird zunächst mit `rtemt_task_create` initiiert, wobei seine Eigenschaften zugewiesen bekommt, wie einen Namen, eine anfängliche Priorität und ob er präemptiv ist.

Die Priorität eines Tasks wird auf Benutzerebene zwischen 1 und 255 festgelegt. Intern im Scheduler können höhere Prioritäten auftreten, um spezielle Anforderungen und Scheduling-Strategien zu unterstützen. Ein niedrigerer Wert entspricht einer höheren Priorität.

Eigenschaften wie Präemption und Timeslicing, die den Ausführungsmodus des Tasks bestimmen, werden bei der Erstellung des Tasks festgelegt. Präemption erlaubt es dem Scheduler, einen laufenden Task zu unterbrechen, um die Ausführung eines höher priorisierten Tasks zu ermöglichen. Timeslicing teilt die CPU-Zeit gleichmäßig zwischen

Tasks gleicher Priorität auf, indem jedem Task ein bestimmter Anteil zugewiesen wird. Für den in dieser Arbeit entwickelten Scheduler werden alle Tasks mit Präemption und ohne Timeslicing ausgeführt.

Anschließend kann er mit `rtems_task_start` gestartet werden, wobei als Einstiegspunkt eine Funktion übergeben wird.

Verschiedene Funktionen erlauben es einer Anwendung, die Tasks zu steuern. Mit `rtems_task_suspend` kann ein Task blockiert werden, d.h. seine Ausführung wird angehalten, bis er durch eine andere Aktion wieder aktiviert wird. Die Funktion `rtems_task_wake_after` lässt einen Task für eine bestimmte Zeit warten, während der er blockiert ist. Andere Funktionen bieten die Möglichkeit den Scheduler oder die Priorität eines Tasks zu ändern.

3.0.2. Thread Control Block

Ein Task wird im RTEMS-Kernel durch die Datenstruktur *`Thread_Control`*, auch bekannt als Thread Control Block (TCB), repräsentiert, die alle Informationen über einen Task enthält, wie z.B. die ID, die Priorität, den aktuellen Zustand und andere kontextbezogene Daten. Der TCB wird unter anderem dazu verwendet, den Lifecycle des Threads zu überwachen und zu steuern. Im Folgenden werden nur die für diese Arbeit relevanten Felder des TCB beschrieben.

Das Feld `current_state` enthält den aktuellen Zustand des Threads und setzt sich bitweise aus einzelnen Zuständen zusammen, von denen die meisten in die Kategorien `STATES_LOCALLY_BLOCKED` und `STATES_BLOCKED` gruppiert sind. Der Zustand wird hauptsächlich verwendet, um festzustellen, ob eine bestimmte Operation am Thread durchführbar ist.

Die Priorität des Threads wird in zwei Feldern gespeichert. Das Feld `current_priority` enthält die aktuelle Priorität, die je nach Scheduler dynamisch angepasst werden kann. In `real_priority` wird die ursprüngliche Priorität gespeichert, die bei der Erstellung des Tasks festgelegt wurde.

Das Attribut `is_preemptible` legt fest, ob ein Thread (nicht-) präemptiv ist. Außerdem wird gespeichert, welcher Scheduler für den Thread zuständig ist [22].

3.0.3. Zustände eines Tasks

Der Lifecycle eines Tasks beschreibt die verschiedenen Zustände, die ein Thread während seiner Existenz durchläuft, sowie die Übergänge zwischen diesen Zuständen. In RTEMS muss sich ein Task immer in einem der fünf zulässigen Zustände befinden [21]:

- *Executing* – wird aktuell ausgeführt
- *Ready* – ist bereit zur Ausführung
- *Blocked* – wartet auf ein Ereignis oder eine Ressource
- *Dormant* – erstellter Task, der noch nicht gestartet wurde
- *Non-existent* – noch nicht erstellt oder gelöscht

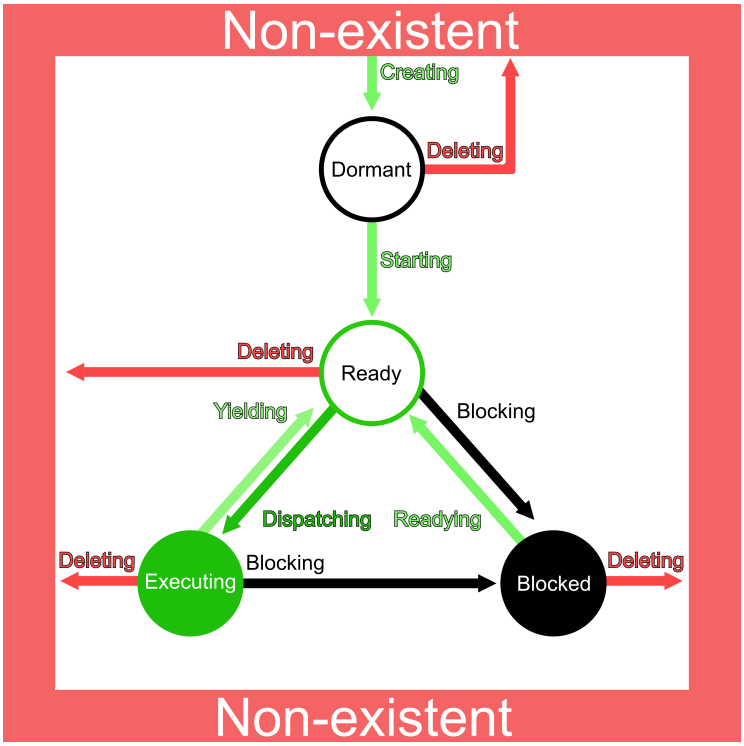


Abbildung 3: Übergänge der Taskzustände

Die möglichen Übergänge zwischen den Zuständen sind in Abbildung 3 dargestellt[2]. Ein Task befindet sich im Zustand *Non-existent* bevor er erstellt wurde oder nachdem er gelöscht wurde. Nach dem Erzeugen eines Tasks mit `rtems_task_create` wird dieser in den Zustand *Dormant* versetzt, in dem er zwar existiert, aber nicht aktiv um Systemressourcen konkurrieren kann. Durch das Starten des Tasks mit `rtems_task_start` wird dieser in den Zustand *Ready* versetzt, so dass er vom Scheduler zur Ausführung auf dem Prozessor ausgewählt werden kann [23].

[2]Bildquelle: [23]

Ein Task im Zustand *Blocked* kann vom Scheduler nicht zur Ausführung ausgewählt werden, da er auf Ressourcen oder Ereignisse wartet. Der laufende Task blockiert sich selbst durch Operationen, die ihn zum Warten zwingen. Dazu gehören die für diese Arbeit relevanten Operationen `rtems_task_wake_after` (Abschnitt 3.0.1.), sowie `rtems_rate_monotonic_period` und `rtems_barrier_wait`, deren Funktionen in späteren Abschnitten erläutert werden. Wenn der Blockierungsgrund nicht mehr besteht, wird der Task in den Zustand *Ready* versetzt und damit freigegeben [23].

Der Vorgang, wenn der Scheduler einen bereiten Task zur Ausführung auswählt, wird *Dispatch* genannt. Abhängig vom Scheduling-Algorithmus wird die aktuelle Priorität der Tasks bestimmt und der Task mit der höchsten Priorität ausgeführt. In einem Einprozessorsystem kann immer nur ein Task gleichzeitig ausgeführt werden, d.h. sich im Zustand *Running* befinden. Wenn der laufende Task präemptiv ist und ein Task mit höherer Priorität bereit ist, muss der laufende Task den Prozessor freigeben. Dieser Vorgang wird als *Yielding* bezeichnet.

Wenn alle Tasks blockiert sind, führt RTEMS einen Idle-Task aus. Dieser hat die niedrigste Priorität und wird unterbrochen, sobald ein anderer Task bereit ist.

3.0.4. Scheduler Plugin Framework

Der RTEMS-Kernel enthält ein Framework zur Unterstützung verschiedener Scheduler, in das auch eigene Scheduling-Algorithmen eingebunden werden können [23]. Das Framework verwendet dazu verschiedene Strukturen, die von jedem Scheduler individuell implementiert werden, um die gewünschte Arbeitsweise zu ermöglichen. Im RTEMS Internals Manual [22] beschreibt Bonato die spezifische Implementierung für den SMP Deterministic Priority Scheduler, die aufgrund der höheren Architekturanforderungen im Vergleich zu Single-Prozessor Schedulern wesentlich komplexer ist. Die allgemeinen Aufgaben der einzelnen Strukturen sind für alle Scheduler gleich und werden im Folgenden beschrieben.

Der *`Scheduler_Context`* enthält den aktuellen Schedulingzustand des Systems, also welche Tasks bereit sind. Dazu kann z.B. ein Tree oder eine Queue verwendet werden, in dem die Threads nach Priorität sortiert sind.

Die *`Scheduler_Operations`* sind die grundlegenden Operationen, die vom RTEMS Kernel aufgerufen werden und bilden die Schnittstelle zu den Schedulerspezifischen Implementierungen.

Jeder Task wird im Scheduler durch eine `Scheduler_Node` repräsentiert, die beim Erstellen des Tasks instanziiert wird. Sie enthält die notwendigen Informationen über den zugehörigen Task, die der Scheduling-Algorithmus benötigt, um den Task im `Scheduler_Context` unter den bereiten Tasks zu platzieren.

3.0.5. Rate Monotonic Manager

Mit dem Rate Monotonic Manager der RTEMS Classic API kann man periodische Tasks gemäß dem in Abschnitt 2.2.2. beschriebenen RMS Algorithmus erzeugen und verwalten.

Die Funktion `rtems_rate_monotonic_create` erstellt für den jeweiligen Task eine Periode, mit der der Ablauf des Tasks gesteuert wird. Im dafür zuständigen Period Control Block werden unter anderem der Zustand, die Periodendauer, Statistiken zur CPU-Nutzung, ein Timer und der zugehörige Task gespeichert.

Der Kern eines periodischen Task muss als Schleife konzipiert sein, die vor jeder Job-Ausführung die Funktion `rtems_rate_monotonic_period` aufruft, die die meiste Logik des RMS enthält. Beim erstmaligen Ausführen der `rtems_rate_monotonic_period` aktiviert diese den Timer der Periode und gibt die Ausführung des ersten Jobs frei.

Sobald ein Job abgeschlossen ist und der Task erneut die o.g. Funktion ausführt, wird er blockiert. Nach Ablauf der Periodendauer triggert der Timer die Interrupt Service Routine (ISR) `Rate_monotonic_Timeout`, den blockierten Task für die Ausführung des nächsten Jobs frei gibt, also den Task in den *Ready* Zustand versetzt. Außerdem führt sie ein eventuell notwendiges Update von der Prioritäten durch, wie in Abbildung 4 zu sehen ist.

Hat der Job seine Ausführung nicht innerhalb der Periodendauer beendet und damit die Deadline verpasst, befindet sich der Task immer noch im Zustand *Ready*. In diesem Fall wird die Deadline erneuert und der Zustand der Periode als *EXPIRED* gesetzt [24].

Durch die gesetzte Flag kann der aktuell laufende Job erkennen, dass er seine Deadline verpasst hat und die Ausführung beenden. Nach dem RMS Algorithmus, ist ein Satz an Tasks nur ausführbar, wenn alle Deadlines erreicht werden. Im Normalfall würde dann der Task mit der verpassten Deadline beendet werden.

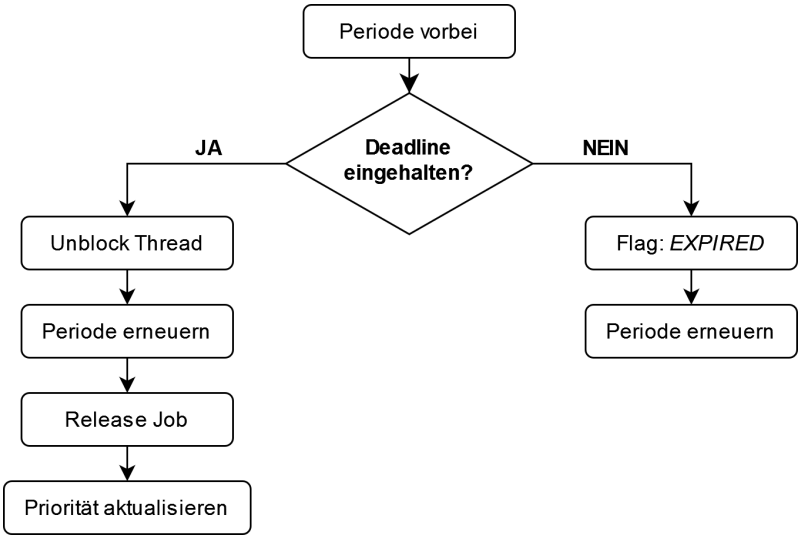


Abbildung 4: Ablaufdiagramm Rate Monotonic Manager Timeout

3.0.6. Clock Management

Der RTEMS Kernel enthält einen Clock Manager, der zur Steuerung aller zeitabhängigenn Prozesse und Elemente des Betriebssystems verwendet wird. Er wird in der Anwendungskonfiguration durch die Optionen

```
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MICROSECONDS_PER_TICK
```

aktiviert und bietet einen *Clock Tick*, der ein grobes Zeitmaß ist, das entsprechend der eingestellten Rate Ticks ausgibt und in einem Time Counter zählt. Zu den ausgegebenen Ticks werden registrierte Interrupt Service Routinen, wie z.B. Scheduling für das Timeslicing oder die Timeout Funktion des RMS ausgeführt. Zusätzlich können die aktuelle Unix-Uhrzeit oder andere Zeitzähler, die beim Booten initialisiert werden, über den Clock Manager abgefragt werden.

4. Anforderungen und Spezifikationen

Der WHA-Scheduler soll in RTEMS 6, der neuesten Version des Betriebssystems, implementiert werden. Diese Version ist nicht kompatibel mit früheren Versionen. Die Implementierung erfolgt in der Programmiersprache C.

Der Scheduling-Algorithmus muss einen geringen Overhead aufweisen, um die Belastung der Systemressourcen, insbesondere die beanspruchte CPU-Zeit und den Speicherbedarf, zu minimieren. Ein geringer Overhead verbessert die Skalierbarkeit und ermöglicht die Ausführung einer größeren Anzahl von Tasks, ohne die Systemleistung zu beeinträchtigen. Umgekehrt führt ein hoher Scheduler-Overhead dazu, dass die verfügbaren Zeitfenster für die Ausführung von Tasks kleiner werden. Dies erhöht die Wahrscheinlichkeit, dass Deadlines nicht eingehalten werden.

Die Logik des WHA-Schedulers basiert auf der Zuweisung und Verarbeitung verschiedener Parameter des Weakly Hard-Modells. In den folgenden Abschnitten werden diese beschrieben und die erforderliche Arbeitsweise des WHA-Schedulers erläutert.

4.1. Weakly-Hard Constraint

Nach dem weakly-hard Modell von Bernat et. al. [1] hat ein Task τ_i die weakly-hard Constraint $\overline{\binom{m_i}{K_i}}$, wobei m_i die Anzahl der tolerierbaren verpassten Deadlines in einer Sequenz von K_i Jobs ist. Diese wurde von Moyano auf $\overline{\binom{w_i}{w_i+h_i}}$ verschärft, die die Weakly-hard Constraint erfüllt und die Menge der möglichen Deadline-Sequenzen weiter einschränkt [2]. Dabei ist w_i die maximale Anzahl aufeinanderfolgender verpasster Deadlines, die durch

$$w_i = \max\left(\left\lfloor \frac{m_i}{K_i - m_i} \right\rfloor, 1\right)$$

berechnet wird. Die Anzahl der erforderlichen Deadline-Hits h_i, nachdem w_i verpasste Deadlines aufgetreten sind, wird mit

$$h_i = \left\lceil \frac{K_i - m_i}{m_i} \right\rceil$$

berechnet.

Weakly-hard Tasks können in zwei Kategorien unterteilt werden:

- **Low-tolerance Tasks** müssen im Fenster K_i mehr Deadlines erreichen, als sie verpassen dürfen, d.h. ein Verhältnis $\frac{m_i}{K_i} < 0.5$ und $m_i > 0$. Sie dürfen also nicht mehr als $w_i = 1$ Deadlines in Folge verpassen.
- **High-tolerance Tasks** haben ein Verhältnis $\frac{m_i}{K_i} \geq 0.5$ und tolerieren im Fenster K_i genauso viele oder mehr verpasste Deadlines als erreichte Deadlines. Für sie gilt immer $h_i = 1$.

4.2. Jobklassen und Joblevel

In dem von Moyano vorgestellten Scheduling-Algorithmus für weakly-hard Tasks [2] hat jeder Task τ_i einen festen Satz von Jobklassen jC_i, der verwendet wird um die Priorität eines Jobs zu bestimmen. Gemäß der weakly-hard Constraint hat ein Task

$$jC_i = K_i - m_i + 1$$

Jobklassen. Eine bestimmte Jobklasse wird als jC_i^q mit $q \in [0, K_i - m_i]$ notiert. Den Jobklassen werden die Prioritäten des Tasks zugeordnet, wobei diejenigen mit einem niedrigen q-Wert eine höhere Priorität erhalten. Dementsprechend hat die Jobklasse $jC_i^{q=0}$ die höchste und $jC_i^{q=K_i-m_i}$ die niedrigste Priorität. Unabhängig von den Tasks ist jede Priorität ist einmalig vergeben.

Um die aktuelle Jobklasse eines Tasks festzulegen, wird die Joblevel Variable jl_i verwendet, die je nach verpassen oder einhalten der Deadlines aktualisiert wird. Das Joblevel wird für jeden Tasks mit

$$jl_i = -(h_i - 1)$$

initialisiert. Wenn ein Job seine Deadline einhält, wird jl_i um 1 erhöht, bis er $K_i - m_i$ erreicht. Generell gilt also für den Joblevel-Wert:

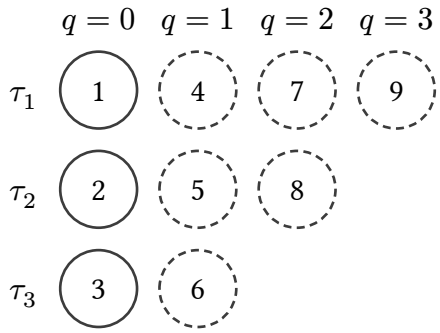
$$jl_i \in \{x \in \mathbb{Z} \mid -(h-1) \leq x \leq K - m\}$$

Sobald ein Task w_i Deadlines verpasst hat, wird das ursprüngliche Joblevel wiederhergestellt. Dazu wird ein Zähler für die Anzahl der verpassten Deadlines benötigt. Dieser muss nach dem Einhalten von h_i Deadlines auf Null zurückgesetzt werden.

Die aktuelle Jobklasse eines Tasks τ_i wird mit

$$jC_i^q \text{ mit } q = \max(0, jl_i)$$

ausgewählt. Die Zuordnung der Prioritäten erfolgt gleichmäßig über die Jobklassen, wie in Tabelle 1 dargestellt. Das bedeutet, dass alle Prioritäten in einer Jobklasse niedriger sind als die Prioritäten in den vorhergehenden Jobklassen. Ein niedriger Wert hat dabei eine hohe Priorität. Tasks mit den meisten Jobklassen erhalten jeweils die höchste und niedrigste Priorität. In der Abbildung stellen die durchgezogenen Kreise die initialen Jobklassen der Tasks dar – diese haben entsprechend die höchste Priorität.



Task	m_i	K_i	jC_i
τ_1	2	5	4
τ_1	1	3	3
τ_1	2	3	2

Tabelle 1: Prioritäten von Tasks mit unterschiedlich vielen Jobklassen

4.3. Aktualisierung der Joblevel und Prioritäten

Aus den oben beschriebenen Definitionen von Weakly Hard Constraint, Jobklassen und Joblevels ergeben sich die Anforderungen an den zu implementierenden WHA Scheduling-Algorithmus. Zunächst müssen die Weakly Hard Parameter für jeden Task initialisiert werden. Dazu benötigt jeder Task für seine Jobklassen einen eigenen Satz von Prioritäten, die abhängig vom Joblevel verwendet werden.

Die Hauptaufgabe des Schedulers besteht in der Aktualisierung der Joblevel, abhängig davon, ob ausgeführte Jobs ihre Deadlines eingehalten haben oder nicht. Eine Änderung des Joblevels führt anschließend zu einer Anpassung der Priorität.

Dabei muss der Scheduler die Weakly Hard Constraint einhalten. Wie beschrieben, wird das Joblevel nach jeder eingehaltenen Deadline bis zum entsprechenden Maximalwert erhöht. Nachdem ein Task w_i Deadlines verpasst hat, muss das Joblevel auf den Initialwert zurückgesetzt werden. Dazu benötigt der Scheduling-Algorithmus für jeden Task einen Zähler, der die verpassten Deadlines erfasst. Dieser Zähler wird auf 0 zurückgesetzt, nachdem h_i Deadlines eingehalten wurden. Da das Joblevel immer mit dem Initialwert $-(h_i - 1)$ startet, erfolgt das Zurücksetzen wenn $jl_i = 1$ erreicht.

Eine Anforderung an den WHA Scheduler ist, dass die Aktualisierung des Joblevels und der Priorität immer am Ende einer Periode erfolgt. Mit dem Rate Monotonic Manager

können periodische Jobs ausgeführt werden, aber eine Aktualisierung der Prioritäten erfolgt nur dann, wenn eine Deadline eingehalten wird oder wenn der Job erneut ausgeführt wird und `rtems_rate_monotonic_period` aufruft.

Wenn ein Job seine Deadline verpasst, kann das beispielsweise daran liegen, dass aktuell ein anderer Thread mit höherer Priorität ausgeführt wird. Unter Verwendung des Rate Monotonic Managers würde der Thread mit der verpassten Deadline seine Priorität allerdings erst ändern können, wenn der höher priorisierte Thread den Prozessor freigibt. Daher ist eine andere Implementierung erforderlich, die eine korrekte Aktualisierung der Joblevel und Prioritäten am Ende einer Periode gewährleistet.

4.4. Verhalten beim Verpassen einer Deadline

In Weakly Hard Scheduling-Algorithmen kann auf eine verpasste Deadline auf unterschiedliche Weise reagiert werden. Eine Möglichkeit besteht darin, den Job sofort zu stoppen, was als Kill Job Verhalten bezeichnet. Hierbei wird die Ausführung des Jobs abgebrochen, um den Prozessor freizugeben und zu verhindern, dass das System in einen instabilen Zustand gerät. Dadurch kann wird sichergestellt, dass andere Jobs weiter ausgeführt werden können, idealerweise ohne ihre Deadlines zu verpassen.

In Abbildung 5 ist ein theoretisches Beispiel dargestellt, wie der WHA Scheduler bei einer verpassten Deadline und Kill Job arbeiten sollte. Beide Tasks (T_1 und T_2) haben das Verhältnis $m_i/K_i = 1/2$, senken und erhöhen ihre Prioritäten also nach nach einer verpassten bzw. eingehaltenen Deadline. Die Pfeile ↕ markieren jeweils das Ende einer Periode und gleichzeitig den Beginn der nächsten. Rote Pfeile bedeuten, dass die Deadline nicht eingehalten wurde.

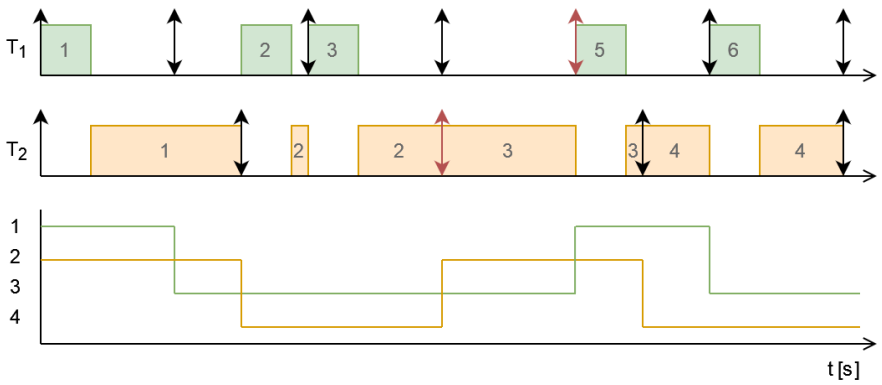


Abbildung 5: Kill Job nach verpasster Deadline

Man sieht, dass Job 2 von T_2 seine Deadline nicht einhält, woraufhin die Priorität erhöht wird und der nächste Job von T_2 sofort beginnt. Dies führt jedoch dazu, dass Job 4 von T_1 nicht ausgeführt werden kann und seine Deadline ebenfalls verpasst. Der Job 4 wird ebenfalls abgebrochen und sofort Job 5 gestartet, da T_1 dann die höchste Priorität hat.

Eine weitere Möglichkeit, mit verpassten Deadlines umzugehen, ist das Skip Next Verhalten. Dabei wird der Job, der die Deadline verpasst hat, weiter ausgeführt und der nächste Job desselben Tasks übersprungen. Damit können z.B. Datenverarbeitungen abgeschlossen werden, deren Ergebnisse bei einem Kill Job verloren gehen würden.

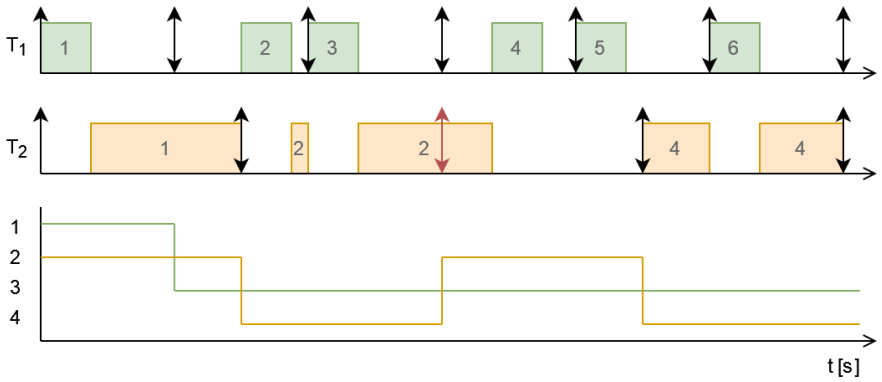


Abbildung 6: Skip Next nach verpasster Deadline

Ein theoretisches Beispiel für Skip Next ist in Abbildung 6 dargestellt. Wie in der vorherigen Abbildung verpasst Job 2 von T_2 seine Deadline, woraufhin die Priorität erhöht wird. Da T_2 nun die höchste Priorität hat, kann der Job weiter ausgeführt werden. Der folgende Job 3 wird entprechend übersprungen.

Koren et. al. [13] haben dieses Verhalten für verschiedene Scheduling Algorithmen untersucht und gezeigt, dass dadurch Überlastungen reduziert werden können. Dies ist auch in Abbildung 6 zu erkennen, da T_1 keine Deadline verpasst. Der Einsatz dieses Verfahrens ist jedoch an weitere Bedingungen geknüpft, wie z.B. eine maximale Anzahl von Jobs, die nacheinander übersprungen werden dürfen.

Da die von Moyano untersuchte Echtzeitanalyse des Weakly Hard Schedulers mit dem Kill Job Verfahren arbeitet [2] und die Verwendung von Skip Next an weitere Bedingungen geknüpft ist, wird in dieser Arbeit das Kill Job Verfahren verwendet. Die Implementierung des Kill Job Verfahrens muss auf Anwendungsebene erfolgen. Dies liegt zum einen daran, dass ein Job nur ein Teil eines laufenden Threads ist und keine eigenständige Instanz, die der Scheduler beenden kann. Zum anderen muss ein Job beim

Beenden die Möglichkeit haben, den Thread in einen stabilen Zustand zu überführen. Beispielsweise müssen Datenstrukturen so zurückgesetzt werden, dass der nächste Job wieder mit ihnen arbeiten kann.

5. Implementierung

Der weakly-hard aware (WHA) Scheduler besteht aus mehreren Elementen, die jeweils unterschiedliche Aufgaben erfüllen und in diesem Abschnitt beschrieben werden. Alle für den WHA-Scheduler implementierte Strukturen und Funktionen werden mit dem Präfix *Scheduler_WHA* benannt.

5.1. Scheduler Strukturen

Die in Abschnitt 3.0.4. beschriebenen Kernel-Strukturen `Scheduler_Context` und `Scheduler_Node` enthalten die für das Scheduling notwendigen Basisdaten. Darüber hinaus benötigen Scheduling-Algorithmen je nach Arbeitsweise zusätzliche Informationen über den Systemzustand und die Threads. Dazu sind scheduler-spezifische Strukturen erforderlich, die diese Daten enthalten und die Kernel-Strukturen als Member verwenden. In den folgenden Abschnitten werden die Strukturen des WHA-Schedulers beschrieben.

5.1.1. Scheduler Context

Der Scheduler Context repräsentiert den aktuellen Zustand des Systems, also welche Threads bereit für die Ausführung sind. Für den WHA-Scheduler gilt, dass jede Priorität einmalig ist. Daher eignet sich ein Tree, um die bereit stehenden Threads zu ordnen.

Im Scheduler-Kontext des RTEMS EDF-Schedulers wird ein Red-Black Tree verwendet, der ein spezieller binärer Suchbaum ist. Da seine Höhe maximal $2\log(n+1)$ beträgt, kann er mit einer Laufzeit von $O(\log n)$ durchsucht werden [25]. Dies garantiert auch bei vielen Threads eine geringe Laufzeit. Eine Implementierung des Red-Black Tree (RBTree) ist im RTEMS Kernel enthalten und wird auch für den WHA-Scheduler verwendet.

```
typedef struct {
  Scheduler_Context Base;
  RBTree_Control Ready;
} Scheduler_WHA_Context;
```

Listing 2: Struktur *Scheduler_WHA_Context*

Listing 2 zeigt die Implementierung des `Scheduler_WHA_Context`. Die zur Ausfürung bereiten Threads sind im RBTree `Ready` gespeichert. Außerdem ist die Kernelstruktut `Scheduler_Context` als Member `Base` enthalten.

5.1.2. Scheduler Node

Jeder Thread wird im RTEMS Scheduler durch eine Scheduler Node repräsentiert. Die für den WHA Scheduler spezifische Struktur heißt `Scheduler_WHA_Node` und ist in Listing 3 dargestellt.

```
typedef struct {
    Scheduler_Node Base;
    Weakly_hard_parameters Thread_parameters;
    RBTree_Node Node;
    Priority_Control priority;
} Scheduler_WHA_Node;
```

Listing 3: Struktur *Scheduler_WHA_Node*

Die *Scheduler_Node* ist darin als Member `Base` enthalten. In der Struktur `Weakly_hard_parameters`, die im Anschluss beschrieben wird, sind alle für das WHA-Scheduling notwendigen Parameter gespeichert. Die `RBTree_Node Node` wird im `Scheduler_WHA_Context` in den RBTree `Ready` eingefügt und verwendet, um die `Scheduler_WHA_Node` aus dem RBTree zu extrahieren. In der `priority` ist die aktuelle Priorität des Threads gespeichert.

5.1.3. Weakly-hard Parameters

In der in Listing 4 abgebildeten Struktur `Weakly_hard_parameters` sind alle Parameter enthalten, die für das WHA-Scheduling notwendig sind und in Abschnitt 4. erläutert wurden. Der Pointer `job_priorities` zeigt auf ein Array, das die zu den Jobklassen zugehörigen Prioritäten enthält. Dementsprechend muss es genau so viele Elemente enthalten, wie es Jobklassen gibt, dessen Anzahl in `job_classes` gespeichert ist.

Die Variable `missed_deadlines` wird mit 0 initialisiert und zählt die verpassten Deadlines. Sie wird auf 0 zurückgesetzt, sobald h_i Deadlines eingehalten wurden. In der Variable `job_level` ist das aktuelle Joblevel des Tasks gespeichert. Es wird mit `initial_job_level` initialisiert und auch beim Verpassen von w_i Deadlines auf diesen Wert zurückgesetzt. Alle Weakly-hard Parameter werden durch die Constraint $\overline{\binom{m_i}{K_i}}$ berechnet, die aber selbst nicht in den Parametern gespeichert wird.

```
typedef struct {
    uint32_t h;
    uint32_t w;
    uint32_t job_classes;
    int32_t job_level;
    int32_t initial_job_level;
    uint32_t missed_deadlines;
    Priority_Control *job_priorities;
} Weakly_hard_parameters;
```

Listing 4: Struktur *Weakly_hard_parameters*

5.2. Operationen

Die in diesem Abschnitt erläuterten Operationen setzen die Kernfunktionalität des WHA Schedulers um. Sie haben direkten Zugriff auf die Bibliotheken des RTEMS Kernels und können von Anwendungen nicht direkt aufgerufen werden.

Im letzten Abschnitt wurde der *`Scheduler_WHA_Context`* beschrieben, in dem bereite Threads in einem RBTree organisiert sind - genau wie beim EDF Scheduler. Dieser RBTree wird vom Scheduling-Algorithmus an vielen Stellen verwendet. Daher wurden große Teile der in Abschnitt 5.2.1. und Abschnitt 5.2.2. beschriebenen Operationen aus der EDF-Schedulerimplementierung übernommen und für den WHA-Scheduler angepasst.

5.2.1. Scheduling Operationen

Im Folgenden sind die grundlegenden Operationen des WHA-Schedulers beschrieben, die an den RTEMS-Kernel angebunden und von diesem aufgerufen werden. Sie sind verantwortlich für die korrekte Durchführung aller Scheduling-Operationen gemäß den in Abschnitt 3.0.3. erläuterten Zuständen. Jede dieser Operationen erhält neben anderen

Argumenten ein `Thread_Control` Objekt, welches alle Daten zu dem zu behandelnden Thread enthält und für dessen Steuerung notwendig ist.

`Scheduler_WHA_Initialize` – Diese Funktion bereitet den WHA-Scheduler für den Betrieb vor. Dafür initialisiert sie die interne Datenstruktur `Scheduler_WHA_Context`, einschließlich des RBTrees, der die bereitstehenden Threads verwaltet.

`Scheduler_WHA_Node_initialize` – Bei der Erstellung eines Tasks durch die Funktion `rtems_task_create` ruft der Kernel diese Funktion auf, um eine `Scheduler_WHA_Node` zu erzeugen. Diese repräsentiert den Task im Scheduler und benötigt zusätzliche Parameter. Da die Funktion nur die initiale Priorität des Tasks erhält, wird er zunächst mit den WHA Parametern $\overline{\binom{m_i}{K_i}} = \binom{1}{1}$ und der gegebenen Priorität initialisiert. Die Anwendung kann später in einem zusätzlichen Konfigurationsschritt (s. Abschnitt 5.3.) die genauen WHA-Parameter für diesen Task setzen.

`Scheduler_WHA_Schedule` – Die Aufgabe dieser Funktion besteht darin, den nächsten Thread zur Ausführung auszuwählen, sofern der aktuelle präemptiv ist. Sie greift dafür auf den RBTree im Scheduler-Kontext zu und wählt den Thread mit der höchsten Priorität aus. Dieser wird als Heir festgelegt, sodass der RTEMS Kernel diesen Thread beim nächsten Dispatch ausführen kann.

`Scheduler_WHA_Yield` – Wenn ein aktuell laufender Thread die CPU durch `rtems_task_yield` freiwillig freigibt, z.B. während Timeslicing wird diese Funktion aufgerufen. Der Thread wird in den Zustand *Ready* versetzt, und der Scheduler wählt einen anderen Thread zur Ausführung aus.

`Scheduler_WHA_Block` – Nachdem ein Thread blockiert wurde, z.B. weil er auf eine Ressource wartet, kommt diese Funktion zum Einsatz. Sie entfernt den Thread aus dem RBTree der bereiten Threads und der Thread mit der nächsthöheren Priorität erhält den Prozessor.

`Scheduler_WHA_Unblock` – Um einen zuvor blockierten Thread wieder in den Zustand *Ready* zu versetzen, wird diese Funktion verwendet. Dazu wird die aktuelle Priorität des Threads gesetzt, welche sich zuvor geändert haben könnte, und der Thread wird in den RBTree eingefügt, sodass er vom Scheduler ausgewählt werden kann.

`Scheduler_WHA_Update_priority` – Die Priorität eines Threads wird mit dieser Funktion aktualisiert. Bei einer Prioritätsänderung wird der Thread entsprechend neu im RBTree positioniert, um die korrekte Scheduling-Reihenfolge sicherzustellen.

`Scheduler_WHA_Release_job` – Diese Funktion dient generell dazu, einen neuen Job zur Ausführung freizugeben. Sie aktualisiert die Priorität des Threads anhand des aktuellen Joblevels.

Die oben genannten Operationen `Scheduler_WHA_Schedule`, `-Yield`, `-Block`, `-Unblock` und `-Update_priority` verwenden die RTEMS Kernel Implementierungen für Uniprozessor Scheduler. Abhängig von der Operation definieren sie die Scheduler-spezifischen Handler und Parameter und übergeben diese an die Kernel-Implementierung, die die grundlegende Logik ausführt.

5.2.2. Zugriff auf Strukturen

Verschiedene Hilfsfunktionen ermöglichen es, auf die Scheduler Strukturen zuzugreifen bzw. sie zu extrahieren. Diese kommen häufig in anderen Operationen zum Einsatz und haben eine hohe Relevanz für den Schedulingalgorithmus. Obwohl sie recht wenig Code enthalten, fördert ihre gute Strukturierung die Testbarkeit und Wartbarkeit des Schedulers. Folgende Funktionen werden verwendet, um auf die *`Scheduler_WHA_Node`* eines Tasks und den *`Scheduler_WHA_Context`* zuzugreifen:

`Scheduler_WHA_Get_context` – Diese Funktion gibt den schedulerspezifischen Kontext *`Scheduler_WHA_Context`* zurück, der im entsprechenden Feld in der Struktur *`Scheduler_Control`* hinterlegt ist.

`Scheduler_WHA_Thread_get_node` – Diese Funktion wird verwendet, um die *`Scheduler_WHA_Node`* eines Threads zu erhalten. Dazu greift sie auf die im Thread Control festgelegte *`Scheduler_Node`* zu und führt ein Casting durch.

`Scheduler_WHA_Node_downcast` – Das Down-Casting einer *`Scheduler_Node`* zum Datentyp *`Scheduler_WHA_Node`* wird durch diese Funktion abstrahiert.

5.2.3. Zugriff auf RBTree

Zum Verwalten und Zugreifen auf die bereiten Threads, die im *`Scheduler_WHA_Context`* im RBTree `Ready` abgelegt sind, wurden folgende Funktionen implementiert:

`Scheduler_WHA_Enqueue` – Ein bereiter Thread wird mit dieser Funktion in den Red-Black Tree des Scheduler Kontext eingefügt. Dabei werden als Sortierungskriterium die Prioritäten der Knoten vergleichen, sodass ein Task mit minimalem Prioritätswert als äußerstes linkes Blatt eingefügt wird.

`Scheduler_WHA_Get_highest_ready` – Diese Funktion gibt den Thread mit der höchsten Priorität zurück. Dieser hat den minimalen Prioritätswert und befindet sich im Red-Black Tree dementsprechend ganz links.

`Scheduler_WHA_Extract` – Die Scheduler Node eines Threads wird mit dieser Funktion aus dem RBTree entfernt. Sie wird verwendet, wenn ein Thread aus dem Satz an bereiten Threads entfernt werden soll, oder ihr Wert im Anschluss aktualisiert wird.

5.2.4. WHA Spezifische Operationen

Neben den bereits genannten, gibt es verschiedene Operationen, die die Logik des WHA-Schedulers umsetzen. Diese Operationen sind verantwortlich für die Festlegung und Aktualisierung der WHA-Parameter eines Threads, sowie das .

`Scheduler_WHA_Node_initialize_values` – ist in Listing 5 dargestellt und initialisiert die *`Weakly_hard_parameters`* der Scheduler Node eines Threads entsprechend den in Abschnitt 4. beschriebenen Berechnungen. Die Werte `m` und `K` werden nur für die Berechnungen verwendet und nicht gespeichert. Der Pointer `priorities` wird zu `job_priorities` kopiert und kein neuer Speicher allokiert. Der Nutzer muss dementsprechend sicher stellen, dass das Array mit den Prioritäten bestehen bleibt.

```
void Scheduler_WHA_Node_initialize_values(
    Scheduler_WHA_Node *node,
    uint32_t            m,
    uint32_t            K,
    Priority_Control   *priorities
)
{
  Weakly_hard_parameters *params = &node->Thread_parameters;
  params->h  = ceil((double) (K - m) / (double) m);
  params->w  = max(floor((double) m / (double) (K - m)), 1);
  params->job_classes       = K - m + 1;
  params->job_priorities    = priorities;
  params->initial_job_level = -(params->h - 1);
  params->job_level         = params->initial_job_level;
  params->missed_deadlines  = 0;
}
```

Listing 5: Scheduler_WHA_Node_initialize_values

`Scheduler_WHA_Update_job_level` – aktualisiert das Joblevel eines Tasks τ_i abhängig davon, ob die Deadline eingehalten oder verpasst wurde. Bei einer eingehaltenen Deadline wird das Joblevel immer um 1 erhöht, bis es $jC_i - 1$ erreicht. Wenn anschließend das Joblevel $jl_i = 1$ ist bedeutet es, dass h_i Deadlines am Stück eingehalten wurden und der Zähler `missed_deadlines` wird zurückgesetzt. Verpassten Deadlines werden mit diesem Zähler erfasst und bei mindestens w_i verpassten Deadlines wird das ursprüngliche Joblevel wiederhergestellt. Der zugehörige Code ist in Listing 6 zu sehen.

```
void Scheduler_WHA_Update_job_level(
    Scheduler_WHA_Node *node,
    bool deadline_hit
)
{
  Weakly_hard_parameters *params = &node->Thread_parameters;

  if (deadline_hit) {
    if (params->job_level < (int32_t)(params->job_classes - 1)) {
      params->job_level++;
    }
    if (params->job_level == 1) {
      params->missed_deadlines = 0;
    }
  } else {
    params->missed_deadlines++;
    if (params->missed_deadlines >= params->w) {
      Scheduler_WHA_Restore_job_level(node);
    }
  }
}
```

Listing 6: Scheduler_WHA_Update_job_level

`Scheduler_WHA_Restore_job_level` – diese Funktion setzt das Joblevel auf den ursprünglichen Wert $jl_i = -(h_i - 1)$ zurück, der in den *`Weakly_hard_parameters`* in `initial_joblevel` gespeichert ist.

`Scheduler_WHA_Job_class_priority` – gibt die Priorität der aktuellen Jobklasse zurück und greift dafür auf das in `job_priorities` hinterlegte Array zu.

5.3. WHA Job Manager

Mit den zuvor beschriebenen Funktionen und Strukturen ist es bereits möglich einen Task vom RTEMS Kernel ausführen zu lassen, der mit den Parametern $\overline{\binom{m_i}{K_i}} = \binom{1}{1}$ initialisiert wurde. Damit ein weakly-hard Task periodische Jobs ausführen kann, wird ein WHA Job Manager benötigt, der die Parameter des Tasks spezifiziert, das Joblevel

mit der `Scheduler_WHA_Update_job_level` abhängig vom Einhalten bzw. Verpassen der Deadlines anpasst und die Priorität aktualisiert.

Für die Verwaltung von periodischen Jobs bietet die RTEMS Classic API den Rate Monotonic Manager, der in Abschnitt 3.0.5. beschrieben ist. Das Verhalten eines mit diesem Manager erstellten periodischen Jobs wird durch die Funktion `rtems_rate_monotonic_period` und die Timeout-Routine `Rate_monotonic_Timeout` gesteuert. Der funktionelle Ablauf, um einen weakly-hard Job zu verwalten, ist prinzipiell gleich wie beim Rate Monotonic Manager. Allerdings muss ein weakly-hard Job anders initialisiert und in der Timeout-Routine anders behandelt werden.

Da abgesehen von den spezifischen Anforderungen für weakly-hard Jobs der notwendige Ablauf ähnlich zum Rate Monotonic Manager sind, werden dessen Funktionalitäten für den WHA Job Manager größtenteils adaptiert und erweitert. Zur Steuerung der Perioden wird weiterhin die `rtems_rate_monotonic_period` in Kombination mit einer eigenen Timeout-Routine verwendet.

5.3.1. Initialisierung

Zur Initialisierung eines weakly-hard Job wurde die Funktion `rtems_wha_job_create` implementiert, dessen Funktionsheader in Listing 7 dargestellt ist. Sie erhält als Argumente einen Namen, einen Pointer zu einer `rtems_id`, die weakly-hard Constraint $\overline{\binom{m_i}{K_i}}$ und einen Pointer zu einem Array, das die Prioritäten des Tasks enthält.

```
rtems_status_code rtems_wha_job_create (rtems_name name,
                                        rtems_id *job_id,
                                        uint32_t m,
                                        uint32_t K,
                                        Priority_Control *priorities);
```

Listing 7: Erstellen eines weakly-hard Job

Ihre Funktionsweise lässt sich in drei wesentliche Schritte unterteilen:

1. Sie erstellt mit `rtems_rate_monotonic_create` eine Periode, die mit `name` benannt wird und ihre ID in `job_id` speichert. Diese Periode befindet sich nach dem Erstellen im inaktiven Zustand. Das bedeutet, dass auch für den Watchdog zwar als Routine die `Rate_monotonic_Timeout` festgelegt wurde, dieser aber noch nicht gestartet wurde.
2. Da der Watchdog noch inaktiv ist, kann im nächsten Schritt die Watchdog-Routine der Periode durch die WHA-spezifische Imple-

mentierung `WHA_Job_manager_Timeout` ersetzt werden. Dazu wird die Kontrollstruktur der Periode über die ID geholt und ein Pointer darauf als `Rate_monotonic_Control *the_period` definiert. Mit `_Watchdog_Initialize(&the_period->Timer, WHA_Job_manager_Timeout)` wird dann die Routine ersetzt.

3. Zuletzt werden die WHA Parameter des Tasks festgelegt, indem die `Scheduler_WHA_Node_initialize_values` mit `m`, `K` und dem Pointer `priorities` als übergebene Argumente aufgerufen wird.

5.3.2. Job Timeout Routine

Der WHA Job Manager erneuert steuert und erneuert die Perioden von weakly-hard Jobs durch die `WHA_Job_manager_Timeout` Routine. Ihre Funktionsweise ist zum Großteil von der Timeout-Routine des Rate Monotonic Manager adaptiert. Wie auch die originale Funktion des Rate Monotonic Managers wird der Timeout-Watchdog beim ersten Aufruf der `rtems_rate_monotonic_period` aktiviert. Der Watchdog ruft dadurch am Ende einer Periode die Timeout-Routine auf, welche den Watchdog jedes Mal erneuert, um nach der folgenden Periode wieder aufgerufen zu werden.

Abbildung 7 zeigt den Ablauf der `WHA_Job_manager_Timeout` Routine. Ihre erste Aufgabe ist, das Joblevel zu aktualisieren, abhängig davon ob der letzte Job seine Deadline eingehalten hat oder nicht. Dafür ruft sie die Funktion `Scheduler_WHA_Update_job_level` auf.

Wenn ein Job abgeschlossenen ist, d.h. seine Deadline eingehalten hat, hat der entsprechende Thread bereits durch die `rtems_rate_monotonic_period` blockiert. In diesem Fall gibt die Timeout-Routine den Thread wieder frei, damit der nächste Job ausgeführt werden kann. Wenn der aktuelle Job noch nicht abgeschlossen ist, bedeutet dies, dass die Deadline verpasst wurde. Damit der Job dies bei seiner Fortsetzung behandeln kann, wird die Flag *EXPIRED* gesetzt.

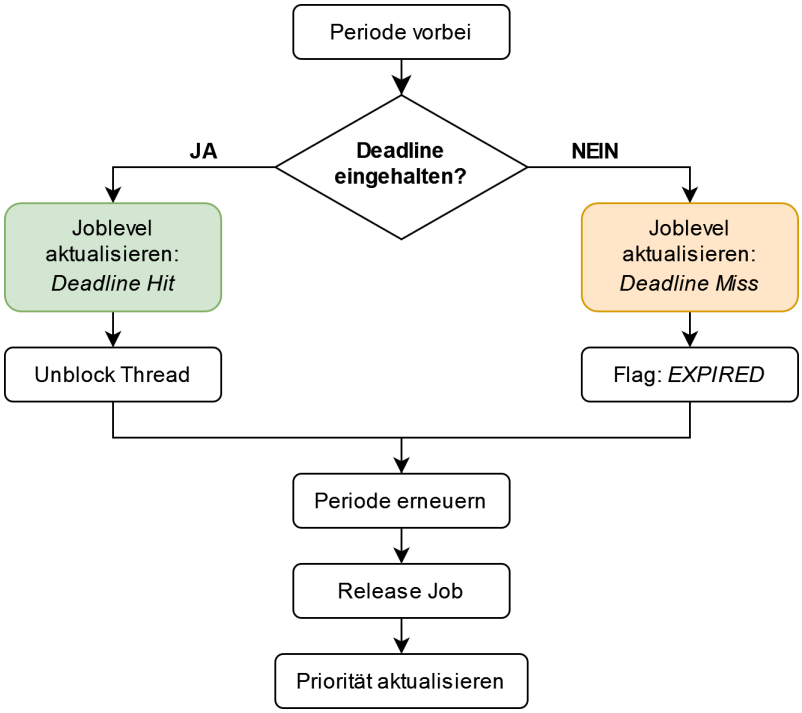


Abbildung 7: Ablaufdiagramm WHA Timeout Routine

Für den WHA Scheduler wurde die Timeout-Routine so angepasst, dass sowohl bei einer eingehaltenen, als auch verpassten Deadline die `Scheduler_Release_job` aufgerufen und eine Prioritätsänderung an den Thread Control Block propagiert. Beim Rate Monotonic Manager wurde dies ausschließlich bei einer eingehaltenen Deadline durchgeführt.

5.4. Konfiguration des Schedulers

Die Konfiguration von RTEMS-Anwendungen erfolgt durch die Verwendung bestimmter Makros, die die Funktionsweise der Anwendung zur Kompilierungszeit definieren. Diese Makros definieren unter anderem den verwendeten Scheduler. Auf diese Weise erlaubt RTEMS auch die Einrichtung eines benutzerdefinierten Schedulers. Dazu ist es notwendig, verschiedene Makros festzulegen, die in diesem Abschnitt erläutert werden. In Listing 8 sind sie spezifischen Konfigurationen für den WHA Scheduler zu sehen. Das Konfigurationsmakro `#define CONFIGURE_SCHEDULER_USER` legt fest, dass RTEMS einen benutzerdefinierten Scheduler verwenden soll. Durch diese Definition wird die Verwendung der anderen von RTEMS implementierten Schedulers ausgeschlossen.

Mit dem Makro `#define CONFIGURE_SCHEDULER` wird eine statische Instanz der Scheduler Datenstruktur *`Scheduler_WHA_Context`* festgelegt. Sie erhält einen spezifischen Namen, wie die stanardmäßigen RTEMS Scheduler auch, um eine eindeutige Identifizierung zu ermöglichen.

Als nächstes müssen die `#define CONFIGURE_SCHEDULER_TABLE_ENTRIES` definiert werden, die im RTEMS Kernel zur Initialisierung der *`Scheduler_Control`* Struktur verwendet werden. Darin werden die Entry Points des WHA-Schedulers festgelegt, die weiter unten erläutert werden. Außerdem wird die maximale interne Priorität festgelegt und dem Scheduler ein RTEMS-Name zur Identifizierung zugewiesen.

Abschließend wird durch `#define CONFIGURE_SCHEDULER_USER_PER_THREAD` die per-thread-Information für den benutzerdefinierten Scheduler definiert

```
#define CONFIGURE_SCHEDULER_USER
#define SCHEDULER_WHA_CONTEXT_NAME( name )
    SCHEDULER_CONTEXT_NAME( WHA_ ## name )

#define CONFIGURE_SCHEDULER
    static Scheduler_WHA_Context SCHEDULER_WHA_CONTEXT_NAME( dflt )

#define CONFIGURE_SCHEDULER_TABLE_ENTRIES {
    &SCHEDULER_WHA_CONTEXT_NAME( dflt ).Base,
    SCHEDULER_WHA_ENTRY_POINTS,
    INT_MAX,
    ( rtems_build_name('W','H','A',' ') )
}

#define CONFIGURE_SCHEDULER_USER_PER_THREAD Scheduler_WHA_Node
```

Listing 8: Konfiguration des WHA Schedulers

Für die Anbindung des WHA Schedulers müssen die relevanten Entry Points verlinkt werden. Diese umfassen die in Abschnitt 5.2.1. beschriebenen Funktionen, die vom RTEMS Kernel zur Initialisierung des Scheduler Kontext bzw. der Scheduler Nodes, sowie zur Durchführung von Scheduling-Operationen aufgerufen werden. Das Makro `#define SCHEDULER_WHA_ENTRY_POINTS`, welches die Entry Points für den WHA Scheduler festgelegt, ist in Listing 9 dargestellt. Dieses wird später verwendet um eine

Strukturinstanz der *Scheduler_Operations*[3] zu initialisieren.

Einige Operationen benötigen keine Scheduler-spezifische Implementierung. RTEMS stellt für diese Fälle Default-Operationen zur Verfügung, die eine grundlegende Funktion enthalten und unabhängig vom gewählten Scheduler funktionieren.

```
#define SCHEDULER_WHA_ENTRY_POINTS \
  { \
    _Scheduler_WHA_Initialize, \
    _Scheduler_WHA_Schedule, \
    _Scheduler_WHA_Yield, \
    _Scheduler_WHA_Block, \
    _Scheduler_WHA_Unblock, \
    _Scheduler_WHA_Update_priority, \
    _Scheduler_default_Map_priority, \
    _Scheduler_default_Unmap_priority, \
    _Scheduler_WHA_Node_initialize, \
    _Scheduler_default_Node_destroy, \
    _Scheduler_WHA_Release_job, \
    _Scheduler_WHA_Cancel_job, \
    _Scheduler_default_Start_idle \
  }
```

Listing 9: Definition der Scheduler Entry Points

[3] Im RTEMS Source-Code: `rtems/score/scheduler.h`

6. Testen und Evaluierung

Der implementierte weakly hard Scheduling Algorithmus muss umfassend getestet und evaluiert werden, um seine fehlerfreie Arbeitsweise sicherstellen zu können. Es ist wichtig, dass bei der Initialisierung der Tasks die Parameter richtig berechnet und festgelegt werden. Anschließend wird die Funktionsweise des Algorithmus überprüft, insbesondere müssen dafür die Funktionen für das Joblevel-Update und die Timeout-ISR genau analysiert werden, um logische Fehler ausschließen zu können.

Ein weiterer zu testender Aspekt ist die Interferenz mehrerer Tasks über den zeitlichen Ablauf hinweg. Die Laufzeit des Scheduling Algorithmus wird gemessen, um zu bestimmen, wie lange er zur Ausführung benötigt. Darüber hinaus wird untersucht, ob Unterschiede zwischen der Simulation für die Real Time Analysis und der tatsächlichen Ausführung durch den RTEMS Kernel auftreten. Falls solche Unterschiede vorhanden sind, wird analysiert, welche Auswirkungen diese haben. Die genauen Testmetriken werden in Abschnitt 6.3. zu jedem Experiment beschrieben.

Zum Testen und Evaluieren wird eine spezielle Testanwendung entwickelt, die erst durch Debugging und dann in verschiedennen Experimenten ausgeführt. Zunächst erfolgt die Ausführung in QEMU, da dies in der Entwicklungsphase einfacher ist. Abschließend wird der Scheduler auf dem ZedBoard getestet, um sicherzustellen, dass er auch auf Hardware funktioniert und um die Laufzeit auf Hardware zu messen. Verschiedene Testszenarien zur Untersuchung werden später detailliert erläutert und die jeweiligen Testziele aufgezeigt.

6.1. Tracing

Die Arbeitsweise des Schedulers und der Ablauf der Testszenarien soll durch Tracing analysiert und nachvollzogen werden. Dafür werden in der Testanwendung Aufrufe der implementierten Scheduler Operationen in Form von Events erfasst und jeweils bestimmte Metadaten aufgezeichnet.

Die gesammelten Trace-Daten sollen anschließend in einer interaktiven Visualisierung ausgewertet werden. Ziel ist es, die korrekte Funktion des Schedulers sicherzustellen, insbesondere im Hinblick auf den korrekten Ablauf der Events und die Aktualisierung

der Prioritäten. Darüber hinaus sollen die gesammelten Daten hinsichtlich der später beschriebenen Metriken ausgewertet werden.

6.1.1. Relevante Events

Um einen detaillierten Überblick über den Ablauf zu erhalten, wäre es theoretisch möglich, eine große Anzahl verschiedener Events aufzuzeichnen. Jede zusätzliche Eventaufzeichnung erhöht jedoch den Tracing-Overhead, was bei einer großen Anzahl von Events die Laufzeit stark verlängern und das Ergebnis verfälschen kann. Daher werden nur die wichtigsten Events aufgezeichnet, um den Overhead gering zu halten und gleichzeitig relevante Daten zur Bewertung der Funktionalität des Schedulers zu gewinnen. Es sollen die folgenden Events erfasst werden:

- Thread Unblock
- Thread Block
- Thread Switch
- Prioritäts Update
- Release Job
- Ein- und Austritt der Timeout Routine
- Deadline Hit und Miss
- Schedulability Fehler

Für die Auswertung der Events ist es wichtig, dass diese immer dem entsprechenden Thread zugeordnet werden können. Insbesondere bei der Ausführung der Timeout-Routine, die keine eigene Thread-ID hat, muss der Bezug zum Thread hergestellt werden.

6.1.2. RTEMS Tracing Möglichkeiten

RTEMS bietet verschiedene Komponenten, um Anwendungen zu tracen:

- **RTEMS Trace Linker** – fügt einer Anwendung Tracing-Funktionen hinzu, indem es beim Linken Tracepunkte in den Programmcode einfügt.
- **RTEMS Capture Engine** – Bietet vordefinierte Tracepoints für Task-Ereignisse mit minimaler Systembelastung und wird zur Laufzeit an RTEMS gebunden. Im Scheduling werden nur Thread-Switches erfasst.
- **RTEMS Event Recording** – Zeichnet verschiedene Events auf, darunter Thread Switches und Interrupt Entry / Exit. Eigene Events können mit `rtems_record_produce` aufgezeichnet werden. Die aufgezeichneten Events werden entweder über eine TCP

Verbindung an einen Host Computer übertragen, oder nach beenden des Tests als Base64 kodierter Dump ausgegeben.

Zum Analysieren der Scheduler Events wurde die Event Recording Komponente[4] verwendet. Damit wurden die Thread Switches und die einzelnen Scheduler Events als benutzerdefinierte Records gespeichert. Der erzeugte Base64 Dump wird mit dem Programm `rtems-record-lttng` in das Common Trace Format (CTF) konvertiert. Dies kann zur Auswertung mit *babeltrace* oder *Eclipse Trace Compass* eingelesen werden.

Eine Auswertung der erzeugten Tracedaten wurde mit dem Eclipse Trace Compass erprobt. Die Thread-Switches sowie die Ausführung der Timeout-Routine können dort korrekt dargestellt und ausgewertet werden. Es hat sich jedoch herausgestellt, dass die Prioritäten der Threads nicht erfasst werden. Entweder aufgrund einer fehlenden Implementierung im RTEMS-Kernel, oder weil der Trace Compass die Daten nicht korrekt zuordnen kann. Da die ausgeführte Timeout-Routine eine eigene Thread-ID hat und nicht direkt mit den anderen Threads verknüpft ist, ist eine korrekte Zuordnung ebenfalls nicht möglich.

6.1.3. Eigene Tracing-Funktion

Die von den RTEMS-Komponenten erzeugten Tracedaten reichen nicht aus, um die Funktionsweise des Schedulers zu beurteilen. Auch die Auswertung im Eclipse Trace Compass ist nicht optimal, um eine gute Scheduling-Analyse zu erhalten. Daher werden eigene Trace-Funktionen implementiert und ein eigenes Python-Skript zur Visualisierung entwickelt. Es ist wichtig, dass diese einen sehr geringen Overhead haben, da die Events sehr häufig auftreten und ein großer Overhead das Tracing-Ergebnis stark verfälschen könnte.

Für eine einfache Extrahierung der binären Tracedaten sollten diese Base64-kodiert und als Text im Terminal ausgegeben werden. Die Verwendung eines Dateisystems in RTEMS ist mit zusätzlichem Aufwand verbunden, daher vereinfacht diese Vorgehensweise den Download von Daten aus QEMU oder dem Zedboard erheblich. Eine einfache Ausgabe der Metadaten mit `printf` zum Zeitpunkt des Events kommt nicht in Frage, da diese Funktion einen hohen Overhead hat und nicht threadsicher ist. In der Timeout-Routine und bei Scheduler-Operationen würde dies zu erheblichen Problemen führen.

[4] https://docs.rtems.org/branches/master/user/tracing/eventrecording.html

Anstatt die Daten bei jedem Ereignis sofort auszugeben, speichert die Tracing-Funktion die Daten während eines Tests im Arbeitsspeicher. Die Metadaten eines Events werden in Frames zusammengefasst und byteweise gespeichert. Nach Abschluss des Tests werden die Base64 kodierteren Daten ausgegeben, ähnlich wie beim oben beschriebenen Event Recording.

Feld	**Event ID**	**Zeit**	**Thread ID**	**Priorität**
Größe	32 Bit	64 Bit	32 Bit	64 Bit

Tabelle 2: Allgemeines Trace Frame eines Events

Für alle Events außer dem Thread Switch Event werden die Daten in einem 24-Byte Frame gespeichert, das aus den vier Feldern *Event ID*, *Zeit*, *Thread ID* und *Priorität* aufgebaut und in Tabelle 2 zu sehen ist. Mit der Funktion

```
void trace_event(uint32_t event, uint32_t thread_id, uint64_t priority)
```

wird ein solches Frame erzeugt. Um die speichereffizient zu arbeiten, werden die Event- und Thread ID jeweils als 32 Bit Integer gespeichert. Die Event ID erfüllt die Funktion der eindeutigen Identifikation des jeweiligen Events, wobei kein direkter Zusammenhang mit dem RTEMS-Kernel besteht. Die Tracing-Funktion ermittelt den Zeitstempel mit der Funktion `rtems_clock_get_uptime`, die die Zeit seit dem Bootvorgang in Nanosekunden liefert. Jede zu analysierende Schedulingoperation ruft die Funktion `trace_event` auf und übergibt ihre EventID sowie die ID und Priorität des betreffenden Threads.

Feld	**Event ID**	**Zeit**	**aktuelle Thread ID**	**aktuelle Priorität**	**nächste Thread ID**	**nächste Priorität**
Größe	32 Bit	64 Bit	32 Bit	64 Bit	32 Bit	64 Bit

Tabelle 3: Trace Frame für ein Thread Switch Event

Für Thread-Switch-Events werden die in Tabelle 3 dargestellten Metadaten erfasst. Im Vergleich zu normalen Trace Events wird zusätzlich zur ID und Priorität des aktuellen Threads auch die ID und Priorität des nachfolgenden Threads gespeichert.

Ein Thread Switch, auch Dispatch genannt, wird nicht vom Scheduler ausgeführt, sondern vom RTEMS Kernel. RTEMS bietet mit sogenannten User Extensions die Möglichkeit, benutzerdefinierte Funktionen in den Kernel einzubinden, die zum Zeitpunkt von Thread-Operationen ausgeführt werden. Durch die Einrichtung der *.thread_switch*

User Extension wird nach einem Thread Switch die definierte Funktion aufgerufen, an die Informationen über die betroffenen Threads übergeben werden:

```
void trace_thread_switch( Thread_Control *executing,
                          Thread_Control *heir );

#define CONFIGURE_INITIAL_EXTENSIONS \
                        { .thread_switch = trace_thread_switch }
```

Die Funktion `trace_thread_switch` setzt die Event ID und den Zeitstempel analog zur `trace_event`. In den beiden Thread Control Blocks `executing` und `heir` sind die IDs und Prioritäten der Threads relevanten enthalten. Zum Schreiben der Daten verwenden die beiden beschriebenen Tracefunktionen einen rekursiven Mutex, um sicherzustellen, dass immer nur ein Thread Daten speichert.

Die abschließende Ausgabe der Tracedaten findet mit der Funktion `void end_trace()` statt. Sie blockiert zunächst die Aufnahme weiterer Tracedaten, um Komplikationen mit der Timeout-Routine oder anderen Scheduler-Operationen zu vermeiden. Danach speichert sie die Endzeit, kodiert die Daten im Base64-Format und gibt sie aus.

6.1.4. Testautomatisierung mit Python

Um die Ausführung der Experimente auf QEMU und dem ZedBoard zu automatisieren, wurde ein Python-Skript entwickelt. Dieses Skript verwendet eine JSON-Datei, in der die auszuführenden Experimente definiert sind. In Listing 10 ist ein Beispiel für die Konfiguration eines Experiments zu sehen. Dabei besteht ein Experiment immer aus einem Satz von Tasks, denen jeweils eine Periodenlänge, eine Worst Case Execution Time (WCET) und die Weakly Hard Parameter m und K zugewiesen werden. Die Periodenlänge und die WCET werden in Millisekunden angegeben. Der angegebene Name wird zur Benennung der erzeugten Dateien verwendet. Mit *end* wird die Testdauer in Sekunden festgelegt.

```
{
  "name": "experiment_bespiel",
  "end": 10.0,
  "tasks": [
    { "period": 100, "wcet": 10, "m": 1, "K": 3 },
    { "period": 200, "wcet": 70, "m": 1, "K": 3 }
  ]
}
```

Listing 10: JSON Konfiguration eines Experiments

Das Skript führt automatisch alle Experimente aus, die in der JSON-Konfiguration angegeben sind. Für jedes Experiment werden anhand der Konfiguration den Tasks, wie in Abschnitt 4.2. erklärt, ihre Prioritäten zugewiesen. Aus den Prioritäten und den anderen Taskparametern generiert das Skript einen C Header (Listing 13), aus der die RTEMS Anwendung den Satz an Tasks entnimmt. Anschließend wird die Anwendung neu kompiliert und auf QEMU oder auf dem ZedBoard gestartet. Dabei werden die Terminalausgaben ausgelesen, als Logdateien gespeichert und die Base64-kodierten Tracedaten extrahiert.

6.1.5. Visualisierung mit Matplotlib

Zur Visualisierung der Testergebnisse wird Python mit der Bibliothek Matplotlib verwendet. Das dafür entwickelte Skript liest die zuvor gesammelten Daten des jeweiligen Experiments ein, dekodiert die Base64-Daten und konvertiert sie eventweise zu den entsprechenden Datenframes. Die Datenframes der Events werden anschließend weiter verarbeitet, um Informationen über die Ausführung der Threads zu gewinnen.

Aus den Events werden Threadzustände synthetisiert, die eine Zeitspanne beschreiben, in der der Thread einen der folgenden Zustände annimmt:

- *RUNNING* – Thread läuft aktuell auf dem Prozessor
- *READY* – Thread ist bereit zur Ausführung
- *BLOCKED* – Thread ist blockiert
- *SCHEDULING* – unterbrochen durch die Timeout Routine

Alle Threadzustände, die während der Laufzeit eines Jobs aufgetreten sind, werden zusammen mit der Information, ob der Job seine Deadline eingehalten hat, in einer Job-Execution zusammengefasst. Zudem werden die Laufzeiten der Timeout-Routinen wird

in Abhängigkeit von der Einhaltung bzw. Nichteinhaltung der Deadline erfasst. Daraus resultieren die später beschriebenen Metriken.

Die aus vom Python Skript erzeugten Abbildungen zeigen, wie in Abbildung 8 zusehen, den zeitlichen Verlauf der Zustände, Events und Prioritäten der ausgeführten Threads. Das obere Zustandsdiagramm stellt die Zustände der Threads als farbige Balken und die Events als Linien oberhalb der Balken dar. Dabei stehen grüne Balken für den Zustand *RUNNING*, blaue für *READY*, rote für *BLOCKED* und graue für *SCHEDULING*. Dabei ist anzumerken, dass sich zur selben Zeit immer höchstens ein Thread im *RUNNING* Zustand befindet. Ein Zustand *SCHEDULING* unterbricht immer den aktuell laufenden Thread. Da diese Unterbrechungen sehr kurz sind, sind diese in den meisten Fällen nur als sehr feine Unterbrechung des laufenden Threads zu erkennen - oder aber an den zur selben Zeit auftretenden Events *DL_MISS* bzw. *DL_HIT*.

6.2. Testanwendung

Zur Evaluation der Experimente wurde eine Testanwendung entwickelt, die die definierten Tasks für die Dauer des Experiments ausführt und die gesammelten Tracingdaten ausgibt. Die Tasks haben einzig die Aufgabe, periodische weakly hard Jobs auszuführen, deren Laufzeit jeweils genau der WCET entspricht. Sie müssen also den Prozessor während ihrer Laufzeit durch ein Busy Wait belegen.

In anderen RTEMS Anwendungen wird häufig die Funktion `rtems_wake_task_after` verwendet, um einen Task für eine bestimmte Zeit warten zu lassen. Durch diese Funktion blockiet ein Task sich selbst bis zum gegebenen Zeitpunkt. Da sie den Prozessor in der Zwischenzeit nicht verwendet, ist sie für diese Aufgabe nicht geeignet.

Eine alternative Möglichkeit, eine Busy-Wait-Funktion zu implementieren, ist als Pseudocode in Listing 11 dargestellt. Hierbei wird die aktuelle Zeit als Startzeitpunkt festgelegt, und die Endzeit wird durch Addition der Worst-Case Execution Time bestimmt. Die while-Schleife wird dann solange ausgeführt, bis die Endzeit erreicht ist.

```
start = now()
end   = start + wcet
while (now() < end) {
  // nichts machen
}
```

Listing 11: Pseudocode Busy Wait Funktion

Diese Funktion erfüllt ihren Zweck, wenn der Thread die gesamte Zeit über auf dem Prozessor läuft. Allerdings arbeitet der Scheduler präemptiv, sodass ein unterbrochener Thread mit dieser Funktion lediglich die verstrichene Zeit seit ihrem Start erfasst, nicht aber die tatsächlich genutzte Prozessorzeit. Die beschriebene Busy Wait Funktion muss dementsprechend so angepasst werden, dass der Thread auch bei Unterbrechungen den Prozessor für die erforderliche Zeit verwendet.

Dafür bietet sich die RTEMS Kernel Funktion `_Thread_Get_CPU_time_used` an, mit der die verwendete Prozessorzeit eines Threads seit seiner Erstellung abgefragt werden kann. Die Prozessorzeit ist im Thread Control Block gespeichert und wird vom Kernel aktualisiert, wenn der Thread den Prozessor übernimmt oder ihn verlässt.

```
void busy_wait(uint32_t milliseconds)
{
  Thread_Control *the_thread;
  Timestamp_Control cpu_time_begin, cpu_time_end;

  the_thread = _Thread_Get_executing();
  cpu_time_begin = _Thread_Get_CPU_time_used(the_thread);
  cpu_time_end = Add_time_interval(cpu_time_begin, milliseconds);

  while (_Thread_Get_CPU_time_used(the_thread) < cpu_time_end) {
    // nichts machen ...
    // (Abbruch bei verpasster Deadline oder Ende des Experiments)
  }
}
```

Listing 12: Busy Wait Funktion für verwendete Prozessorzeit

Eine anpegasste Busy Wait Funktion, die die `_Thread_Get_CPU_time_used` als Zeitgeber verwendet und für die Testanwendungen verwendet wird, ist in Listing 12 dargestellt. Da es sich um die Zeit seit der Erstellung des Threads handelt, muss

eine Startreferenz gesetzt und die Endzeit durch Addition der zu nutzenden Zeit in Millisekunden berechnet werden. Die while-Schleife wird solange ausgeführt, bis der Thread den Prozessor für die entsprechende Zeit benutzt hat. Zwei Sonderfälle für einen Abbruch, die in Listing 12 nicht ausgeführt werden, sind das Verpassen der aktuellen Job-Deadline oder das Ende des Experiments.

Wie zuvor erklärt, wird für jedes Experiment immer ein Satz von Tasks ausgeführt. Dabei besitzt jeder Task mehrere Parameter und Prioritäten, die in einer *task_config* Struktur gespeichert werden (Listing 13). Die Konfigurationen dieser Tasks, einschließlich ihrer Parameter und Prioritäten, sind in dem konstanten Array `task_set` zusammengefasst. Dieses Array wird als *task_config* initialisiert und global gespeichert. Das Python-Skript, das die Experimente ausführt und den C Header generiert, überschreibt ausschließlich das `task_set` Array und die Variable `num_tasks`, welche die Anzahl der Tasks festlegt.

```c
typedef struct {
  uint32_t task_index;
  uint32_t period;
  uint32_t wcet;
  uint32_t m;
  uint32_t K;
  uint64_t *priorities;
} task_config;

const int num_tasks = 2;
const task_config task_set[] = {
  { 0, 100, 10, 1, 3, (uint64_t[]){1, 3, 5} },
  { 1, 200, 70, 1, 3, (uint64_t[]){2, 4, 6} }
}
```

Listing 13: Task Konfiguration

Die Testanwendung erzeugt im Init-Thread die in der Konfiguration definierten Tasks, wobei jeder Task jeweils seine höchste Priorität als Startpriorität und einen Pointer auf seine task_config als Argument erhält. Nach der Erzeugung werden alle Tasks gestartet. Jeder Task initialisiert seine Weakly Hard (WH) Parameter mit Hilfe des WHA Job Managers und startet anschließend den periodischen Job, der die Busy Wait Funktion ausführt. Der Init-Thread wartet, bis alle Tasks beendet sind, was durch die Verwendung

einer RTEMS-Barriere erreicht wird. Am Ende ruft die Testanwendung die Funktion `end_trace` zur Ausgabe der Tracedaten auf und beendet sich selbst.

6.3. Experimente

Während der Entwicklungsphase wurden einfache Testanwendungen in QEMU gestartet und mit GDB debugged, um den Programmablauf des Kernels und die Funktionsweise der Scheduler-Operationen zu verstehen. Diese Testanwendungen wurden auch zur Fehlersuche und -behebung verwendet, wobei keine Datenerfassen mittels Tracing stattfand. Zur Entwicklung des Schedulers waren diese Tests unerlässlich, sie werden aber in diesem Abschnitt nicht beschrieben, da sie keinen großen analytischen Mehrwert für die Evaluation bieten. Dabei musste die Anwendung ohne Compileroptimierungen und mit Debuginformationen kompiliert werden. Dies würde auch zu längeren Ausführungszeiten und fehlerhaften Messungen bei der Tracing-Analyse führen.

Zur Evaluierung des implementierten WHA Schedulerts wurden verschiedene weitere Tests bzw. Experimente durchgeführt. Dafür wurden mit den in Abschnitt 6.1.3. beschriebenen Funktionen Tracepunkte in den Scheduler Operationen eingefügt, um den Ablauf visualisieren und analysieren zu können. Die Testanwendung wurde mit dem Optimierungslevel 2 kompiliert, das standardmäßig für RTEMS-Anwendungen verwendet wird.

Im Folgenden werden die durchgeführten Experimente beschrieben. Dabei zielt jedes Experiment auf die Untersuchung von bestimmten Verhaltensweisen oder verschiedenen Metriken ab, die jeweils in der Experimentbeschreibung genannt werden. Anschließend werden in Abschnitt 6.4. die Ergebnisse der einzelnen Experimente beschrieben und bewertet.

6.3.1. Verpassen und Einhalten von Deadlines

Das erste Experiment zielt im Allgemeinen darauf ab, die Funktionsweise des Schedulers zu analysieren. In diesem Zusammenhang wird untersucht, ob die Tasks gemäß ihrer Prioritäten korrekt zur Ausführung ausgewählt werden. Wichtig ist auch, ob die Aktualisierung der Prioritäten nach einer verpassten oder erreichten Deadline korrekt funktioniert, wofür eine detaillierte Betrachtung der Timeout-Routine durchgeführt wird.

Zur Durchführung des Experiments wurde das in Tabelle 4 enthaltene Task-Set erstellt, welches zwei High-Tolerance-Tasks enthält. Die konfigurierten WECTs und Perioden resultieren in einer Gesamtauslastung von 108,3 %, was in jedem Fall zu einem Verpassen von Deadlines führen sollte.

Task	WCET [ms]	Periode [ms]	m	K
τ_0	70	200	1	2
τ_1	220	300		

Tabelle 4: Konfiguration Experiment Deadlines

6.3.2. Vergleich mit Simulation

In vier Experimenten werden der implementierten WHA Scheduler mit der Python Simulation verglichen. Tabelle 5 zeigt die entsprechenden Konfigurationen der Experimente, die jeweils aus zwei Tasks bestehen. Der Task τ_0 hat in allen Experimenten eine Auslastung von 50%, während für Task τ_1 unterschiedliche WCETs und Periodenlängen definiert sind. Daraus ergeben sich in den ersten drei Experimenten Gesamtauslastungen von 75%, 100% und 125%. In den ersten drei Experimenten werden die Tasks als Low-Tolerance-Tasks mit einem m/K-Verhältnis von 1/3 ausgeführt. Im letzten Experiment wird der gleiche Tasksatz wie im dritten Experiment verwendet, allerdings als High-Tolerance-Tasks mit einem m/K-Verhältnis von 4/5.

Ziel dieser Experimente ist es, mögliche Unterschiede zwischen dem implementierten Scheduler und der Simulation zu identifizieren und die Schedulability (Planbarkeit) zu untersuchen. Eine Konfiguration von Tasks ist nur schedulable, wenn keiner der Tasks in seiner höchsten Priorität eine Deadline verpasst.

Experiment	Task	WCET [ms]	Periode [ms]	m	K	Last
1	τ_0	50	100	1	3	75%
	τ_1	50	200			
2	τ_0	50	100	1	3	100%
	τ_1	100	200			
3	τ_0	50	100	1	3	125%
	τ_1	150	200			
4	τ_0	50	100	4	5	125%
	τ_1	150	200			

Tabelle 5: Konfigurationen für den Vergleich mit der Simulation

6.3.3. Ausführung vieler Tasks

Während die letzten Experimente sich auf unterschiedliche Lasten und die generelle Funktionsweise des Schedulers konzentrieren, wird in den folgenden Experimenten das Verhalten des Schedulers bei der Ausführung von vielen Tasks untersucht.

Für die Experimente werden die beiden in Tabelle 6 abgebildeten Tasksets verwendet, die aus jeweils 5 Tasks bestehen und eine Gesamtlast von 80% bzw. 56%. Jedes Taskset wird einmal mit Low-Tolerance-Tasks und einmal mit High-Tolerance-Tasks ausgeführt (siehe Tabelle 7).

In diesen Experimenten mit vielen Tasks wird die Arbeitsweise des Schedulers im zeitlichen Verlauf analysiert und das Verhalten bei unterschiedlichen Toleranzen verglichen. Die Experimente werden sowohl auf dem ZedBoard als auch in QEMU durchgeführt, wobei die Periodenlängen, sowie die Laufzeiten der Jobs und der Timeout-Routine ausgewertet und verglichen werden. Für die QEMU Experimente wird ein Ubuntu Host mit einem Intel Xeon W-2295 Prozessor (3.0 GHz) und 256 GB RAM verwendet.

	Task	WCET [ms]	Periode [ms]	Last	Gesamt
Set 1	τ_0	60	200	30 %	80%
	τ_1	30	150	20 %	
	τ_2	150	1000	15 %	
	τ_3	40	400	10 %	
	τ_4	10	200	5 %	
Set 2	τ_0	50	200	25 %	56%
	τ_1	50	1000	15 %	
	τ_2	50	400	10 %	
	τ_3	10	200	5 %	
	τ_4	20	2000	1 %	

Tabelle 6: Task Set Konfigurationen mit unterschiedlicher Last

Experiment	Set	m	K
5	1	1	3
6		2	3
7	2	1	3
8		2	3

Tabelle 7: Parameter m/K für die Experimente 5-8

6.4. Ergebnisse

6.4.1. Verpassen und Einhalten von Deadlines

Für dieses Experiments wurden die Perioden der Threads synchron gestartet. Dies wurde erreicht, indem beide Threads nach der Initialisierung zunächst ihre Periode mit der Funktion `rate_monotonic_period` aktivieren, bevor die Jobs ausgeführt werden.

Die Ausführung der periodischen Jobs ist in den folgenden Abbildungen dargestellt. Im oberen Diagramm sind jeweils die Zustände der Threads über die Zeit dargestellt. Grüne Balken zeigen an, dass der entsprechende Thread ausgeführt wird. Rote Balken bedeuten, dass ein Thread blockiert ist, da er seinen aktuellen Job bereits beendet hat und auf die nächste Periode wartet. Blaue Balken bedeuten, dass der Thread zur Ausführung bereit ist, aber aufgrund einer höheren Priorität eines anderen Threads warten muss.

Die vertikalen dünnen gestrichelten Linien zeigen den Beginn einer Periode. Das untere Diagramm zeigt den zeitlichen Verlauf der Prioritäten.

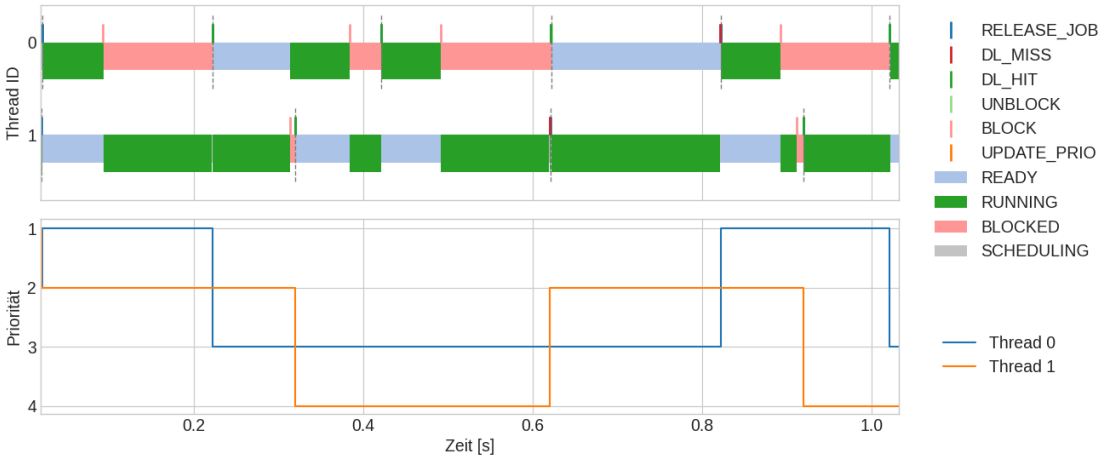


Abbildung 8: Experiment zu Deadlines

Abbildung 8 zeigt den zeitlichen Verlauf des Experiments bis etwa zur ersten Sekunde. Im Diagramm sieht man, dass stets der Thread mit der höchsten Priorität ausgeführt wird, sofern er sich nicht im Zustand *BLOCKED* befindet. Die zwei Tasks haben jeweils mehrere Jobs ausgeführt, wobei nicht alle Deadlines eingehalten wurden.

Eine Aktualisierung der Priorität eines Tasks muss immer am Ende einer Periode durch die Timeout Routine stattfinden. Die Ausführung der Timeout-Routine ist im Diagramm nur als sehr feine Unterbrechung im *READY* Zustand des zu der Zeit ausgeführten Threads erkennbar. Dies liegt daran, dass die Routine nur eine sehr kurze Ausführungszeit benötigt.

In der Konfiguration wurden für beide Tasks die Parameter $m = 1$, $K = 2$ festgelegt, weshalb für beide Tasks $w = 1$ und $h = 1$ gilt. Das bedeutet, dass nach jeder erreichten oder nicht erreichten Deadline eine Anpassung der Priorität erfolgen muss. Dieses Verhalten kann im Verlauf der Prioritäten verifiziert werden: Die ersten Jobs der zwei Tasks haben ihre Deadline eingehalten, weshalb ihre Prioritäten gesenkt wurden. Bei ca. 0,6 Sekunden hat der zweite Job von Thread 1 die Deadline verpasst, wodurch das Joblevel zurückgesetzt und damit die Priorität erhöht wurde. Das gleiche Verhalten ist bei ca. 0,8 Sekunden zu beobachten, wenn Thread 0 seine Deadline verpasst.

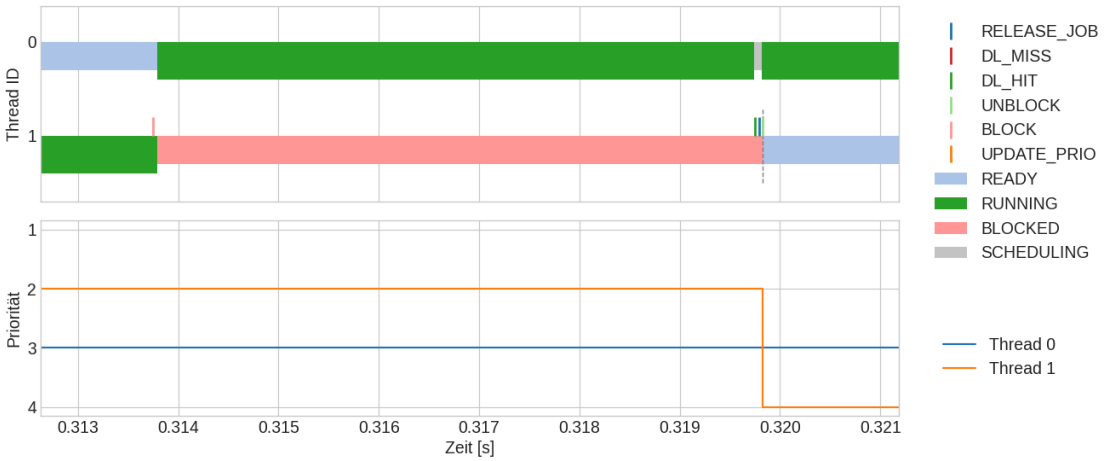


Abbildung 9: Deadline Hit

In Abbildung 9 ist zu sehen, wie der erste Job von Thread 1 seine Deadline einhält und die Priorität angepasst wird. Der Job ist bei 0,314 Sekunden beendet, daher hat sich der Thread selbst blockiert und Thread 0 wird im Anschluss ausgeführt. Bei 0,32 Sekunden endet die Periode von Thread 1, wodurch die Timeout-Routine ausgelöst und die Priorität aktualisiert wird.

Dazu unterbricht sie die Ausführung von Thread 0, weshalb dieser im Diagramm vorübergehend vom Zustand *RUNNING* in den Zustand *SCHEDULING* wechselt. Dennoch ist die ausgeführte Timeout-Routine mit Thread 1 assoziiert, da auf dessen Achse die Ereignisse *DL_HIT*, *RELEASE_JOB*, *UPDATE_PRIO* und *UNBLOCK* dargestellt sind. Nach Beendigung der Timeout-Routine wird Thread 1 in den Zustand *READY* versetzt. Die Ausführung von Thread 0 wird fortgesetzt, da dieser dann die höchste Priorität hat.

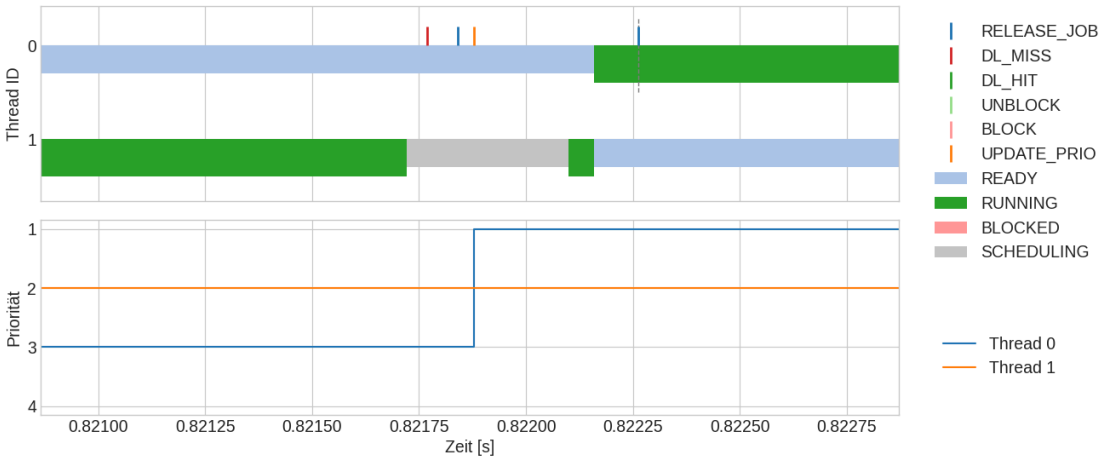


Abbildung 10: Deadline Miss

Das Verhalten beim Verpassen einer Deadline wurde ebenfalls untersucht. In Abbildung 10 ist die Visualisierung der Tracing-Daten mit Fokus auf der verpassten Deadline von Thread 0 dargestellt, die bei etwa 0,82 Sekunden liegt.

Wie schon zuvor beobachtet, unterbricht die Timeout Routine den laufenden Thread, in diesem Fall ist das Thread 1. Bezogen auf den Thread 0 wurden die Events *DL_MISS*, *RELEASE_JOB* und *UPDATE_PRIO* aufgezeichnet. Daran ist erkennbar, dass der Thread 0 seine Deadline verpasst hat und daraufhin die Priorität aktualisiert wurde. Da der Thread sich bereits im Zustand *READY* befindet, muss keine Unblock Operation stattfinden.

Nach Verlassen der Timeout-Routine greift der RTEMS-Kernel sofort ein und veranlasst einen Thread-Switch. Sobald Thread 0 wieder den Prozessor erhält, läuft der alte Job weiter. An der gesetzten *EXPIRED* Flag erkennt er, dass er die Deadline verpasst hat. Thread 0 beendet deshalb seinen aktuellen Jobund startet einen neuen, was am zweiten *RELEASE_JOB* Events erkennbar ist.

6.4.2. Vergleich mit Simulation

Jedes der in Tabelle 5 aufgeführten Experimente wurde sowohl auf dem ZedBoard als auch in einer Python-Simulation durchgeführt. Die Testanwendung für das ZedBoard, die in diesen Experimenten verwendet wird, startet die Perioden der Threads nicht synchron. Das bedeutet, dass jeder Thread seine Periode erst initialisiert, wenn er zum ersten Mal ausgeführt wird.

Die aus diesen Experimenten resultierenden Diagramme der Prioritätsverläufe werden in diesem Abschnitt analysiert. Wie in den vorherigen Diagrammen ist die Zeit auf der x-Achse aufgetragen. In der Simulation wird diese in Ticks gemessen, wobei ein Tick 10 ms entspricht. Aus den Beobachtungen der Experimente werden entsprechende Schlüsse gezogen.

Experiment 1

Für dieses Experiment ergeben sich aus den Weakly Hard Parametern $m = 1$ und $K = 3$ die Werte $h = 2$ und $w = 1$. Dies bedeutet, dass die Priorität eines Threads erst nach dem Erreichen von zwei Deadlines schrittweise verringert wird. Sobald jedoch eine Deadline verpasst wird, wird das Joblevel zurückgesetzt, was dazu führt, dass der Thread die höchste Priorität annimmt. Dieses Verhalten ist auch in den Diagrammen sichtbar.

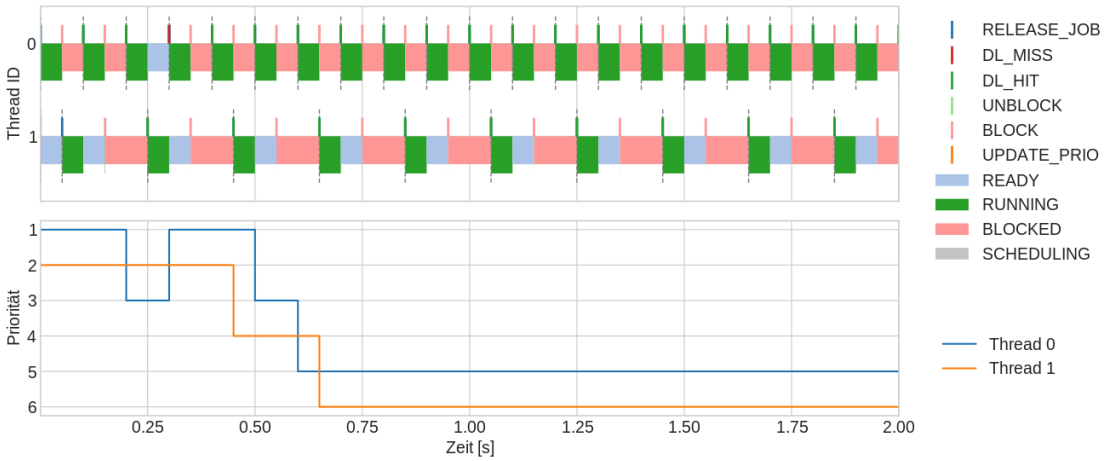


Abbildung 11: Experiment 1 - ZedBoard

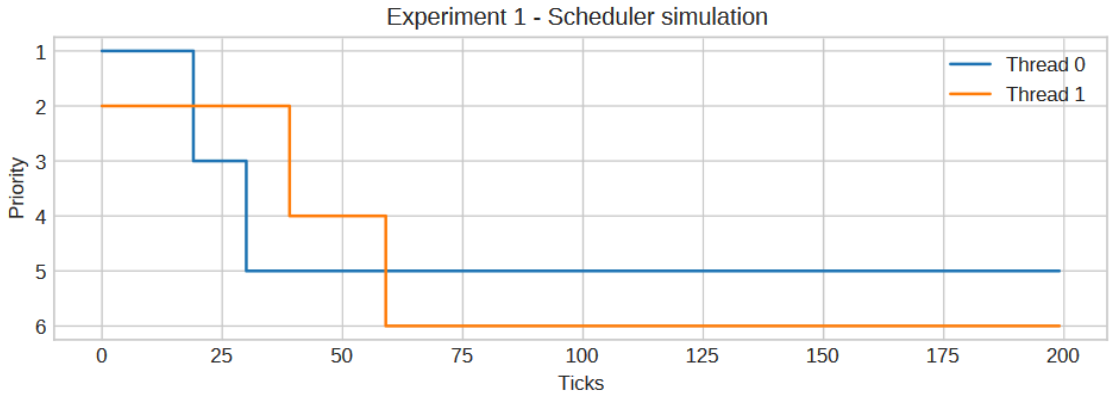


Abbildung 12: Experiment 1 - Simulation

Die Prioritätsverläufe des WHA-Schedulers (Abbildung 11) und der Simulation (Abbildung 12) zeigen im Allgemeinen eine große Ähnlichkeit. In beiden Diagrammen ist zu erkennen, dass sich das System aufgrund der geringen Auslastung nach einiger Zeit entspannt und beide Threads ihre niedrigste Priorität annehmen. Bei der Ausführung auf dem ZedBoard hat der Thread 0, anders als in der Simulation, eine Deadline verpasst.

Ein weiterer erkennbarer Unterschied besteht darin, dass in der Testanwendung die Perioden erst gestartet werden, wenn der jeweilige Thread zum ersten Mal ausgeführt wird. Thread 0 beginnt mit der höchsten Priorität, startet also seine Periode als erster und gibt den Prozessor erst frei, wenn der erste Job beendet ist. Da die Periode von Thread 1 erst nach der Blockierung von Thread 0 gestartet wird, ergibt sich eine Verschiebung von 50 ms im Prioritätsverlauf von Thread 1.

Experiment 2

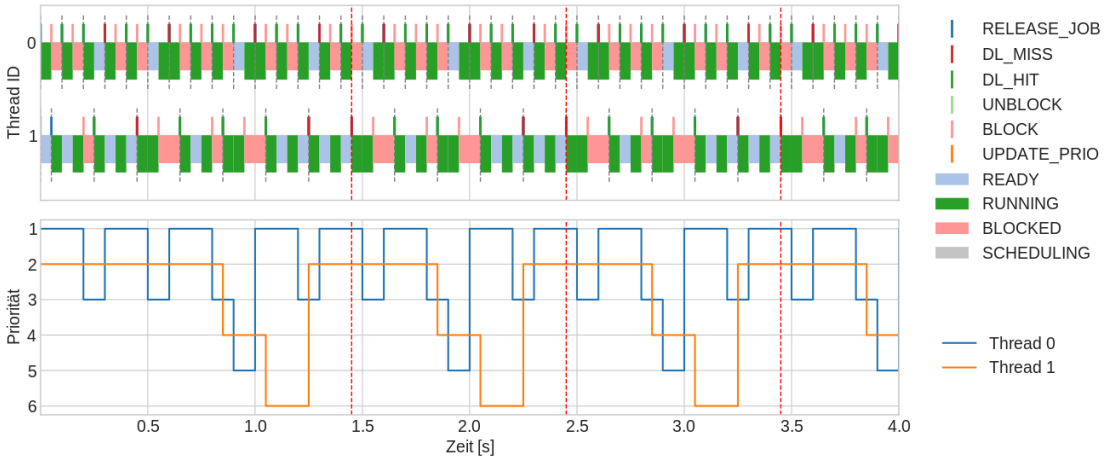


Abbildung 13: Experiment 2 - ZedBoard

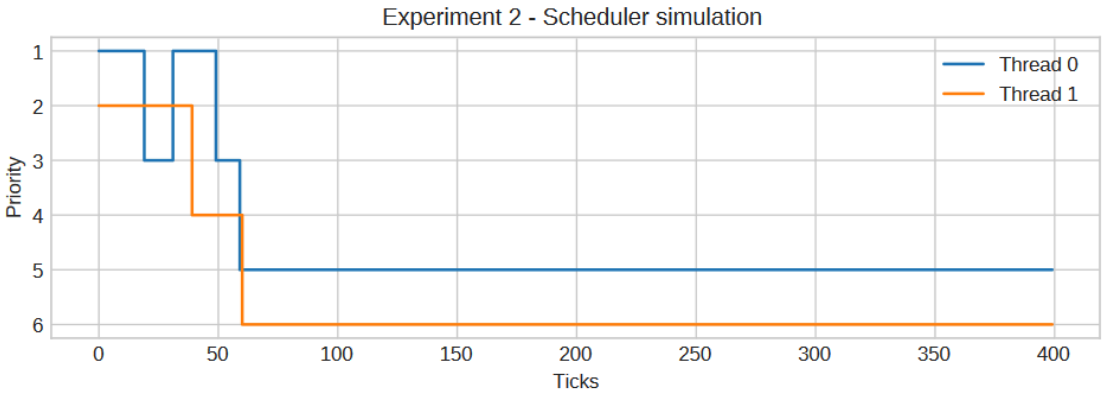


Abbildung 14: Experiment 2 - Simulation

Bei der Durchführung des zweiten Experiments sind große Unterschiede aufgetreten. In der Simulation entspannt sich das System nach kurzer Zeit, ähnlich wie im ersten Experiment. Es wäre anzunehmen, dass das hier verwendete Taskset mit einer Gesamtauslastung von 100% ausführbar ist.

Auf dem ZedBoard hingegen wurden viele Deadlines nicht eingehalten und beide Threads kehren immer wieder in ihre höchste Priorität zurück. Zudem treten mehrfach Fehler beim Scheduling auf, weil der Thread 1 Deadlines verpasst hat, während er bereits in seiner höchsten Priorität war. Die Zeitpunkte dieser Fehler sind in Abbildung 13 durch vertikale rote Linien markiert. Dementsprechend ist dieses Taskset nicht schedulable, obwohl dies der Simulation nach anzunehmen wäre.

Experiment 3

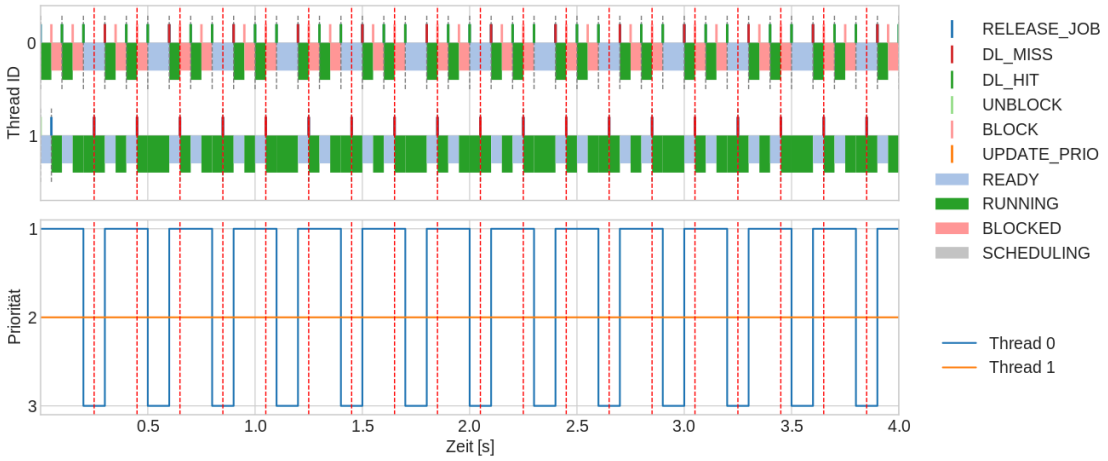


Abbildung 15: Experiment 3 - ZedBoard

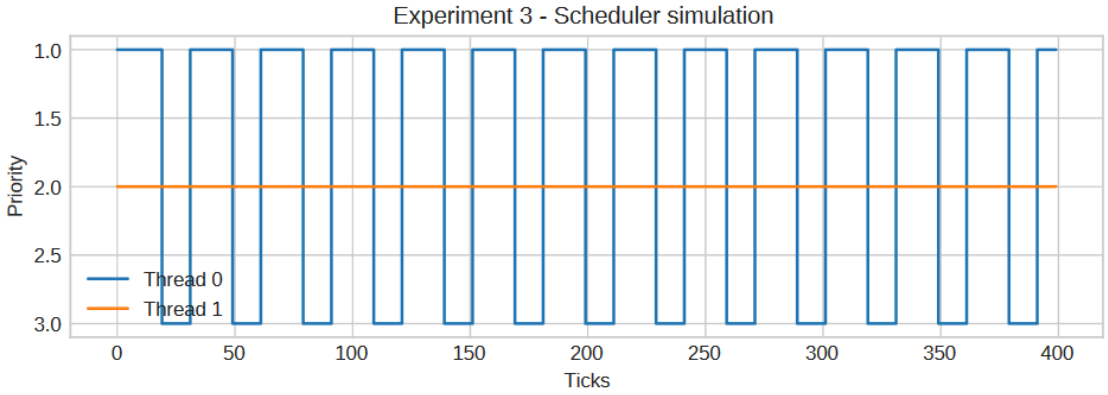


Abbildung 16: Experiment 3 - Simulation

Test und Simulation dieses Experiments mit 125 % Gesamtauslastung zeigen sehr ähnliche Ergebnisse. Thread 0 verpasst gelegentlich eine Deadline, erreicht aber auch einige, weshalb seine Priorität kontinuierlich wechselt. Thread 1 hingegen verpasst alle Deadlines und versucht deshalb ständig, das Joblevel zurückzusetzen.

Laut Simulation ist das Task-Set in diesem Experiment also nicht schedulable, was sich auch bei der Ausführung auf dem ZedBoard zeigt.

Experiment 4

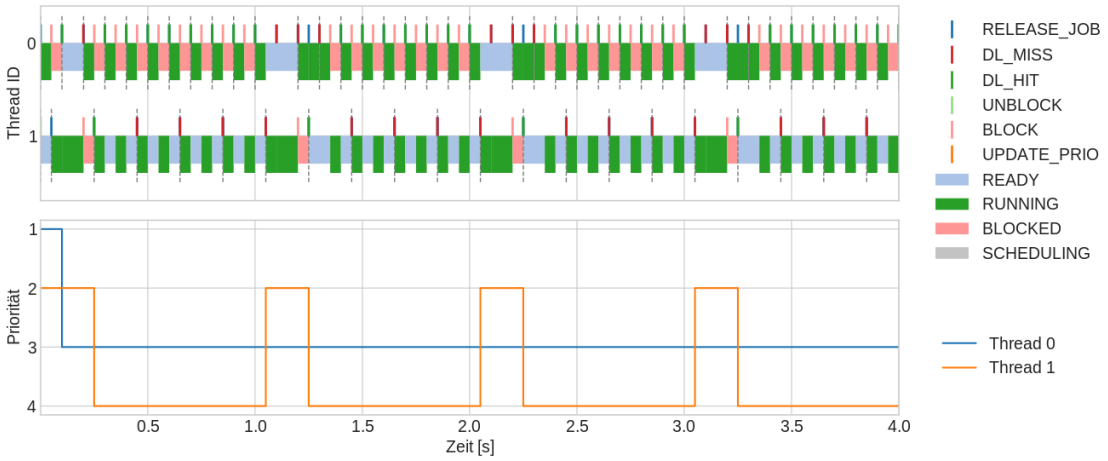


Abbildung 17: Experiment 4 - ZedBoard

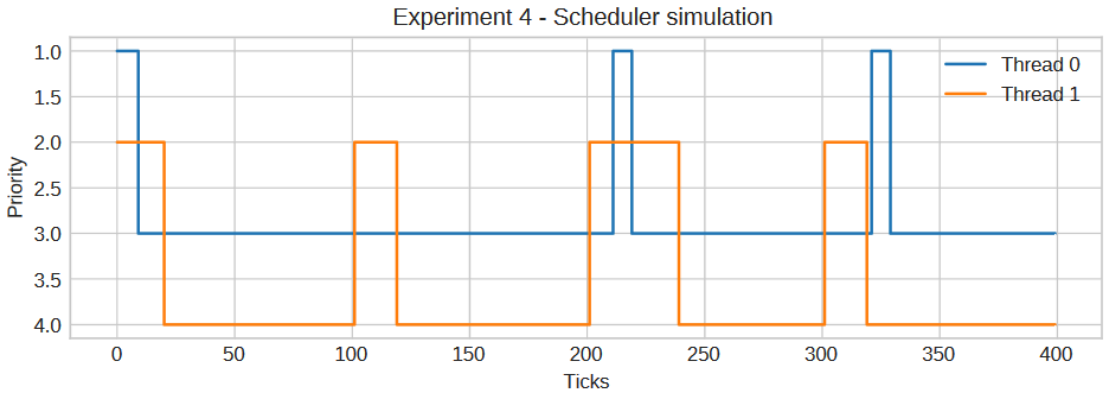


Abbildung 18: Experiment 4 - Simulation

In diesem Experiment wurde ein Taskset mit High-Tolerance-Tasks verwendet, das die gleiche Auslastung hat wie im vorherigen Experiment. Beide Tasks haben die Parameter $m = 4$ und $K = 5$, woraus sich $h = 1$ und $w = 4$ ergeben. Aufgrund der hohen Auslastung treten viele verpasste Termine auf. Thread 1 verpasst immer vier Deadlines hintereinander, danach erhöht er kurz seine Priorität. Dieses Verhalten ist - mit kleinen Abweichungen - sowohl im Test als auch in der Simulation zu beobachten. Trotz einer Auslastung von 125% ist dieses Taskset durch den Einsatz von High-Tolerance-Tasks schedulable.

6.4.3. Ausführung vieler Tasks

Dieser Abschnitt behandelt die Auswertung der in Tabelle 7 aufgeführten Experimente, die jeweils in QEMU als auch auf dem ZedBoard durchgeführt wurden. Dazu werden insbesondere die Zustands- und Prioritätsdiagramme analysiert.

Das in Abschnitt 6.1.5. beschriebene Python Skript zur Auswertung der aufgezeichneten Tracedaten ermittelt für jeden Thread Statistiken zur Periodenlänge, Joblaufzeit und Laufzeit der Timeout-Routine. Bei den aufgezeichneten Metriken konnten Unterschiede zwischen QEMU und ZedBoard festgestellt werden, auf die in diesem Abschnitt eingegangen wird. Da die Unterschiede zwischen QEMU und ZedBoard bei allen Experimenten recht ähnlich sind, beschränkt sich der Vergleich auf die Daten von Experiment 5.

Experiment 5

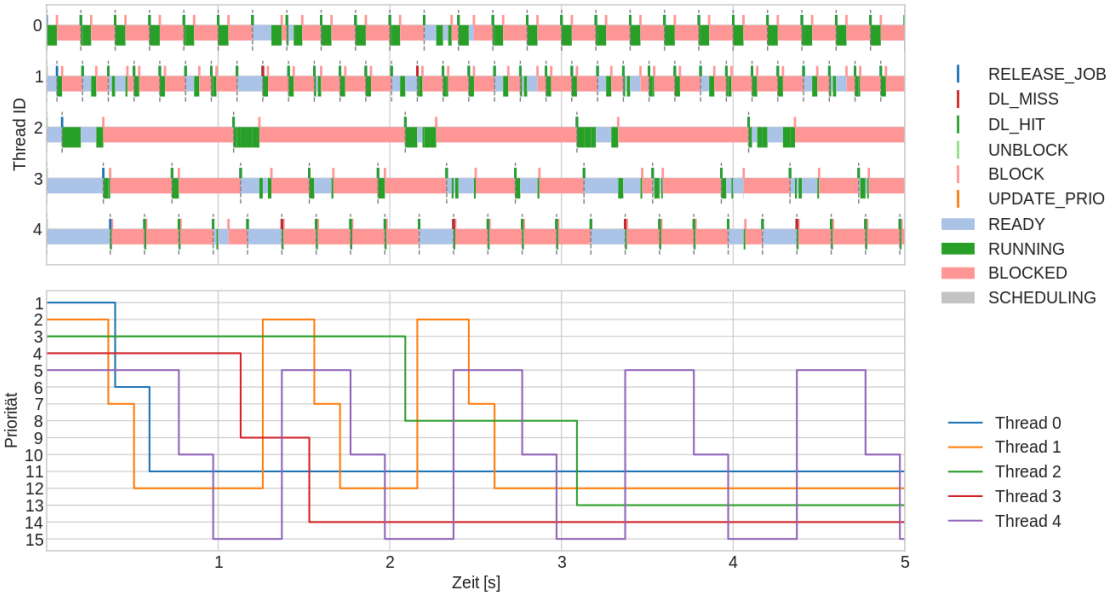


Abbildung 19: Experiment 5 - ZedBoard

In Experiment 5 wurden fünf Low-Tolerance-Tasks mit einer Gesamtlast von 80% ausgeführt. Der Verlauf der Thread-Zustände und -Prioritäten ist in Abbildung 19 dargestellt. Bei der hohen Anzahl an Threads fällt auf, dass die Jobs relativ kurz sind und die Threads die meiste Zeit im blockierten Zustand auf die nächste Periode warten. Es ist zu erkennen, dass nach ca. 3 Sekunden alle Threads bis auf Thread 4 ihre niedrigste Priorität erreicht haben. Thread 4 verpasst in regelmäßigen Abständen einige Deadlines und wechselt dadurch immer wieder in seine höchste Priorität.

Vergleich zwischen ZedBoard und QEMU

Wie bereits in der Einleitung zu diesem Abschnitt erwähnt, wurden die Experimente 5-8 auf dem ZedBoard und in QEMU durchgeführt. Dabei konnten nur marginale Unterschiede in den Zustands- und Prioritätsdiagrammen festgestellt werden, wie der Vergleich der Abbildungen 19 und 20 zeigt. Die zweite verpasste Deadline von Thread 1 tritt in der QEMU-Ausführung eine Periode später auf. Darüber hinaus gibt es kleine zeitliche Unterschiede.

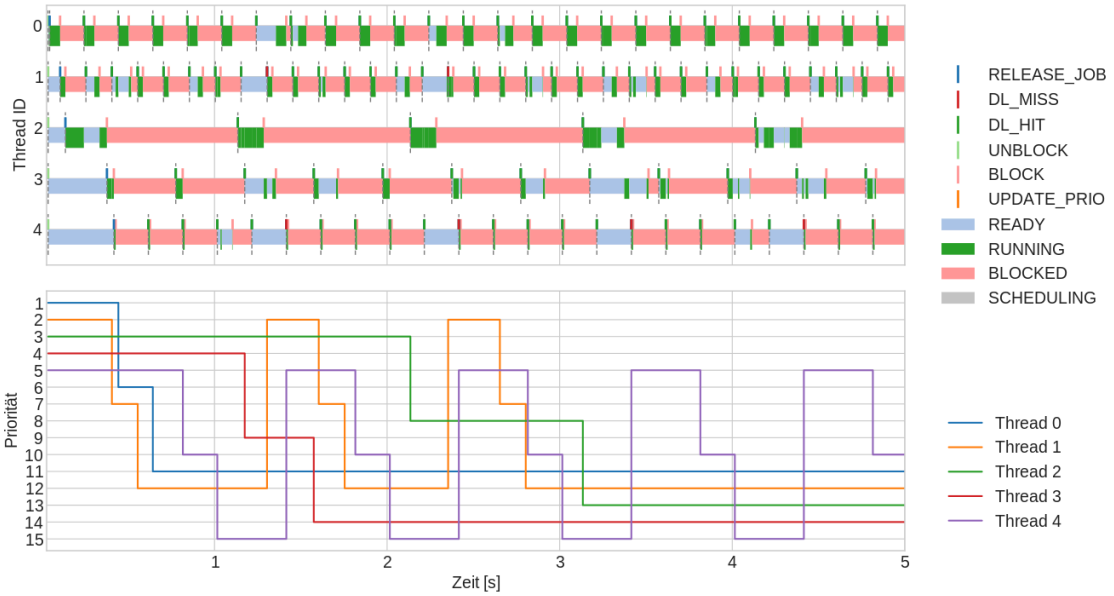


Abbildung 20: Experiment 5 - QEMU

In allen Experimenten nehmen die Zustandsdiagramme nach einiger Zeit die gleiche Verlaufsform an, unabhängig davon, ob sie in QEMU oder auf dem ZedBoard ausgeführt werden. Daher werden nur die Diagramme der ZedBoard-Ausführungen bei der Auswertung der folgenden Experimente betrachtet.

Trotz der nahezu identischen Diagramme gibt es kleine zeitliche Unterschiede zwischen QEMU und ZedBoard. Dazu werden die in Experiment 5 aufgezeichneten Metriken Periodenlänge, Joblaufzeit und Laufzeit der Timeout-Routine in den folgenden Abbildungen analysiert.

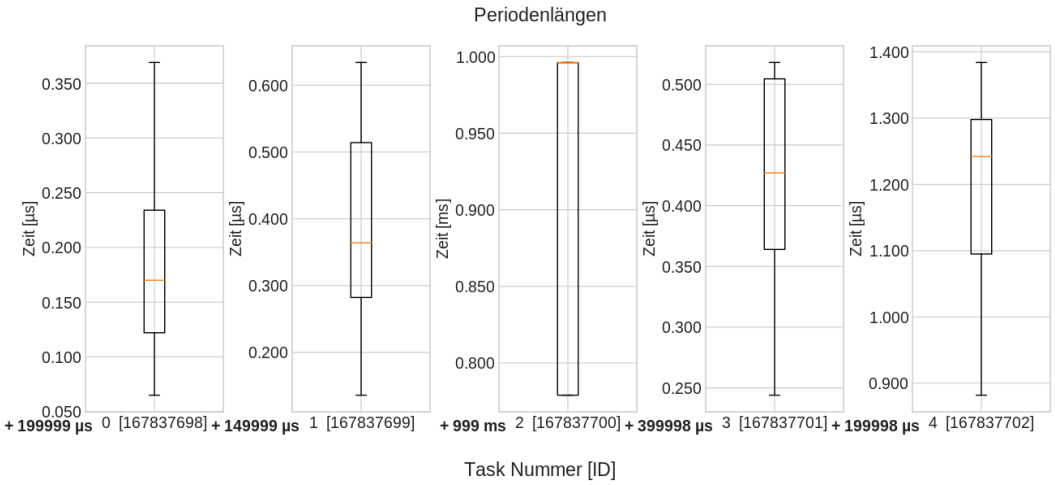


Abbildung 21: Periodenlängen der Tasks - ZedBoard (Experiment)

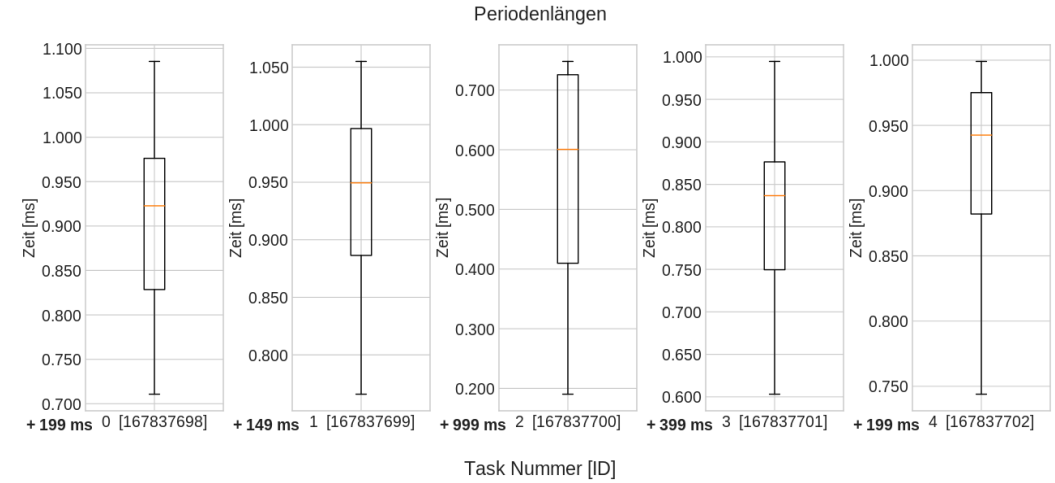


Abbildung 22: Periodenlängen der Tasks - QEMU (Experiment 5)

Zur Analyse der Periodenlängen wurden die aus den Tracing-Daten extrahierten Werte taskweise als Boxplots dargestellt. Die Zahlenwerte liegen im Millisekunden- bzw. Mikrosekundenbereich und würden aufgrund ihrer großen Länge auf der y-Achse unübersichtlich werden. Daher wurde für jeden Boxplot ein Referenzwert berechnet und nur die Abweichung dazu auf der y-Achse dargestellt. Der Referenzwert ist als fettgedruckter Wert links unter dem jeweiligen Diagramm angegeben.

Der Vergleich der Abbildungen 21 und 22 zeigt, dass auf beiden Plattformen alle Tasks mit ihrer jeweiligen Periodenlänge ausgeführt wurden. Es treten geringe Abweichungen auf, die beim ZedBoard ca. ± 100 ns und bei QEMU ca. ± 100 µs betragen. Die Erneuerung der Perioden ist auf dem ZedBoard sind dementsprechend deutlich präziser und konstanter.

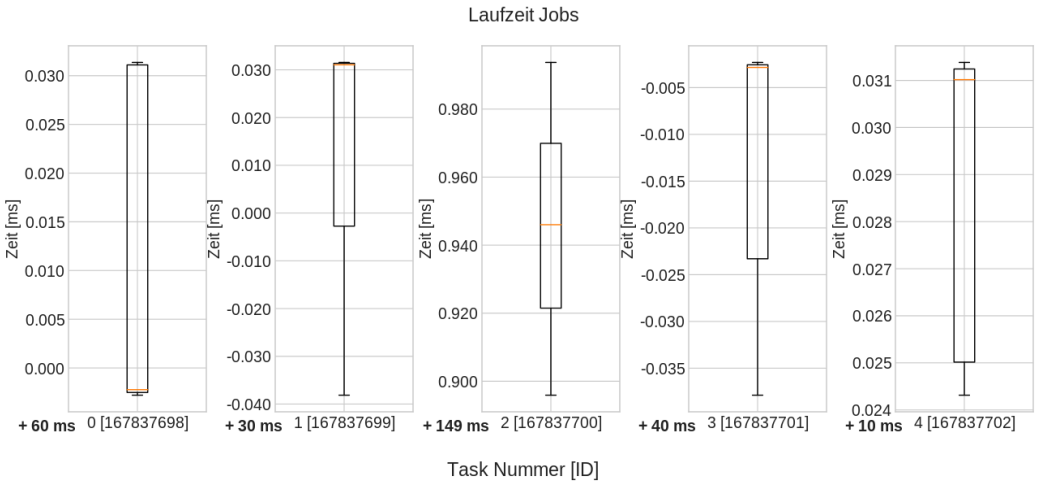


Abbildung 23: Joblaufzeiten der Tasks - ZedBoard (Experiment 5)

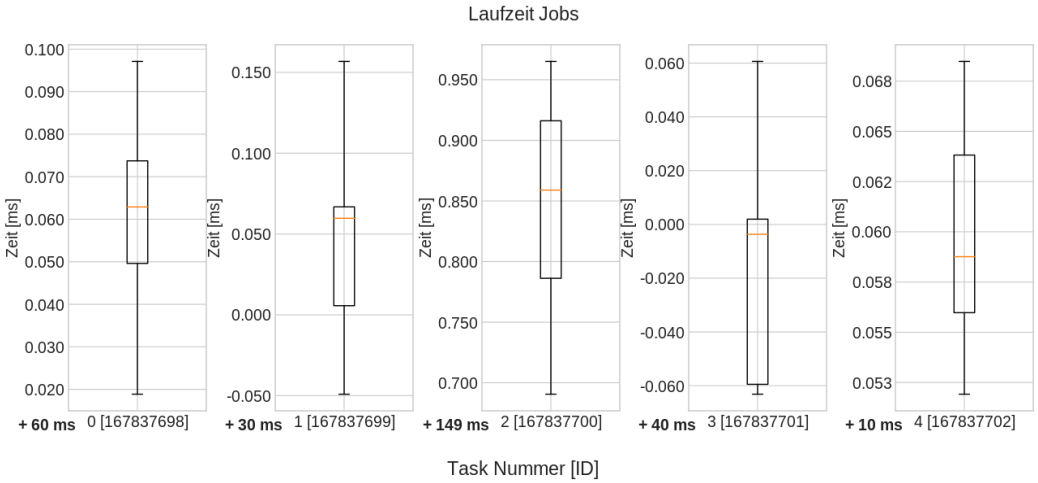


Abbildung 24: Joblaufzeiten der Tasks - QEMU (Experiment 5)

In den Abbildungen 23 und 24 sind die Statistiken der Joblaufzeiten auf dem ZedBoard bzw. in QEMU taskweise dargestellt. Am Referenzwert links unter den Plots ist zu erkennen, dass die Tasks alle mit ihrer zugewiesenen WCET ausgeführt werden. Mit Ausnahme von Task 2 haben jedoch alle Tasks eine etwas höhere Laufzeit als die WCET. Dies kann darauf zurückgeführt werden, dass die festgelegte WCET in der Busy-Wait-Funktion verbraucht wird, der Task aber noch etwas Zeit für weitere Funktionsaufrufe benötigt. Auch bei den Joblaufzeiten sind die Abweichungen in der QEMU-Ausführung etwas größer.

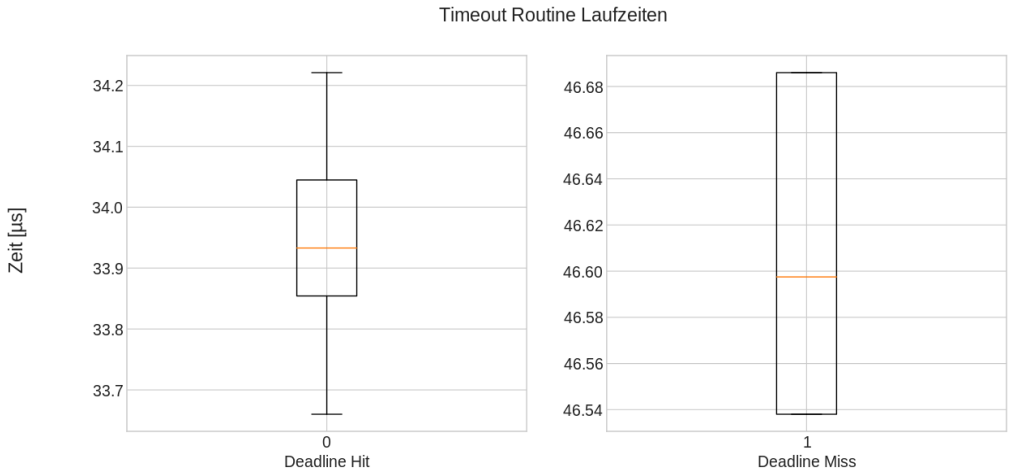


Abbildung 25: Laufzeiten der Timeout Routine - ZedBoard (Experiment 5)

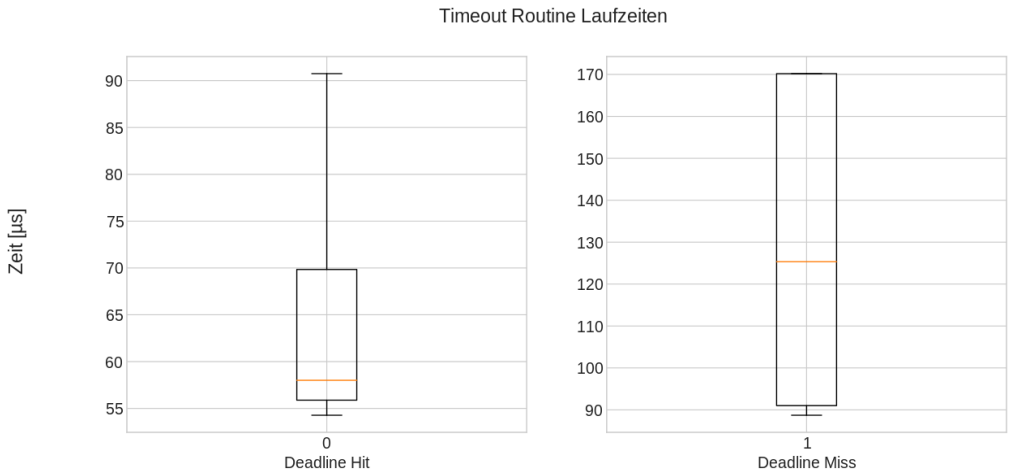


Abbildung 26: Laufzeiten der Timeout Routine - QEMU (Experiment 5)

Die Abbildungen 25 und 26 zeigen die Laufzeiten der Timeout-Routine in Abhängigkeit davon, ob die Routine eine verpasste oder eine eingehaltene Deadline behandelt. Zu den Messungen ist anzumerken, dass nur wenige Daten über die verpassten Deadlines vorliegen, da in allen Experimenten der Großteil der Deadlines eingehalten wurde.

Der Scheduler benötigt auf dem ZedBoard etwa 34 µs für eine eingehaltene Deadline und etwa 46,6 µs nach einer verpassten Deadline. Dabei treten nur sehr geringe Abweichungen < 1 µs auf. Im Gegensatz dazu sind die Laufzeiten in QEMU wesentlich höher: Für eine eingehaltene Deadline beträgt die Laufzeit etwa 58 µs, bei einer verpassten Deadline etwa 125 µs. Darüber hinaus treten in QEMU wesentlich höhere Abweichungen in den Laufzeiten auf.

Obwohl das Hostsystem der QEMU-Experimente wesentlich leistungsstärker ist, sind alle zuvor untersuchten Metriken in QEMU höher. Dies liegt vermutlich daran, dass das Hostbetriebssystem neben den Experimenten noch viele weitere Tasks schedulen muss. Zudem müssen die Instruktionen der ARM-kompilierten Testanwendung in die Instruktionen des Hostbetriebssystems übersetzt werden, was zusätzlichen Overhead verursacht.

Experiment 6

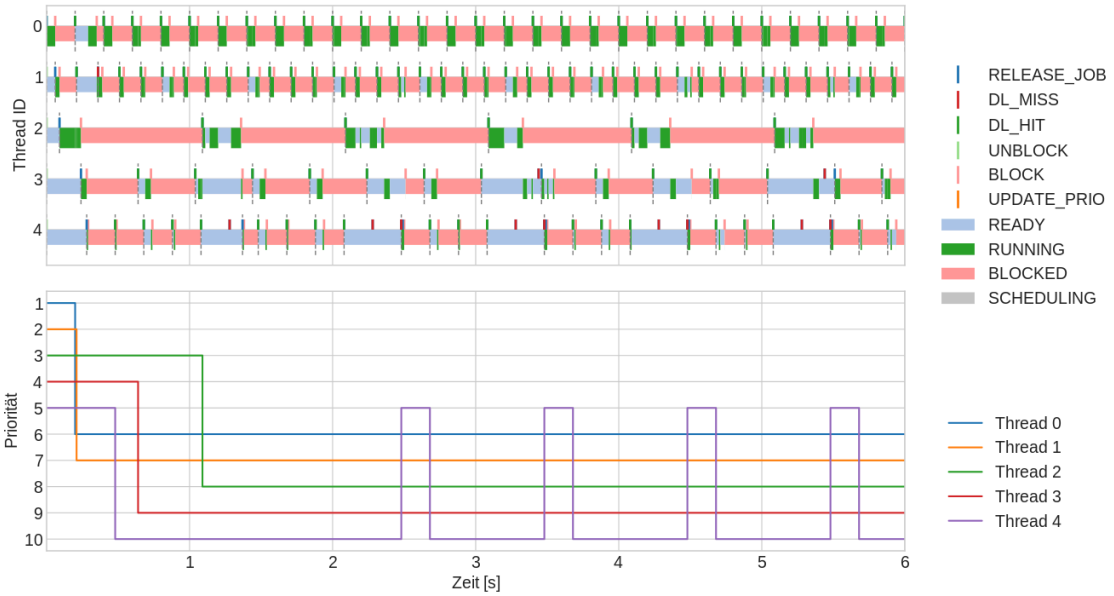


Abbildung 27: Experiment 6 - ZedBoard

In Experiment 6 wurde das Taskset mit hoher Auslastung und High-Tolerance-Tasks verwendet. Das resultierende Tracedatendiagramm ist in Abbildung 27 dargestellt. Es ist zu erkennen, dass die Tasks sehr schnell in ihre niedrigste Priorität wechseln. Der Vergleich mit Experiment 5 zeigt, dass trotz gleicher Auslastung mehr Deadlines verpasst werden. Insbesondere Thread 4, der die niedrigste Priorität hat, verpasst aufgrund von Interferenzen mit anderen Tasks häufig zwei Deadlines. Er wechselt dann kurzzeitig auf seine höchste Priorität.

Experiment 7

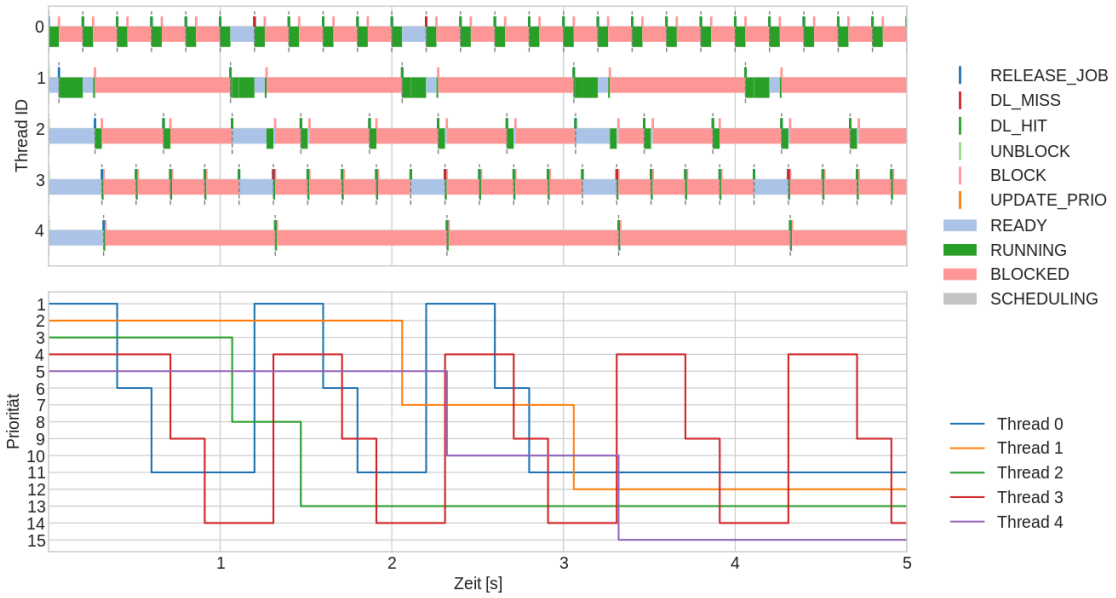


Abbildung 28: Experiment 7 - ZedBoard

In diesem Experiment wurde das Taskset 2, das eine geringe Gesamtlast hat, mit Low-Tolerance-Tasks ausgeführt. Wie in den anderen Experimenten konvergiert der Traceverlauf zu einer sich wiederholenden Form (Abbildung 28). Insgesamt gibt es sehr wenige verpasste Deadlines, nur in Thread 3 gibt es regelmäßig verpasste Deadlines. Das Muster ist ähnlich wie in Experiment 5.

Betrachtet man den Zusammenhang zwischen den Threads in beiden Experimenten, so zeigt sich, dass die regelmäßig verpassten Deadlines in der Periode des Threads mit der höchsten Auslastung auftreten. In Experiment 5 ist dies Thread 2, in Experiment 7 ist das Thread 1. Dies demonstriert, dass durch hochpriorisierte Threads mit relativ hohen Laufzeiten das Verpassen von Deadlines in Threads mit niedriger Priorität induziert wird.

Experiment 8

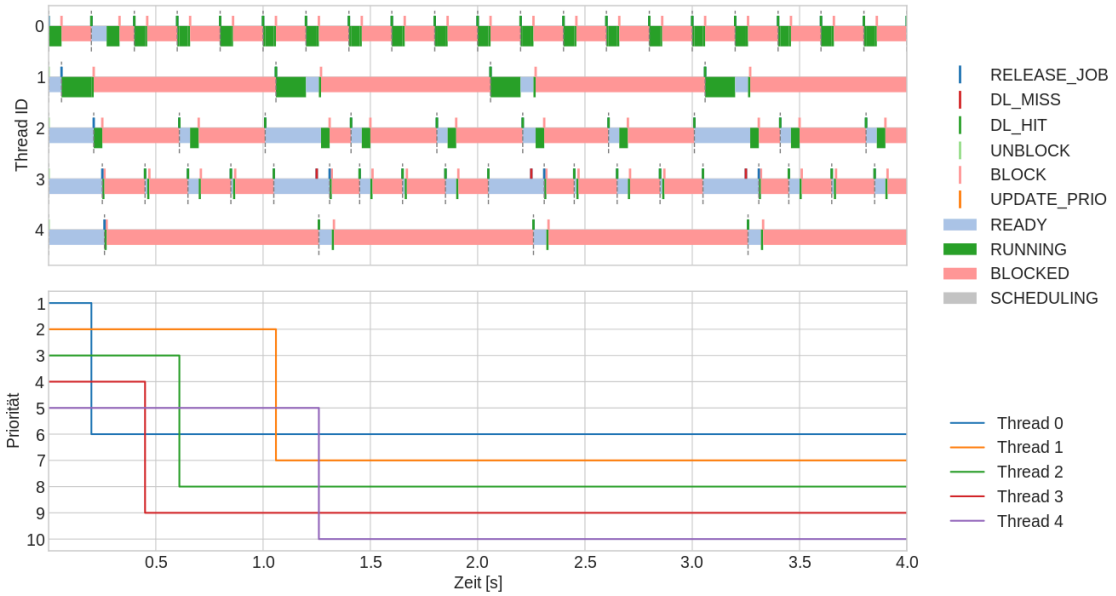


Abbildung 29: Experiment 8 - ZedBoard

Im Rahmen dieses Experiments wurde das Taskset 2 mit einer geringen Last und Tasks mit einer hohen Toleranz ausgeführt. Das entsprechende Trace-Diagramm ist in Abbildung 29 dargestellt. Wie bereits in Experiment 6 beobachtet, führt die Verwendung von High-Tolerance-Tasks zu einer schnellen Reduzierung der Prioritäten.

In diesem Experiment wurden zwar einige Deadlines verpasst, aber deutlich weniger als in Experiment 6. Der Grund dafür ist wahrscheinlich die geringere Gesamtlast. Alle Threads verbleiben in ihrer niedrigsten Priorität.

6.5. Diskussion der Ergebnisse

Um die Funktionsweise des implementierten Weakly Hard Schedulers zu analysieren und zu verifizieren, wurden verschiedene Experimente durchgeführt. Dazu wurden Tasksets definiert und in Testanwendungen je nach Experiment auf dem ZedBoard, in QEMU oder in einer Python-Simulation ausgeführt.

Während der Ausführung wurden Tracedaten zur anschließenden Visualisierung und Auswertung gesammelt. Diese haben gezeigt, dass der implementierte Scheduling-Algorithmus die in Abschnitt 4. beschriebenen Anforderungen erfüllt. Der Scheduler führt eine korrekte Aktualisierung der Joblevel am Ende einer Periode durch, wodurch die Prioritäten je nach Weakly Hard Parametern zurückgesetzt oder verringert werden.

Die Timeout-Routine erneuert die Perioden ebenfalls korrekt, wie die Auswertung der Metriken zeigt. Durch die erfolgreiche Ausführung auf der Hardware wurde demonstriert, dass der Scheduler auch bereits in der Praxis verwendet werden kann.

Beim Vergleich der Experimente mit der Simulation wurden Unterschiede in den Tracedaten festgestellt. Das Taskset mit 100 % Auslastung kann vom implementiertem WHA Scheduler nicht fehlerfrei ausgeführt werden, weil Deadlines in der höchsten Priorität des jeweiligen Threads verpasst werden. Dabei handelt es sich zwar nur um einen soft error, der keinen Ausfall des Kernels hervorruft, aber das Taskset wird nicht nach den Anforderungen des weakly hard Modells ausgeführt.

Die Unterschiede zwischen Simulation und Testlauf sind auf die Auslastung von 100% und die ausschließliche Verwendung von Low-Tolerance-Tasks zurückzuführen. Bei der Ausführung der Jobs verbringen diese ihre WCET in der Funktion Busy Wait. Da weitere Funktionsaufrufe in den Tasks und der Scheduler selbst zusätzlichen Overhead verursachen, kommt es im Experiment zu verpassten Deadlines. Die Simulation hingegen berücksichtigt keinen Overhead, weshalb sie das Taskset ausführen kann.

Bei der Verwendung von High-Tolerance-Tasks hingegen können auch Tasksets mit einer Gesamtauslastung von über 100% ausgeführt werden. In diesem Fall treten zwar viele verpasste Deadlines auf, aber ein entsprechendes Taskset ist mit geeigneten weakly hard Parameterkonfigurationen trotzdem schedulable.

In der aktuellen Implementierung kann ein periodischer Job an der *EXPIRED* Flag erkennen, ob er eine Deadline verpasst hat. Schedulability-Fehler hingegen werden bisher nur zur nachträglichen Auswertung durch einen Trace-Punkt erfasst. Für eine Behandlung des Fehlers durch die Anwendung wäre es hilfreich, z.B. ein *WHA_ISSUE* Flag zu setzen, die einen Fehler im Scheduling für den entsprechenden Job markiert.

Aus den Experimenten mit fünf Tasks konnten verschiedene Erkenntnisse über den Zusammenhang zwischen Prioritäten, Auslastung und Toleranz von Tasks gewonnen werden. In Abhängigkeit von der Auslastung erhöht oder senkt der Scheduler die Prioritäten der Tasks. Über die gesamte Laufzeit betrachtet, verursachen Jobs mit hoher Last periodisch kurzzeitige Lastanstiege. Dadurch verpassen andere Jobs, die zu diesem Zeitpunkt eine niedrigere Priorität haben, ihre Deadline und erhöhen vorübergehend ihre Priorität. Die Empfindlichkeit dieses Verhaltens, d.h. wie schnell ein Task seine Priorität erhöht, hängt von den Weakly Hard-Parametern des entsprechenden Tasks ab. Bei der Verwendung von High Tolerance wird die Priorität erst nach mehreren

verpassten Deadlines erhöht. Wie der Vergleich der Experimente 5 und 6 zeigt, kann die Verwendung von High Tolerance Tasks jedoch auch dazu führen, dass bei gleichem Taskset mehr Deadlines verpasst werden.

Es ist zu beachten, dass verpasste Deadlines auch bei einer Gesamtlast von $< 100\%$ auftreten können. Die Zuordnung der Prioritäten zu den Tasks wurde ohne Sortierung vorgenommen, daher haben Tasks mit einer großen Periode teilweise relativ hohe Prioritäten. Diese können ihre Jobs meist früh abschließen und beanspruchen daher nur ein kleines Intervall ihrer Periode. Andere Jobs mit niedrigen Prioritäten und kurzen Perioden verpassen dadurch Deadlines. Eine Sortierung der Tasks nach ihrer Periodenlänge, so wie beim Rate Monotonic Scheduling, würde die Anzahl der verpassten Deadlines reduzieren, da die Jobs von Tasks mit kurzen Perioden so höher priorisiert wären.

Außerdem ist zu berücksichtigen, dass es sich bei den Experimenten nur um die Ausführung synthetischer Tasksets handelt. Obwohl die Experimente einen Einblick in die Ausführbarkeit und die Laufzeiten geben, wären Tests in realen Anwendungsfällen erforderlich, um die tatsächliche Effizienz des Schedulers zu ermitteln.

7. Fazit und Ausblick

7.1. Zusammenfassung der Arbeit

Diese Bachelorarbeit beschäftigt sich mit der Implementierung und Evaluierung eines weakly-hard aware Schedulers im Real-Time Operating System RTEMS. In vielen Real-Time Anwendungen werden harte Real-Time Modelle verwendet, bei denen die ausgeführten Tasks alle Deadlines einhalten müssen, obwohl es oft tolerierbar wäre, einige Deadlines zu verpassen. Der in dieser Arbeit implementierte Scheduling Algorithmus verwendet das Weakly-Hard Real-Time Modell, welches das gelegentliche Verpassen von Deadlines erlaubt.

Zur Implementierung des von Moyano beschriebenen Algorithmus, wurde die RTEMS Scheduler API verwendet. Dieser Algorithmus unterteilt jeden Task in Jobklassen und weist ihnen Prioriräten zu. Basieren auf einem Joblevel, das beim Verpassen bzw. Einhalten von Deadlines aktualisiert wird, verwendet der Algorithmus für jeden Job eine bestimmte Priorität. Auf diese Weise kann der Weakly-Hard-Aware-Scheduler die Prioritäten an die Arbeitslast anpassen und gleichzeitig die Zuverlässigkeit des Systems bei sehr hohen Lasten gewährleisten.

Der Scheduler wurde zunächst in QEMU getestet und anschließend auf echter Hardware, dem Xilinx Zynq ZedBoard, validiert. Die Evaluierung umfasste eine Reihe von Experimenten, in denen die korrekte Funktionsweise des Schedulers verifiziert und mit einer Python-Simulation desselben Algorithmus verglichen wurde. Zu diesem Zweck wurden in den Experimenten Tracing-Funktionen verwendet, um Daten über den Scheduler zu sammeln und anschließend visuell zu analysieren sowie verschiedene Metriken auszuwerten.

7.2. Kritische Reflexion

Die in dieser Arbeit vorgestellte Scheduler-Implementierung zeigt das Potential und die Herausforderungen des Weakly-Hard-Modells. Es wird deutlich, dass ein solcher Scheduler durch die dynamische Vergabe von Prioritäten und die Toleranz gegenüber verpassten Deadlines sehr gut mit schwankenden Systemlasten umgehen kann und sogar Auslastungen von über 100% bewältigen kann. Durch die erfolgreiche Ausführung

auf der Hardware konnte gezeigt werden, dass der Scheduler auch bereits in der Praxis eingesetzt werden kann.

Die Implementierung und Analyse eines solchen Systems ist jedoch aufgrund der Komplexität des Modells mit einem hohen Aufwand verbunden. Die Implementierung in den RTEMS Kernel wurde erfolgreich durchgeführt, wobei bestehende Komponenten des Rate Monotonic Managers zur Steuerung der periodischen Tasks modifiziert werden mussten. Dies wurde auf diese Weise implementiert, da die RTEMS Scheduler API nicht optimal für die Unterstützung von Schedulern für periodische Tasks ausgelegt ist. Obwohl der Rate Monotonic Manager die notwendigen Funktionen für periodische Tasks enthält, sind diese nicht Teil der Scheduler API. Das bedeutet, dass ein neuer Scheduler diese Funktionen vollständig selbst implementieren müsste.

7.3. Ausblick

Diese Arbeit hat gezeigt, dass der Weakly-Hard Scheduling Algorithmus für Einprozessorsysteme funktioniert und anwendbar ist. Um ihn in modernen Systemen effektiv einsetzen zu können, wäre es sinnvoll, ihn für die Unterstützung von Mehrprozessorsystemen weiterzuentwickeln. In diesem Zusammenhang wäre auch eine vollständige Trennung des in RTEMS implementierten Schedulers vom Rate Monotonic Manager sinnvoll, so dass ein eigenständiger Scheduler verfügbar wäre.

Der Vergleich mit der Simulation hat gezeigt, dass es Unterschiede gibt. Dementsprechend könnten die Experimente genutzt werden, um die Eigenschaften des Scheduling-Algorithmus in der Simulation abzubilden, um eine realistischere Simulation zu ermöglichen.

Literaturverzeichnis

[1] G. Bernat, A. Burns, und A. Liamosi, „Weakly hard real-time systems“, *IEEE Transactions on Computers*, Bd. 50, Nr. 4, S. 308–321, Apr. 2001, doi: 10.1109/12.919277.

[2] G. Moyano und Z. Hammadeh, „Response time analysis for weakly-hard real-time tasks under global scheduling“, 2024.

[3] J. Stankovic, „Misconceptions about real-time computing: a serious problem for next-generation systems“, *Computer*, Bd. 21, Nr. 10, S. 10–19, Okt. 1988, doi: 10.1109/2.7053.

[4] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*, Fourth edition. Cham, Switzerland: Springer, 2024. doi: 10.1007/978-3-031-45410-3.

[5] M. Joseph und P. Pandya, „Finding Response Times in a Real-Time System“, *The Computer Journal*, Bd. 29, Nr. 5, S. 390–395, Jan. 1986, doi: 10.1093/comjnl/29.5.390.

[6] T. Baker, „Multiprocessor EDF and deadline monotonic schedulability analysis“, in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, Dez. 2003, S. 120–129. doi: 10.1109/REAL.2003.1253260.

[7] M. Bertogna und M. Cirinei, „Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms“, in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, Dez. 2007, S. 149–160. doi: 10.1109/RTSS.2007.31.

[8] A. Burns, „Scheduling hard real-time systems: a review“, *Software Engineering Journal*, Bd. 6, Nr. 3, S. 116–128, 1991.

[9] C. Johns, J. Sherrill, S. Hubner, B. Gras, und G. Bloom, „FreeBSD and RTEMS, Unix in a Real-Time Operating System“, Aug. 2016, [Online]. Verfügbar unter: https://freebsdfoundation.org/wp-content/uploads/2016/08/FreeBSD-and-RTEMS-Unix-in-a-Real-Time-Operating-System.pdf

[10] M. Hamdaoui und P. Ramanathan, „A dynamic priority assignment technique for streams with (m, k)-firm deadlines“, *IEEE Transactions on Computers*, Bd. 44, Nr. 12, S. 1443–1451, Dez. 1995, doi: 10.1109/12.477249.

[11] H. Choi, H. Kim, und Q. Zhu, „Job-Class-Level Fixed Priority Scheduling of Weakly-Hard Real-Time Systems“, in *2019 IEEE Real-Time and Embedded Techno-*

logy and Applications Symposium (RTAS), Apr. 2019, S. 241–253. doi: 10.1109/RTAS.2019.00028.

[12] R. I. Davis und A. Burns, „A survey of hard real-time scheduling for multiprocessor systems“, *ACM Computing Surveys*, Bd. 43, Nr. 4, S. 1–44, Okt. 2011, doi: 10.1145/1978802.1978814.

[13] G. Koren und D. Shasha, „Skip-Over: algorithms and complexity for overloaded systems that allow skips“, in *Proceedings 16th IEEE Real-Time Systems Symposium*, Dez. 1995, S. 110–117. doi: 10.1109/REAL.1995.495201.

[14] C. L. Liu und J. W. Layland, „Scheduling algorithms for multiprogramming in a hard-real-time environment“, *Journal of the ACM (JACM)*, Bd. 20, Nr. 1, S. 46–61, 1973.

[15] L. Sha und J. B. Goodenough, „Real-time scheduling theory and Ada“, *IEEE Computer*, Bd. 23, Nr. 4, S. 53–62, 1990.

[16] F. Brömer, P. Kenny, A. Lund, C. Gremzow, und D. Lüdtke, „FPGA-based fault injector for SEU-robustness analysis of ScOSA“, in *Deutscher Luft- und Raumfahrtkongress 2022*, Deutsche Gesellschaft für Luft- und Raumfahrt, Sep. 2022. [Online]. Verfügbar unter: https://elib.dlr.de/189225/

[17] T. Freitag, „Acceleration of an Autoencoder using a FPGA-SoC in a High-Performance Node of a Distributed Onboard Computer“, Dezember 2022. [Online]. Verfügbar unter: https://elib.dlr.de/192913/

[18] F. Eichstaedt, J. Budroweit, und F. Stehle, „Investigation of the Zynq-7000 Integrated XADC under Proton Irradiation“, in *2023 IEEE Radiation Effects Data Workshop (REDW)(in conjunction with 2023 NSREC)*, IEEE, 2023, S. 1–4.

[19] M. Koenig, „Cross compiling and debugging for RISC-V64 with Qemu and VS Code“. Zugegriffen: 25. Juli 2024. [Online]. Verfügbar unter: https://medium.com/@e1d1/cross-compiling-and-debugging-for-risc-v64-with-qemu-and-vs-code-de36b262d2f2

[20] the RTEMS Project, „RTEMS Classic API Guide“. Zugegriffen: 23. Juli 2024. [Online]. Verfügbar unter: https://docs.rtems.org/branches/master/c-user/index.html

[21] the RTEMS Project, „RTEMS Classic API: Task Manager“. Zugegriffen: 23. Juli 2024. [Online]. Verfügbar unter: https://docs.rtems.org/branches/master/c-user/task/background.html

[22] L. Bonato, „RTEMS Internals Manual -how the kernel works-“. Zugegriffen: 23. Juli 2024. [Online]. Verfügbar unter: https://ftp.rtems.org/pub/rtems/people/sebh/rtems-4.11-internals-manual-luca-bonato.pdf

[23] the RTEMS Project, „RTEMS Classic API: Scheduling Concepts“. Zugegriffen: 23. Juli 2024. [Online]. Verfügbar unter: https://docs.rtems.org/branches/master/c-user/scheduling-concepts/background.html

[24] K.-H. Chen, G. von Der Brüggen, und J.-J. Chen, „Overrun handling for mixed-criticality support in RTEMS“, in *WMC 2016*, 2016.

[25] J. Morris, „Red-Black Trees“. Zugegriffen: 1. August 2024. [Online]. Verfügbar unter: https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html