Faculty of Electrical Engineering,
Mathematics & Computer Science

Efficient Task Dispatching for Real-Time Systems: A Case Study in FreeRTOS

Florian Hagens
M.Sc. Thesis
June 2023

Supervisors:
dr. ing. K.H. Chen
dr. ir. A.L. Varbanescu
dr. ing. A. Chiumento

Computer Architecture for
Embedded Systems group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
Drienerlolaan 5
7522 NB Enschede
The Netherlands

Abstract

Real-time operating systems (RTOS) are vital for managing time-sensitive applications and ensuring that tasks are executed according to strict deadlines. A key component of RTOS is the task dispatcher, which plays a critical role in task periodicity. The primary objective of optimizing task dispatchers is to reduce computation overhead, leading to more efficient systems and better guarantees of meeting deadlines. This thesis presents an in-depth evaluation of various implemented task dispatching methods based on five distinct data structures, focusing on their impact on computation overhead and performance in FreeRTOS.

Through rigorous experimentation using a real-world setup, we assessed the merits and drawbacks of each data structure and its corresponding task dispatcher implementation. Our findings revealed that the efficiency of a task dispatcher is heavily influenced by the specific task set and its size. For smaller task sets, the current List-based implementation in FreeRTOS is satisfactory; however, its efficiency diminishes as the task set size increases. We identified cases where alternative task dispatchers, such as those based on Red-Black Trees (RBT) and Heaps, outperformed the List-based implementation. Moreover, we observed that certain dispatching methods are better suited for specific task set configurations.

Our research contributes to the existing knowledge on task dispatching methods in real-time operating systems and provides valuable insights for system designers and developers in optimizing their task dispatchers. By examining the performance of different dispatchers under various task sets and configurations, we highlight the potential for improved system performance in specific scenarios. Additionally, our study emphasizes the importance of tailoring the task dispatcher to the specific task set at hand, which can contribute to the development of more efficient, reliable, and high-performance real-time systems.

Acknowledgment

I would like to express my sincere gratitude to my thesis supervisor, Dr. Kuan-Hsun Chen, for his insightful expertise, practical advice, and the opportunity to work under his guidance. His inputs during our weekly meetings and responsiveness to my progress updates were instrumental in the successful completion of this thesis. Furthermore, our collaboration on the papers that have been accepted by OSPERT and CompSys provided an enlightening experience. I appreciate the opportunity to have developed this thesis under your guidance, Dr. Chen. Thank you for your support and the learning experience.

Contents

Abstract 1

1 Introduction 13

2 Background 17
- 2.1 Real-Time Systems . . . 17
- 2.2 Real-Time Operating Systems . . . 19
- 2.3 Task Dispatchers . . . 22
  - 2.3.1 Task Dispatchers in General Purpose Operating Systems . . . 22
  - 2.3.2 Task Dispatchers in Real-time Operating Systems . . . 22
- 2.4 FreeRTOS . . . 24
  - 2.4.1 Task Dispatcher in FreeRTOS . . . 25

3 Real-world Measurement Setup 33
- 3.1 ESP-IDF FreeRTOS . . . 34
- 3.2 Embedded Device and PlatformIO . . . 35
- 3.3 GDB & OpenOCD . . . 36
- 3.4 Measurements . . . 38

4 Methodology 41
- 4.1 Task Dispatching Data Structures . . . 42
  - 4.1.1 Time Complexities . . . 48
- 4.2 Task Dispatcher Implementations . . . 51
  - 4.2.1 Implementation to FreeRTOS . . . 52
  - 4.2.2 List . . . 56
  - 4.2.3 Bucket of Ignorance . . . 57
  - 4.2.4 Binary Search Tree . . . 57
  - 4.2.5 Heap . . . 59
  - 4.2.6 Red-Black Tree . . . 61
- 4.3 FreeRTOS Test Application . . . 62
- 4.4 Limitations of Testing on Embedded Devices . . . 62

5 Evaluation **65**
5.1 Task Set Synthesis . . . 65
5.2 Worst-Case Computation Overhead . . . 67
5.2.1 List . . . 68
5.2.2 Bucket of Ignorance . . . 69
5.2.3 Binary Search Tree . . . 70
5.2.4 Heap . . . 71
5.2.5 Red-Black Tree . . . 73
5.2.6 Comparison of Worst-Case Computation Overhead . . . 76
5.3 Task Sets with Homogeneous Period Distribution . . . 78
5.3.1 List . . . 79
5.3.2 Bucket of Ignorance . . . 80
5.3.3 Binary Search Tree . . . 81
5.3.4 Heap . . . 82
5.3.5 Red-black tree . . . 84
5.3.6 Comparison of Homogeneous Period Task Sets Overhead . . . 85
5.4 Task Sets with Uniform Period Distribution . . . 88
5.5 Automotive Benchmark Period Distribution . . . 92

6 Conclusion **95**

References **97**

Appendices

A Pseudo code implemented data structure functions **101**
A.1 List . . . 101
A.1.1 Insertion . . . 101
A.1.2 Remove First Task . . . 102
A.2 Bucket of Ignorance . . . 102
A.3 Red-black Tree . . . 104

List of Figures

2.1 Utility function for hard real-time tasks. 18
2.2 Utility function for firm real-time tasks. 18
2.3 Utility function for soft real-time tasks. 18
2.4 Layered scheme for GPOS architecture. 20
2.5 Layered scheme for RTOS architecture. 21
2.6 Task dispatching routine. 23
2.7 FreeRTOS software layers. 25
2.8 Flowchart illustrating the operation sequence for delaying tasks with regards to the task dispatcher in FreeRTOS. 28
2.9 Scenarios for adding the task to the task dispatcher data structure. . . 29
2.10 Flowchart illustrating the operation sequence for tick incrementation with regards to the task dispatcher in FreeRTOS. 31

3.1 Real-world Measurement Setup. 33
3.2 Real-world Interconnects for Debugging: Comparison of `ESP-PROG` (left), `Built-in` (middle and right) Options. 37
3.3 Debugging Options in GDB Visual Studio Code: An Overview of Key Features and Functionality. 39

4.1 FreeRTOS list structure 56
4.2 Flow diagram of the Bucket of Ignorance insertion function. 58
4.3 Flow diagram of the BST insertion function. 59
4.4 Flow diagram of the Array-based Heap insertion function. 60
4.5 Flow diagram of the Array-based iterative first task removal function. . 61

5.1 Worst-case computation overhead of List-based task dispatcher task insertion implementation per task set size. 68
5.2 Worst-case computation overhead of BST-based task dispatcher task insertion implementation per task set size. 70
5.3 Worst-case computation overhead of BST-based task dispatcher first task retrieval implementation per task set size. 71

5.4 Worst-case computation overhead of BST-based task dispatcher first task removal implementation per task set size. 71
5.5 Worst-case computation overhead of Heap-based task dispatcher task insertion implementation per task set size. 72
5.6 Worst-case computation overhead of Heap-based task dispatcher first task removal implementation per task set size. 73
5.7 Worst-case computation overhead of RBT-based task dispatcher task insertion implementation per task set size. 74
5.8 Worst-case computation overhead of RBT-based task dispatcher first task retrieval implementation per task set size. 74
5.9 Worst-case computation overhead of RBT-based task dispatcher first task removal implementation per task set size. 75
5.10 Worst-case computation overhead comparison of different task dispatcher task insertion implementations. 76
5.11 Worst-case computation overhead comparison of different task dispatcher first task retrieval implementations. 77
5.12 Worst-case computation overhead comparison of different task dispatcher first task removal implementations. 78
5.13 Overhead measurements for List-based task dispatcher task insertion implementation for homogeneous task sets. 79
5.14 Overhead measurements for BoI-based task dispatcher task insertion implementation for homogeneous task sets. 80
5.15 Overhead measurements for BST-based task dispatcher task insertion implementation for homogeneous task sets. 81
5.16 Overhead measurements for Heap-based task dispatcher task insertion implementation for homogeneous task sets. 82
5.17 Overhead measurements for Heap-based task dispatcher first task removal implementation for homogeneous task sets. 83
5.18 Overhead measurements for RBT-based task dispatcher task insertion implementation for homogeneous task sets. 84
5.19 Overhead measurements for RBT-based task dispatcher first task retrieval implementation for homogeneous task sets. 85
5.20 Overhead measurements for RBT-based task dispatcher first task removal implementation for homogeneous task sets. 85
5.21 Overhead comparison of different task dispatcher task insertion implementations for homogeneous task sets. 86
5.22 Overhead comparison of different task dispatcher first task retrieval implementations for homogeneous task sets. 87

5.23 Overhead comparison of different task dispatcher first task removal implementations for homogeneous task sets. 88
5.24 Overhead comparison of different task dispatcher task insertion implementations for uniform task sets. 89
5.25 Overhead comparison of different task dispatcher first task retrieval implementations for uniform task sets. 90
5.26 Overhead comparison of different task dispatcher first task removal implementations for uniform task sets. 91

List of Tables

2.1 Definitions of terms used in the task dispatcher description. 27

4.1 Worst-case time complexity of task dispatcher data structures, where n represents the number of items in the data structure and m represents the number of hierarchical arrays. 50
4.2 Average-case time complexity of task dispatcher data structures, where n represents the number of items in the data structure and m represents the number of hierarchical arrays. 51

5.1 Distribution of task periods in automotive benchmark [1]. 66
5.2 Normalized distribution of tasks in automotive benchmark, excluding angle-synchronous tasks. 66
5.3 Worst case computation overhead of List-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher. 69
5.4 Worst-case computation overhead of BST-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher. 70
5.5 Worst-case computation overhead of Heap-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher. 73
5.6 Worst case computation overhead of RBT-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher. 75
5.7 Scaled down automotive distribution measurement results given in CPU cycles [1]. 93

Chapter 1

Introduction

”Give me six hours to chop down a tree,
and I will spend the first four sharpening the axe.”
—Abraham Lincoln

In embedded systems, the efficient and timely execution of tasks is of great importance. These systems often require periodic task executions and adherence to strict deadlines to maintain their functionality and ensure expected operation. Examples of real-world periodic systems include air traffic control systems, automated vehicle control systems, industrial robotics, and heart pacemakers. Periodicity is important in these systems as it ensures that critical operations are performed at regular intervals, maintaining synchronization between different components, and enabling the system to provide consistent and expected performance. Periodicity and deadlines are essential to maintaining system stability, responding to time-sensitive events, and ensuring predictable behavior in real-time applications.

The scheduler and the task dispatcher play a crucial role in meeting these requirements. In our review of the current literature, we noticed a strong focus on various scheduling algorithms, while the task dispatcher seemed to be less extensively explored. Recognizing the importance of the task dispatcher in efficient task execution in embedded systems, this thesis will contribute to filling this perceived gap. The scheduler is responsible for managing and organizing tasks based on their priorities and deadlines, allocating resources, and ensuring that tasks are executed according to their pre-defined periods. Meanwhile, the task dispatcher is responsible for initiating task execution by selecting the most suitable task from the scheduler’s output, introducing the task periodicity.

Optimized task dispatchers can significantly improve the efficiency of task scheduling and thus overall system performance. By basing task dispatchers on suitable data structures, we can reduce computational overhead, thereby enhancing the predictability of the system’s behavior. In the realm of task dispatcher optimization,

several notable studies have contributed to the development of improved methods. For instance, the potential of hardware-based scheduling in real-time operating systems (RTOS) [2] [3] [4]. Additionally, research on efficient data structures for timers, such as hashed and hierarchical timing wheels, provide valuable insights for the development of more effective task dispatchers [5]. Furthermore, the work on improved task management techniques for enforcing EDF scheduling on recurring tasks highlights the importance of refining task management processes to enhance task dispatcher efficiency [6]. The presence of these studies in the literature serves as a strong foundation for the current research, as they demonstrate the importance of optimizing task dispatchers to improve overall system performance. The fact that researchers have dedicated their efforts to investigating task dispatcher optimization underscores the significance and potential impact of such optimizations in the field of embedded systems.

Abraham Lincoln's quote, "Give me six hours to chop down a tree, and I will spend the first four sharpening the axe," serves as a fitting metaphor for the central theme of this thesis. Much like how Lincoln emphasized the importance of investing time in selecting and preparing the right tool for the task at hand, this research focuses on implementing and evaluating various task dispatching methodologies. By doing so, the research aims to identify the most efficient dispatching implementation for a diverse range of task sets, thus ensuring that the "axe" of real-time operating systems is sharpened for optimal performance.

In order to address this possible improvement, this research adopts a systematic and practical approach, focused on implementing and evaluating the task dispatcher of FreeRTOS based on five distinct data structures. To address the potential improvement of reducing computation overhead in task dispatching, this research focuses on implementing and evaluating the task dispatcher of FreeRTOS using five distinct data structures: List, Bucket of Ignorance (BoI), BST (Binary Search Tree), Heap and Red-Black Tree (RBT). The selection of data structure enhancement is predicated on the assumption that the task dispatcher's efficiency is primarily impacted by the underlying data structure employed to organize tasks. The chosen data structures will serve as the foundation for exploring how each implementation affects the computational overhead of the task dispatcher.

In this thesis, we aim to provide a detailed comparison of selected task dispatcher implementations suitable for embedded systems, mainly based on computation overhead. Our methodology involves both a theoretical analysis and practical implementation of these task dispatchers on an actual embedded device, specifically the ESP32-S3-DevKitC-1. By doing so, we ensure that the findings are grounded in real-world application scenarios and offer a high-level overview of each implementation's advantages and disadvantages.

This hands-on approach will allow for an accurate assessment of each task dispatcher's effectiveness when integrated into a functional system. The research evaluates their computational overhead, thus providing valuable insights for developers and researchers working in the field of embedded systems.

In order to evaluate the performance of the different task dispatcher implementations, this study utilizes CPU cycle measurements as the primary metric. Specifically, we focus on the CPU cycles consumed during the crucial operations of task insertion, first task retrieval, and first task removal within the kernel. CPU cycles offer a precise, low-level measure of computational effort, allowing us to accurately quantify the overhead associated with each task dispatcher implementation. This method enables a direct comparison of each implementation's efficiency, thereby shedding light on their relative strengths and weaknesses.

This thesis is structured into several chapters, each addressing a different aspect of the research on the implementation and evaluation of the different task dispatcher implementations in FreeRTOS. Below is a brief outline of how the thesis is organized and what each chapter contains:

- **Background:** This chapter lays the foundation for the rest of the thesis, providing essential information on RTOSes, task dispatchers and FreeRTOS.
- **Real-World Measurement Setup:** This chapter describes the experimental setup used for evaluating the task dispatcher implementations. It discusses some of the tools, hardware and software that is used in this research, to ensure a good understanding of the environment used for the evaluation of the task dispatcher implementations.
- **Methodology:** In this chapter, we will detail the methodology used in this research. We begin by exploring potential data structures that could serve as the basis for task dispatchers. Next, we delve into the implementation process for selected task dispatchers that use these data structures. We then describe the test application developed to generate task sets, which will be used to evaluate the task dispatchers. Subsequently, we present the limitations of the utilized embedded device, the ESP32-S3-DevKitC-1. Understanding these limitations is crucial as it provides context for the evaluation of the different task dispatcher implementations.

- **Evaluation:** This chapter presents the evaluation of the various task dispatcher implementations using different sets of task sets. The performance of each implementation is analyzed based on a real-world automotive benchmark, where we measure the worst-case scenario, homogeneous period distribution, uniform period distribution and a distribution according to the automotive benchmark.

- **Conclusion:** The Concluding chapter summarizes the key findings of the research, offers insights into the implications of the results in the context of RTOS development and outlines potential avenues for future research.

To ensure a thorough evaluation of the work presented in this thesis, the complete source code has been made available exclusively to the reviewers. The code can be accessed through two separate repositories, which can be found at the following links:

- FreeRTOS kernel with additional dispatcher implementations: `https://gitlab.utwente.nl/s2626160/pio_freertos`

- Test application with easily adjustable task sets: `https://gitlab.utwente.nl/s2626160/generic_freertos_measuring_application`

Chapter 2

Background

This chapter of the thesis provides an essential foundation for the research presented in the subsequent chapters. In this chapter, we discuss the fundamental concepts and components of real-time operating systems, with a specific focus on task dispatchers and their use in FreeRTOS.

We will go over the basics of real-time systems and their characteristics, as well as explore the role of real-time operating systems in managing and executing time-critical tasks. A key aspect of RTOSes (Real-Time Operating Systems) is the task dispatcher, which is responsible for task periodicity. We will examine the differences between task dispatchers in general-purpose operating systems and real-time operating systems.

Furthermore, we will introduce FreeRTOS, a widely adopted real-time operating system and discuss its task dispatcher implementation in detail. By gaining a good understanding of the concepts and mechanisms presented in this background section, readers will be well-equipped to comprehend the research and analysis that follow. This foundational knowledge is crucial for both the evaluation of our work and the broader context of real-time systems and operating systems in general.

Overall, this chapter provides a background on RTOSes, task dispatchers and FreeRTOS, which is essential to understand the motivation and context of this research.

2.1 Real-Time Systems

Real-time systems (RTSs) are designed to provide predictable and responsive behavior, maintaining reliability as a priority over sheer performance. A fundamental aspect of RTSs is ensuring that processing tasks are executed within specified timeframes, irrespective of the system load. A critical distinction arises between the logical correctness of the processing results and their adherence to the required

timing constraints. Consequently, real-time systems are classified according to their deadlines, real-time tasks can be distinguished into three categories [7]:

- **Hard:** Producing results after its deadline may cause catastrophic consequences on the system under control. (Figure 2.1)
- **Firm:** Producing results after its deadline is useless for the system but does not cause any damage. (Figure 2.2)
- **Soft:** Producing results after its deadline still has some utility for the system, although causing performance degradation. (Figure 2.3)

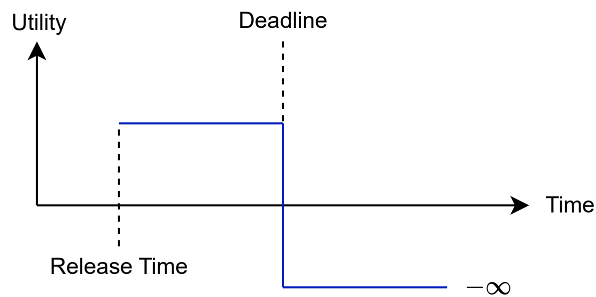

Figure 2.1: Utility function for hard real-time tasks.

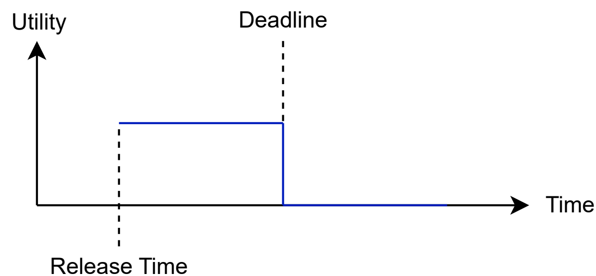

Figure 2.2: Utility function for firm real-time tasks.

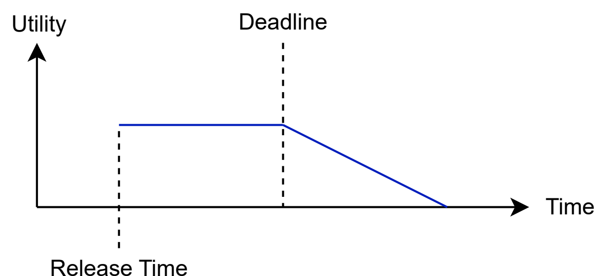

Figure 2.3: Utility function for soft real-time tasks.

In real-time systems, tasks are composed of smaller execution entities known as jobs (or task instances). Each job, denoted as J_i, can be characterized by a set of fundamental parameters: $J_i = (r_i, C_i, d_i)$, where:

- r_i is the release time (or arrival time a_i): The time at which the job becomes ready for execution.
- C_i is the computation time or Worst-Case Execution Time (WCET): The time necessary for the processor to execute the job without interruption.
- d_i is the absolute deadline: The time by which the job should be completed.

Activating a job by a fixed interval of time, a task is said to be periodic. A periodic task denoted as τ_i has its first job $\tau_{i,1}$ activated at time ϕ_i, known as the task phase. Subsequent job activations $\tau_{i,j+1}$, occur at time $r_{i,j+1} = r_{i,j} + T_i$, where T_i represents the task period.

Tasks can also exhibit irregular job activation patterns, leading to the classification of aperiodic tasks. Specifically, an aperiodic task τ_i is defined such that the activation time of job $\tau_{i,j+1}$ is greater than or equal to that of its previous job ($r_{i,j+1} \geq r_{i,j}$)

When an aperiodic task has a defined minimum time interval between the activations of successive jobs, it is referred to as a sporadic task. In other words, a sporadic task τ_i is a task in which the time difference between the activations of two adjacent jobs $\tau_{i,j}$ and $\tau_{i,j+1}$ is never less than a certain value T_i, denoted by $r_{i,j+1} \geq r_{i,j} + T_i$. In this context, T_i is referred to as the minimum interarrival time. This concept is similar to the task period used in the definition of periodic tasks, but it represents a minimum limit rather than a fixed interval.

To realize RTSs, an RTOS is needed to ensure that the system behavior is predictable and the OS should manage the timing and scheduling of the tasks [8]. In the following section, we will compare RTOSes with General-Purpose Operating Systems (GPOS) to motivate the need for efficient task dispatchers.

2.2 Real-Time Operating Systems

The first distinction between General Purpose Operating Systems (GPOS) and RTOSes lies in their application scope and programming approach. GPOS are designed to support a broad range of unknown applications and permit regular programming by end-users. In contrast, RTOSes cater to predefined applications, and programming is exclusively performed by system designers.

Another critical difference concerns the manner in which system resources are accessed. In GPOS, processes typically do not access hardware directly, as depicted in Figure 2.4. When an application requires hardware access, a system call

is made either directly to the OS or through the corresponding middleware. The OS then interacts with the hardware, either directly or via an appropriate driver, and responds accordingly. The OS kernel acts as a Hardware Abstraction Layer (HAL). This approach offers wide hardware support by simplifying device driver programming and providing a consistent foundation. Moreover, it enables portability as standardized system calls facilitate program compilation and execution across various platforms without substantial modifications. Additionally, the HAL promotes isolation and protection by maintaining system stability, coordinating access to shared resources, and preventing unauthorized access to other programs' resources.

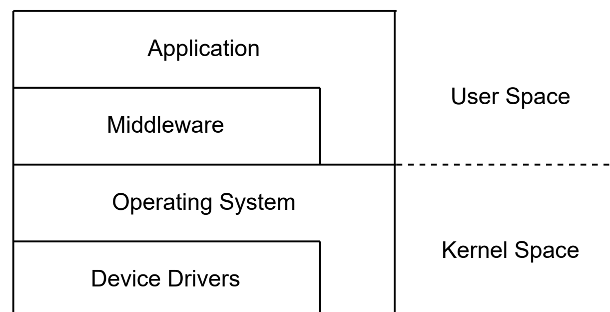

Figure 2.4: Layered scheme for GPOS architecture.

However, an essential disadvantage arises from relinquishing control over resource access. Since applications cannot directly influence the operation and prioritization of mediating layers, precise predictions concerning timing are unattainable. Additionally, indirect access generates overhead; although generally negligible on modern PCs, such overhead is undesirable in embedded systems where resource efficiency is paramount.

In contrast to GPOS, which typically integrates numerous functions and drivers directly into the kernel, RTOSes adhering to the microkernel principle relocate many of these functions to the user space. In such systems, the kernel is responsible for a limited set of tasks, including memory and process management and fundamental synchronization and communication functions. An Application Programming Interface (API) is provided to ensure the correct utilization of these kernel functions.

RTOSes generally adhere to a set of principles that differentiate them from GPOS. First, they follow the "everything is a task" approach, wherein the OS does not need to directly support a majority of devices, such as network interfaces or hard disks. Instead, tasks are responsible for managing device access and control as illustrated in Figure 2.5.

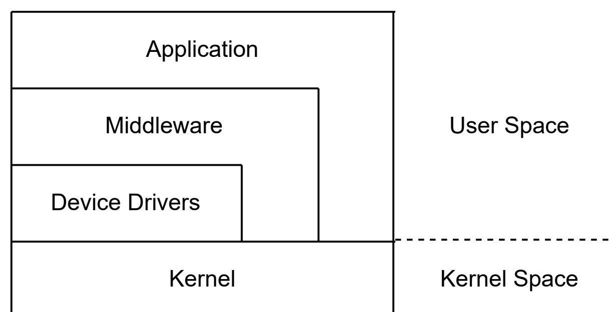

Figure 2.5: Layered scheme for RTOS architecture.

Second, RTOSes are typically designed for a specific purpose, and the software they employ is extensively tested, rendered reliable, correct and safe. Consequently, protecting mechanisms are considered optional features in such a system. Tasks perform their own I/O operations and directly access hardware. Although protection may be beneficial for safety or security reasons, it is not required for normal operations.

Lastly, RTOSes facilitate efficient management of interrupts, owing to the robust and reliable nature of their software. Unlike in GPOS, where the OS typically manages interrupts, RTOSes allow tasks to handle their own interrupts more directly. This does not mean interrupts are unrestricted, but they are managed in a way that supports the real-time, deterministic requirements of RTOS applications. This approach streamlines program sequences and minimizes overhead. However, it is worth noting that this requires stringent software design and testing to ensure that interrupt handling does not disrupt the system's real-time behavior.

Specifically, scheduling and task dispatching play a crucial role in managing system resources and meeting the timing requirements of tasks. So far we have already discussed the fundamental differences between GPOSes and RTOSes. We now dive a bit deeper into the scheduler and task dispatcher component in RTOSes.

Schedulers are responsible for determining which tasks should be executed at any given moment, with the primary goal of meeting real-time constraints and maintaining system predictability. Various scheduling algorithms can be employed in an RTOS, such as Rate-Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) scheduling [9] [10]. Given a queue of jobs, released by their corresponding tasks, the scheduler is responsible for determining which job should be executed, according to their pre-defined rules.

Task dispatchers serve as a critical component in managing task execution. The task dispatcher is responsible for managing the periodicity of tasks by waiting for

certain conditions and determining when a task should be moved to the "ready" state for execution. In the following section, we show the differences between dispatchers in GPOS and RTOSes to motivate why the efficiency of task dispatchers is of interest in this thesis.

2.3 Task Dispatchers

Task dispatchers are crucial components of modern operating systems, responsible for introducing periodicity in tasks and processes. In this chapter, we will explore task dispatchers in both GPOS and RTOSes. Specifically, we will examine the `timerqueue` in Linux, a widely-used GPOSes, and look into the task dispatcher structure in RTOSes.

2.3.1 Task Dispatchers in General Purpose Operating Systems

To be able to really look into the implementation of task dispatchers in GPOSes and be able to compare their current state-of-the-art implementation and that of the RTOS state-of-the-art implementation we have to consider that an operating system must be open source in order to check what data structures are used for their respective task dispatching implementation. In this chapter we look into what data structure Linux uses, Linux is the most popular open-source operating system in computing [11].

Since the Linux kernel is open source we can inspect their respective code base at how a comparable mechanism is implemented in a general-purpose OS. In the Linux kernel, the `timerqueue` is used to release timers based on their respective expiry time. When a timer is created in the kernel, it is added to the `timerqueue`, which is implemented as a Red-Black Tree (RBT) structure, sorted by the timer's expiration time [12]. The `timerqueue` is managed by the kernel's timer interrupt handler, which periodically checks the `timerqueue` to determine if any timers have expired. If a timer has expired, the kernel invokes the timer's callback function, which typically performs a specific action, such as waking up a sleeping process or rescheduling a task. In the following chapter, we will examine that this implementation is very similar to the generic implementation in RTOSes.

2.3.2 Task Dispatchers in Real-time Operating Systems

The task dispatcher is the part of the RTOS kernel maintaining the task periodicity by managing the tasks, making sure that the correct task is going to be executed. In Figure 2.6 a generic overview of a task dispatcher is presented. Let τ be a data

structure that contains a set of n tasks, where each task τ_i is defined by its release time r_i. In other words, $\tau = \{\tau_1(r_1), \tau_2(r_2), ..., \tau_n(r_n)\}$. t is the time counter used to maintain the periodicity of the task dispatcher. R is the ready queue, where all the ready-to-be-executed tasks are stored. The dispatcher waits on a tick from the kernel by which it enters the critical state, i.e. making sure there are no interrupting factors that might interfere with the dispatching routine. At this point, the tick counter is increased by one because a tick occurred. Now the first (lowest release time) task in the data structure is checked. If it is the case that the first task is ready to be executed (i.e. the release time has been reached or even passed) the task is added to the ready queue and removed from the data structure of the task dispatcher. After this a new task can be added to the dispatcher.

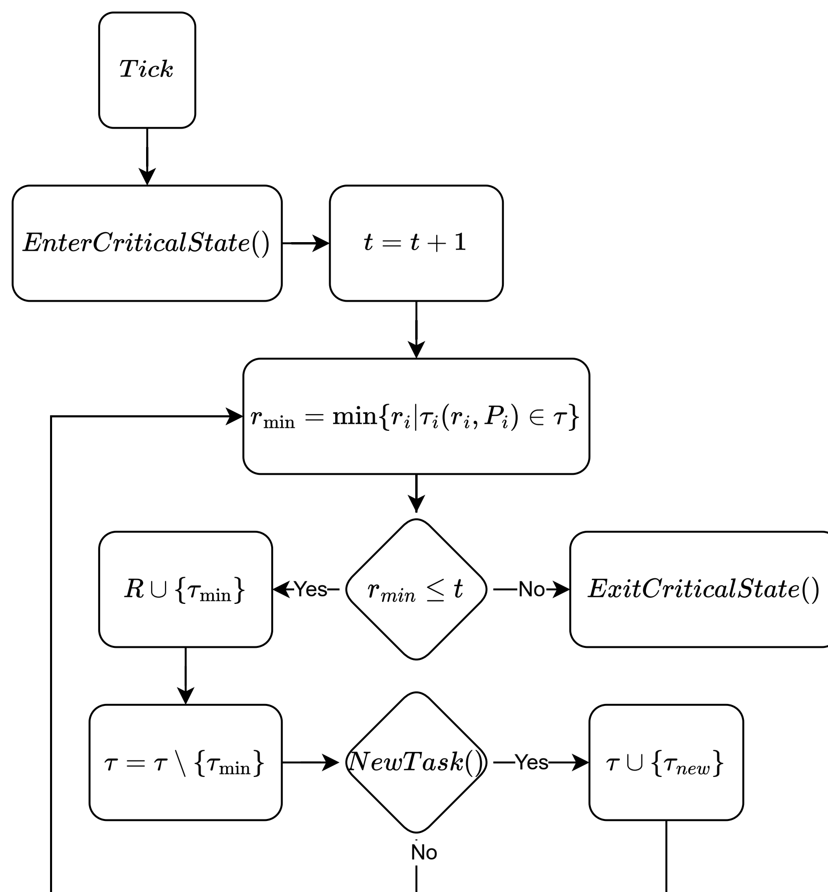

Figure 2.6: Task dispatching routine.

Note that if a task is periodic it could be the case that the periodic task is scheduled again immediately by defining a task by not only its release time r_i, but also its period p_i such that $\tau = \{\tau_1(r_1, p_1), \tau_2(r_2, p_2), ..., \tau_n(r_n, p_n)\}$. If this is the case the $NewTask()$ could be replaced by updating the release time by $r_i = r_i + p_i$ and inserting τ_i into the data structure again.

Implementations in Different RTOSes Based on the principle explained above, it is up to the designers how they want to implement the task dispatchers. We give a brief overview of the task dispatcher mechanisms deployed in two popular RTOSes used in practice, RTEMS and Zephyr [13].

In RTEMS (Real-Time Executive for Multiprocessor Systems) task management is accomplished through a combination of internal data structures, primarily utilizing list and RBT structures [14]. RTEMS manages ready tasks using lists organized by priority levels. Each list represents a distinct priority level, and tasks are enqueued within these lists according to their priority. This allows RTEMS to maintain an efficient structure for identifying the next task to be executed based on its priority.

To manage time-related functionality, including task delays, RTEMS deploys a so-called watchdog mechanism. The watchdog system consists of an RBT sorted on priority, wherein each node represents a timed event. When a task requires a delay, a watchdog is set up with the appropriate timeout and subsequently inserts it into the RBT.

As time advances, RTEMS decrements the remaining time for the watchdog in the RBT. When the remaining time reaches 0, the associated task is unblocked and moved back to the suitable ready task list (based on its priority).

The task dispatcher in Zephyr, for maintaining the periodicity, is implemented by the timeout queue. When a new task is added to the timeout queue, it is inserted into its correct position based on its expiration time. The timerqueue is based on relative time, so not on absolute time, where an event is based on the relative tick count of the previous event [15]. The insertion operation requires updating the relative tick counts of the affected nodes, which can be done into both an RBT and a doubly-linked list in Zephyr.

Upon the expiration of a task's timeout, the task is removed from the timeout queue and returned to its appropriate priority queue. This removal operation involves updating the relative time fields of the subsequent nodes in the timeout queue.

The delta list-based timeout queue also enables efficient updating of task timeouts. If a task's timeout needs to be modified, it can be removed from the timeout queue, its expiration time updated, and then reinserted into the queue.

2.4 FreeRTOS

FreeRTOS is a widely used open-source RTOS for microcontrollers and small microprocessors that is distributed freely under the MIT open-source license [16] [17]. FreeRTOS is written in C and provides an RTOS with a small footprint and low overhead [18]. FreeRTOS also provides support for multiple processors and has been ported to many different architectures [19]. The goal of this section is not to give a

detailed description of FreeRTOS with all of its insights into implementation or functionality, but rather to give a base-level understanding that may be required for the rest of the thesis. Figure 2.7 presents the structure of the software layers used for a FreeRTOS application.

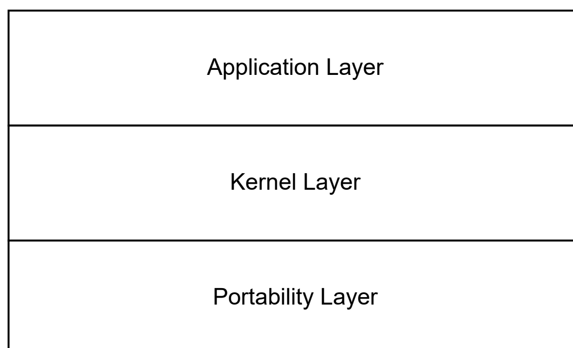

Figure 2.7: FreeRTOS software layers.

In this research, our attention is primarily concentrated on the kernel layer, striving to limit our interaction with the application and portability layer to a necessary minimum. The portability layer manages system-related functions such as memory management and other hardware-specific related matters. The application layer is where developers that make use of FreeRTOS make their application by defining and creating their respective tasks [20]. The core code of the FreeRTOS kernel is contained within just two C files, `list.c` and `task.c` [21] [22]. This kernel is the interface between hardware and the RTOS application. For developers making use of FreeRTOS, it is disencouraged to make changes to the kernel. The kernel provides everything that is required by the application code e.g. creating tasks, scheduling, timer management and many other RTOS features.

2.4.1 Task Dispatcher in FreeRTOS

The task dispatcher in FreeRTOS is primarily governed by two major components: the system tick and delay functions. The system tick is an internal timer that ticks at regular intervals. It provides a time base that allows the system to track the passage of time, and it serves as the heartbeat that drives the scheduling of tasks.

On the other hand, delay functions are used to ensure periodicity in a FreeRTOS application. Periodicity refers to the regular intervals at which tasks are scheduled for execution, an essential aspect of predictable performance in real-time systems. Delay functions help manage this by allowing tasks to be delayed for a set period, facilitating the scheduling and execution of tasks at their predetermined times.

In this section, we discuss how the task dispatcher in FreeRTOS works generically. We will not go into every detail and leave out e.g. edge cases of the standard

routine, hooks and checks of the kernel. Only information regarded as relevant with regard to the understanding of the task dispatcher within FreeRTOS is presented. After this section we have a base understanding of how the task dispatcher works in the case of FreeRTOS. The structure corresponds mostly to Subsection 2.3.2, where τ is a data structure that contains a set of n tasks, where each task τ_i is defined by its release time r_i. In other words, $\tau = \{\tau_1(r_1), \tau_2(r_2), ..., \tau_n(r_n)\}$. t is the tick counter used to maintain the periodicity of the task dispatcher. R corresponds with the ready queue, where all the ready-to-be-executed tasks are stored, such that $R = \{R_1(r_1), R_2(r_2), ..., R_n(r_n)\}$. In addition for this part of the task dispatcher in FreeRTOS we have a value MAX which holds the largest value the tick counter can have. The τ^O is a data structure (set) that holds the overflowed tasks ($\tau^O = \{\tau^O_1(r_1), \tau^O_2(r_2), ..., \tau^O_n(r_n)\}$), i.e. tasks that have a release time past the MAX value. S represents the suspended task list in FreeRTOS. This list is a repository for tasks that have been explicitly suspended, preventing them from being triggered prematurely by any timing events. The suspended task list S is defined as: $S = S_1(r_1), S_2(r_2), ..., S_n(r_n)$, where $S_i(r_i)$ symbolizes a suspended task. This suspended state guarantees that the task will not execute or be disturbed by timing events until it is deliberately resumed. This adds a layer of control in the real-time operation, ensuring tasks perform as expected without unexpected disruptions. Furthermore, u is introduced, which holds the (global) value for the next unblock time, which stores the next time a task currently in the dispatcher should be moved to the ready queue. Lastly, we have i, w and w_p which hold the value for the release time increment of the task[1], the wake-up time and the previous wake-up time respectively. For a summary of these terms and their definitions, please refer to Table 2.1.

Task delays: Starting off with the delay functions in FreeRTOS. There are two main delay functions `xTaskDelayUntil()` and `vTaskDelay()` which are the functions that can be used by periodic tasks to ensure a constant execution frequency[2]. The primary difference between `vTaskDelay()` and `xTaskDelayUntil()` is the way they handle time delays. While `vTaskDelay()` uses a relative time delay, `xTaskDelayUntil()` uses an absolute time delay. The use of absolute time in `xTaskDelayUntil()` makes it more suitable for applications that require precise timing. This is why we go through a generic presentation of `xTaskDelayUntil()`[3], in Figure 2.8, to see how the insertion end of the data structure within the task dispatcher of FreeRTOS operates.

[1]When a task is periodic this value would be the period of the task.

[2]Or by non-periodic tasks to state their next release time

[3]`xTaskDelayUntil()` and `vTaskDelay()` match for the most part, looking at Figure 2.8 the main difference from $w = w_p + i$ up until $w_p = w$, `vTaskDelay()` does not use this part since it takes into account relative time compared to the absolute time used by `xTaskDelayUntil()`.

Table 2.1: Definitions of terms used in the task dispatcher description.

Term	Definition
τ	A set of tasks, each task τ_i defined by its release time r_i
t	The tick counter, used to maintain the periodicity of the task dispatcher
R	The ready queue, where all the ready-to-be-executed tasks are stored
MAX	The largest value the tick counter can have
τ^O	A set holding the overflowed tasks, i.e., tasks with a release time past the MAX value
S	The suspended task list, holding tasks that should be suspended
u	The next unblock time, storing the next time a task should be moved to the ready queue
i	The release time increment of the task
w	The wake-up time of the task
w_p	The previous wake-up time of the task

First, the task delay is called, this initially triggers the critical state, making sure that the functionalities until the exit of the critical state can not be influenced or interrupted. Next, the wake-up time is set to the previous wake-up time combined with the requested time increment of the task. Now we check if the tick counter did not exceed the previous wake-up time. Should this condition be met, it indicates that the tick count has overflowed since the function's last call. In this scenario, a delay for the task should only occur if the wake time has also overflowed, and it surpasses the tick counter. If these conditions align, the situation can be considered as if neither the tick count nor the wake time had overflowed. If this is however not the case the previous wake time is updated and the critical section is exited and the function stops. If the tick counter did exceed the previous wake time this means that the tick counter was not overflowed and in this case we will either delay if the wake time has overflowed and/or the tick counter is less than the wake time. If not, similar to the earlier decision, the previous wake time will simply be updated and the critical section will be exited. If however the $\{w|w < w_p\} \cap \{w|w > t\}$ or $\{w|w < w_p\} \cup \{w|w > t\}$ conditions did hold true, the previous wake time will be updated but we do not exit the critical state. The scenarios in which the task should be added to the data structure (hence no immediate exit) are visually presented in Figure 2.9 to give a more clear overview. In the upper case, there is a tick count overflow, in the second case the wake time has overflowed and in the last case there is no overflow, but the tick count is less than the wake-up time, if any of these cases appear the task should not immediately exit the critical state.

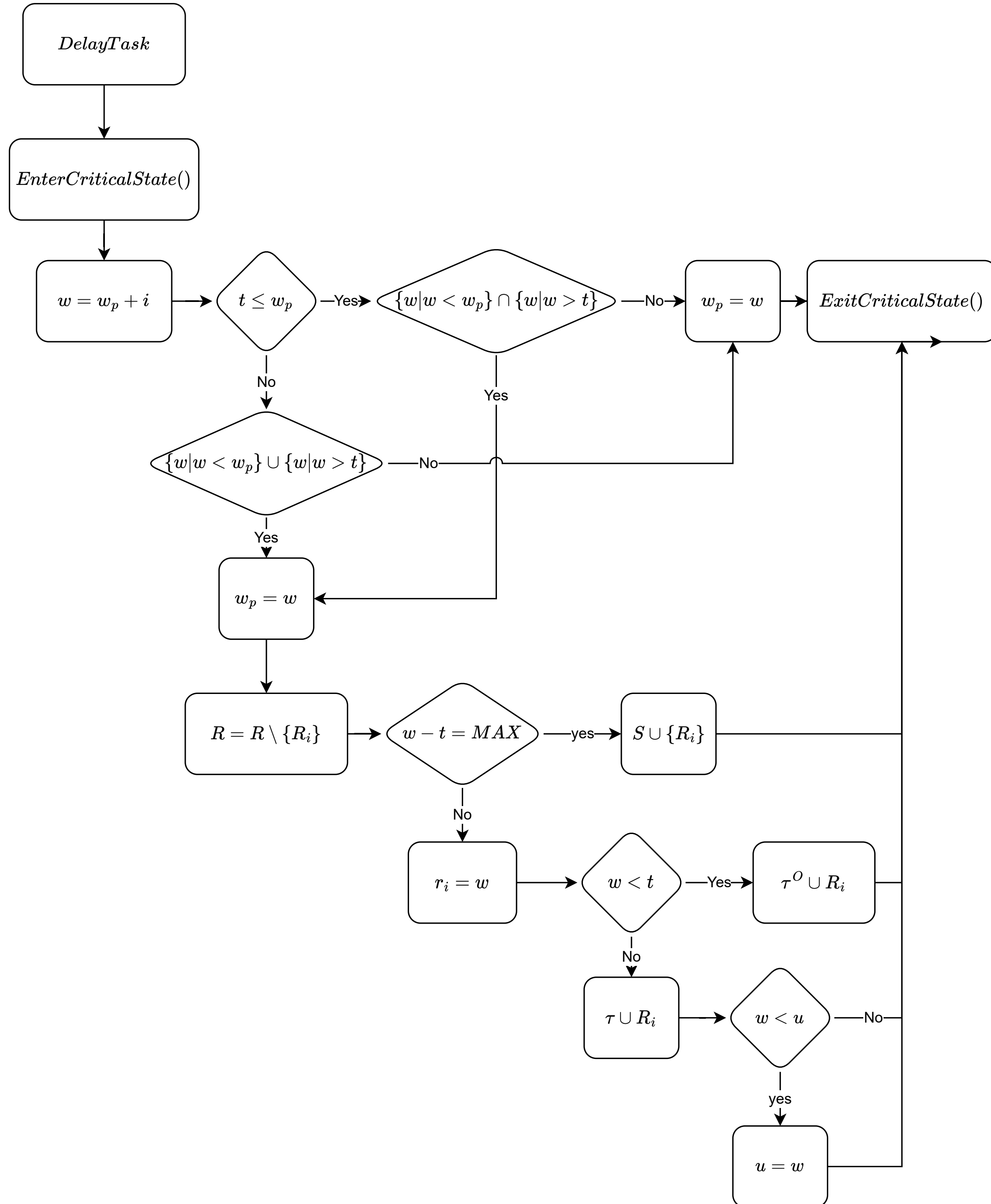

Figure 2.8: Flowchart illustrating the operation sequence for delaying tasks with regards to the task dispatcher in FreeRTOS.

Instead, we take the current task (which is in the ready queue) out of the ready queue and check if wake time without the tick count reaches the maximum value, if true, the task is added to the suspended task list and we will exit the critical section. If it does not reach the max release time of the task, R_i will be set to this wake time and

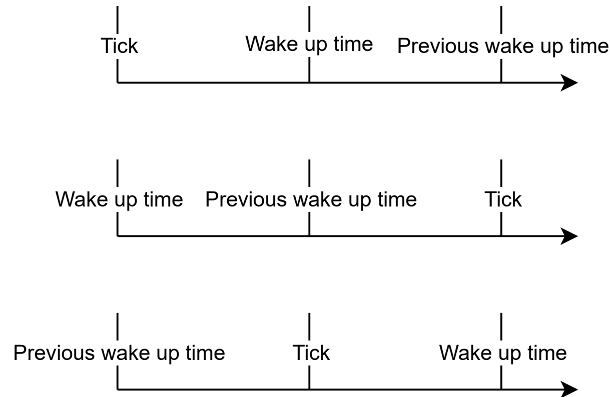

Figure 2.9: Scenarios for adding the task to the task dispatcher data structure.

we check if the wake time is smaller than the tick count. If the wake time is smaller than the tick count, it implies a condition known as an 'overflow'. To understand this concept, consider the following: In our system, the 'tick count' works like a clock, incrementing at regular intervals. It is cyclical, meaning it resets back to zero after reaching the maximum value, MAX. 'Wake time', on the other hand, represents the time when a task is scheduled to wake up and is typically based on the task's release time and its period.

When we say 'wake time has overflowed', it means that the wake time is scheduled beyond the current cycle of the tick count, i.e., it is scheduled to occur after the tick count resets and starts from zero again. In other words, if the wake time is smaller than the current tick count, it indicates that the wake time has been scheduled for the next cycle of the tick count.

To manage such tasks that have their wake times in the next cycle of the tick count, FreeRTOS uses a special structure we call the 'overflowed task dispatcher data structure' or τ^O. So, when the condition of $w < t$ is met, the task is moved into this τ^O structure, waiting for the tick count to reset and reach its wake time in the next cycle.

If the wake time is not overflowed, the task is put in the normal task dispatcher data structure. Lastly, we check if the wake time is smaller than the next unblock time, if this is the case this means the unblock time should be updated to the wake time, since this wake time occurs before the previously set unblock time. After this, we will exit the critical section and the scheduler continues.

If the tick counter did not exceed the last wake time there is a check if the wake time is lower than the last wake time and the wake time is higher than the tick counter. If this is not true this will result in an update of the previous wake time and then the exit of the critical section, resulting in the resumption of the scheduler.

However, if the counter did exceed the previous wake time, there is a check if it is true that either the wake time is lower than the previous wake time or the wake time

is higher than the tick counter. If both are false this also results in the update of the previous wake time and exit of the critical section and resumption of the scheduler. When however, the wake time is indeed lower than the previous wake time and the wake time is larger than the tick count, this means the tick count has overflowed.

System ticks: The system tick interrupt is generated by a hardware timer at a fixed frequency. When the interrupt occurs, the `xPortSysTickHandler()` function is called. The `xPortSysTickHandler()` function is an integral part of the FreeRTOS kernel and is automatically called by the system tick interrupt. It is implemented as a weak function in the FreeRTOS portability layer, which means that it can be overridden by the user if necessary. In the ESP-IDF framework, the `xPortSysTickHandler()` function is implemented in the `esp-idf/components/freertos/port.c` file, and it is configured to run on the ESP32's main CPU by default. The other core functionalities of the data structure of the task dispatcher are the retrieval and removal from the data structure. This is handled by the function `xTaskIncrementTick()`, which is primarily called by this `xPortSysTickHandler()`. In Figure 2.10 the flow of the `xTaskIncrementTick()` from the task dispatcher is presented. In the `xTaskIncrementTick()` there is also handling done with regards to the ready queue in relation to preemption and time slicing, we do however not regard this as part of the task dispatcher and as so will not include this in the description.

Starting off at the moment the tick comes in, the critical state is entered, i.e. making sure that the routine is not interrupted or influenced during execution, followed by an increment of the tick counter. Now there is a check if the tick counter is at its highest value, if this is the case the data structure (set) of overflowed tasks will become the normal task dispatcher data structure of tasks and vice versa. Next up we check if the tick counter is larger than or equal to the next unblock time. If this is not the case, there is nothing left to do and we leave the critical state. If it is the case we check if the data structure of tasks is empty, if this is the case the next unblock time will be set to the maximum of the tick counter and we exit the critical state. When the data structure of tasks is not empty we find the task with the lowest release time and check if the lowest release time is larger than the tick counter. When this is true the next unblock time will be set to the lowest release time of the data structure. When it is false the task is removed from the data structure and added to the ready queue and we go back to check if the task dispatcher data structure is empty again.

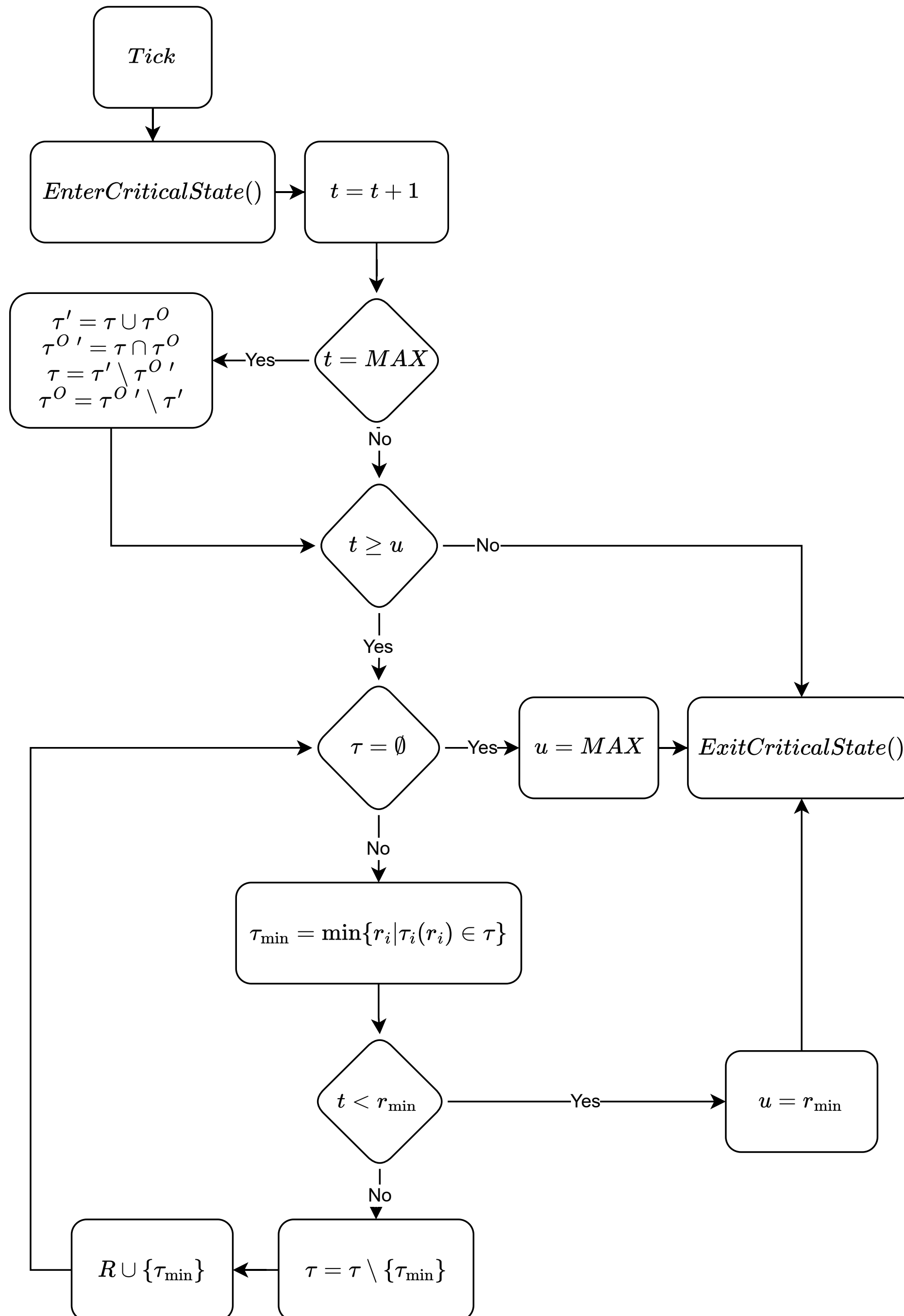

Figure 2.10: Flowchart illustrating the operation sequence for tick incrementation with regards to the task dispatcher in FreeRTOS.

Chapter 3

Real-world Measurement Setup

The accurate measurement of the performance of a system is crucial in any research study. In this chapter, we present the real-world measurement setup that is utilized in our research, which involves the software, hardware and interfacing used to collect the data for this research. Of which an overview is presented in Figure 3.1.

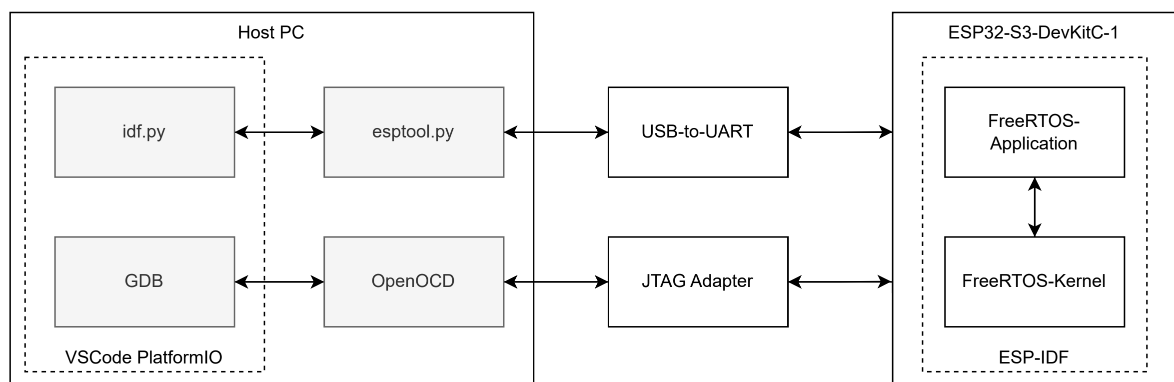

Figure 3.1: Real-world Measurement Setup.

First, we will describe the ESP-IDF FreeRTOS kernel used in our research, which is a FreeRTOS kernel inside a development framework that provides a platform for software development specifically designed for the ESP32 series.

Next, we will introduce the embedded device used in our research, an ESP32 microcontroller. We will describe the PlatformIO structure used to configure and manage this device and the possibility of easily changing devices or other configurations. In PlatformIO we can make a streamlined development process simplifying the configuration, managing of devices and toolchains for (future) development.

To debug the system and get more insight into the measurement results we utilize GDB and OpenOCD. We will provide an overview of how these can be used to enable reliable debugging and measuring of the embedded device.

Finally, we will explain the measurement process, which utilizes two major methods, one with the use of the debugging tool set and one with run-time measurements.

Overall, this chapter aims to provide a comprehensive understanding of the real-world measurement setup utilized in our research. By detailing the software, hardware, and interface components used to collect data, we aim to ensure the accuracy and reliability of our research findings.

3.1 ESP-IDF FreeRTOS

For FreeRTOS, version V10.4.3 is used in the ESP-IDF (Espressif IoT Development Framework) version 4.4.1. ESP-IDF is a set of software tools and libraries developed by Espressif Systems for building applications on their System-on-Chip (SoC) devices, such as the ESP32 which is used in this research and described briefly in Section 3.2 [23]. This set of tools and libraries includes an RTOS which is a modified implementation of FreeRTOS. The FreeRTOS used in this research is a version modified for use with the ESP-IDF, which we will simply refer to as 'FreeRTOS' throughout this thesis. This version extends the capabilities of the original FreeRTOS to support multi-core functionalities on various ESP targets [24]. While the original FreeRTOS is a single-core RTOS, our focus will be on this multi-core variant provided by ESP-IDF.

This FreeRTOS kernel can be found after installing ESP-IDF and PlatformIO (see Section 3.2) in
`.platformio\packages\framework-espidf\components\freertos`. With regard to our research, there are no relevant differences between the FreeRTOS we used and the original FreeRTOS other than the Symmetric Multiprocessing (SMP) mentioned before.

Two essential tools play a significant role in the development of the FreeRTOS kernel modifications and application in ESP-IDF, which streamline the process heavily. These are the `idf.py` and `esptool.py` presented in Figure 3.1. We use `idf.py` as a development tool that automates development tasks such as building, flashing and monitoring target devices. We use `esptool.py` for custom firmware and bootloader flashing, as well as reading and writing the device's flash memory.

3.2 Embedded Device and PlatformIO

In this research, we have chosen the ESP32 as the primary hardware platform for measuring computation overhead within the task dispatcher of FreeRTOS.

The ESP32 series are low-cost, low-power microcontrollers that are well-suited for embedded applications. One of the primary reasons for selecting the ESP32 platform in our measurement setup is its compatibility with the ESP-IDF with its support for FreeRTOS, which is described in Section 3.1.

We have tested the setup with multiple ESP32 devices (ESP32-C3-DevKitC-02, ESP-WROVER-KIT-VE and the ESP32-S3-DevKitC-1) to see if the established toolchain and implemented task dispatcher implementations would also work on other ESP32 devices with only changing minor board dependent configuration settings, which was successful.

The embedded device utilized for the measurements discussed in Chapter 5 is the ESP32-S3-DevKitC-1. This development board, designed by Espressif, is equipped with a dual-core 32-bit Xtensa LX7 microprocessor capable of reaching clock speeds of up to 240 MHz. A Multitude of features such as Wi-Fi (IEEE 802.11 b/g/n-compliant) and Bluetooth (BLE, Bluetooth 5). For our research, two features on this device that make it easier for us to do measurements are the USB-to-UART bridge and the JTAG controller. As presented in Figure 3.1 these functionalities are used in our setup.

PlatFormIO is used in this research in order to make our implementations more easily reusable for follow-up research, as well as to simplify/automate the toolchain. PlatformIO is an open-source ecosystem with cross-platform build system, library manager, and full support for over 1400 development boards [25]. This includes very popular platforms such as Arduino, ESP8266, ESP32, STM32 and Raspberry Pi. This makes it very easy to switch boards or even platforms during a project. This decision was made after originally ESP-IDF was used in the extension provided by Espressif, which made it more laborious to switch even between two different ESP32 devices. For this research, PlatformIO is set up such that a large multitude of ESP devices can be used to run the same code without having to adapt a large part of the project manually. The `platformio.ini` file is used to set flags for compilation of the code for the given hardware. An example is `board = esp32-s3-devkitc-1`. In the given example the board that is defined is the `esp32-s3-devkitc-1`, which means that all the required settings for this specific board are set correctly. This option makes it easy to switch boards without changing the implementation code. Many other flags can be found and configured by consulting the PlatformIO documentation[1].

[1] e.g. in https://docs.platformio.org/en/latest/projectconf/index.html

3.3 GDB & OpenOCD

The debugging method described in this chapter is designed to provide reliable results. We give some insights into the use of GDB and OpenOCD to ensure proper debugging and instrumentation of the system, which enables us an extra manner of collecting data for our measurements to ensure accuracy.

OpenOCD, working in conjunction with GDB, serves as our debugging tool of choice for this research. OpenOCD (Open On-Chip Debugger) is an open-source JTAG debugger, known for its compatibility with a variety of devices, and can be controlled from GDB (GNU debugger) [26]. GDB, on the other hand, is a versatile tool that allows developers to inspect their program during execution, providing valuable insights that aid in debugging and optimization processes [27].

The combination of OpenOCD and GDB enables us to read data from the MCU and debug our application with minimal impact on its run-time behavior. This approach presents a significant advantage over alternatives such as using print statements, which can lead to considerable alterations in kernel behavior.

To facilitate the understanding of the program during execution, GDB employs four main elements, each serving a distinct purpose in the debugging process [27]:

- Program startup can be controlled and customized by specifying various execution parameters.
- Execution of the program can be paused, resumed, or terminated on specified conditions or events.
- Developers can inspect the program's internal state, such as memory content, variable values and call stack when the execution is paused.
- Changes can be made to the program's state, such as variable values or memory content, while the program is paused to observe the effect of those changes on program behavior.

JTAG interface, OpenOCD and GDB can be interconnected to enable debugging for a variety of ESPs including the ESP32-S3-DevKit-1 which is used for the measurements of this research [28].

The interfacing of the JTAG interface can be different per device, this should be checked per device before usage. However due to the setup of PlatformIO during this research, changing between these JTAG interfaces can easily be adjusted as was the case for changing to another board, described in Section 3.2. This is also done in the `platformio.ini` file, under `debug_tool`. E.g. some ESP devices have an onboard debugger that can be used without the use of an external debugger, in this

case, the debug_tool can be set to debug_tool = esp-builtin. Other ESPs that do not have an onboard debugger or if it is preferred to use an external debugger, can, in this case, use a debugger such as the ESP-PROG, by adjusting the setting debug_tool to debug_tool = esp-prog. A wide variety of external JTAG debuggers can be used for debugging ESP devices[2].

In this research the esp-prog and esp-builtin were tested and used. Figure 3.2 showcases three distinct setups, two of which (middle and right) demonstrate the interconnect with a built-in JTAG debugger, where one (middle) utilizes a DIY solution, a USB-cable is cut open, and the D- of the MCU is connected to the D- of the USB-cable, D+ to D+, 5V to V_BUS, and Ground to Ground. The third setup illustrates the interconnect with an ESP-PROG (left). Their interconnect can be found in the documentation by Espressif on the page of the respective JTAG debugger or ESP board.

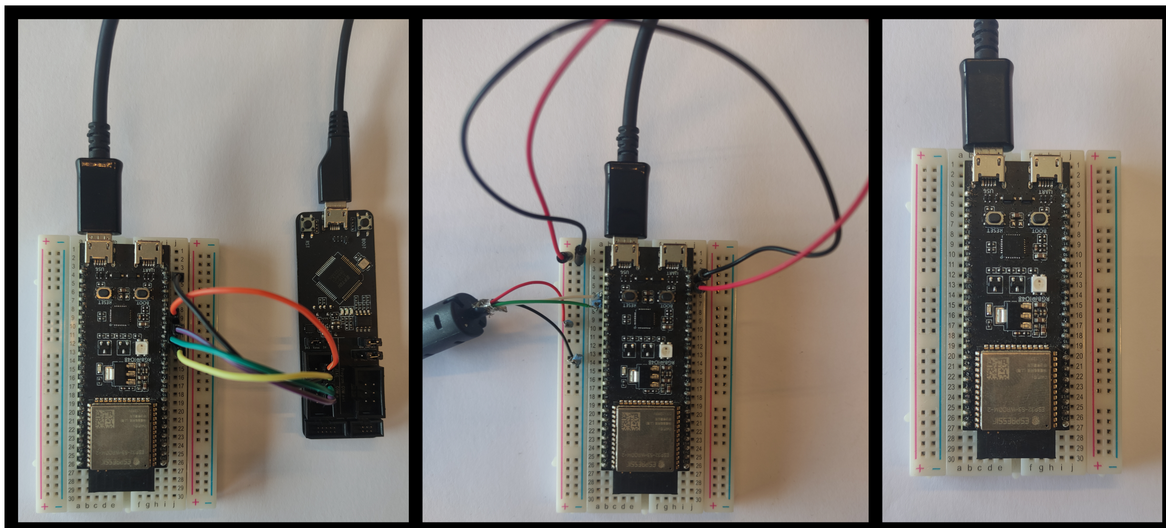

Figure 3.2: Real-world Interconnects for Debugging: Comparison of ESP-PROG (left), Built-in (middle and right) Options.

We have 3 debug options defined in the launch.json, which are the PIO Debug, PIO Debug (skip Pre-Debug) and the PIO Debug (without uploading) options. The first two are identical, the only difference is the presentation of build results, where the main difference is that the first option outputs the results in the VSCode Debug Console and the second option in the VSCode built-in Terminal. The third option however, is somewhat different, this option allows for the debugger to be run without re-uploading the code (i.e. no compiling and building of the code is required).

[2]https://docs.platformio.org/en/latest/plus/debugging.html

The debugging options described in this section make it possible to see what is happening ”inside” of the microcontroller. We can e.g. read out registers, set breakpoints in the code run-time where we want to check behavior and keep track of variables and registers.

In Figure 3.3 we showcase the most frequently utilized debugging options during this research. On the top right, the defined debug options mentioned earlier can be seen. Next, we discuss some useful information that can be extracted from the microcontroller using this debugging approach.

The `VARIABLES` includes `Local`, `Global` and `Static` variables which can all be expanded to see all the variables available to us. The `WATCH` allows adding elements to be monitored during run-time. The `BREAKPOINTS` displays the breakpoints placed in the code at specific points, where the program execution is paused to examine the system's state. (You can also place conditional breakpoints, e.g. only pause when a variable is bigger than x or if the function is executed y times.) Furthermore, we have the `REGISTERS`. Here we can check what the current values are in all the registers of the device. Currently, at the top, we see the `ccount` register, which we use to track the computation overhead of the device as we will present in Section 3.4.

Finally, to analyze the system's behavior, variables can be modified in the debugger to observe the resulting response from the system.

3.4 Measurements

In our research, we have identified the CPU cycle counter—a tool that measures the number of processor cycles elapsed—as an effective mechanism for quantifying the overhead of the task dispatcher. In this section, we present some of the advantages of using the CPU cycle counter for this purpose as well as the methods used to obtain the CPU cycle counter overhead.

Firstly, the CPU cycle counter provides a high-resolution measurement of time, with an accuracy down to a single clock cycle. This level of resolution was considered very important for measuring the overhead of software that is measured based on computation performance. By using the CPU cycle counter, we can obtain a detailed understanding of the impact of our implementations on the overhead of the task dispatcher implementations.

Secondly, the CPU cycle counter has a very low overhead, as it is a hardware-based mechanism for measuring time. There is little to none additional processing time, ensuring that the measurement mechanism itself does not add significant overhead to the system. This allows us to isolate the impact of our implementations of the computation overhead.

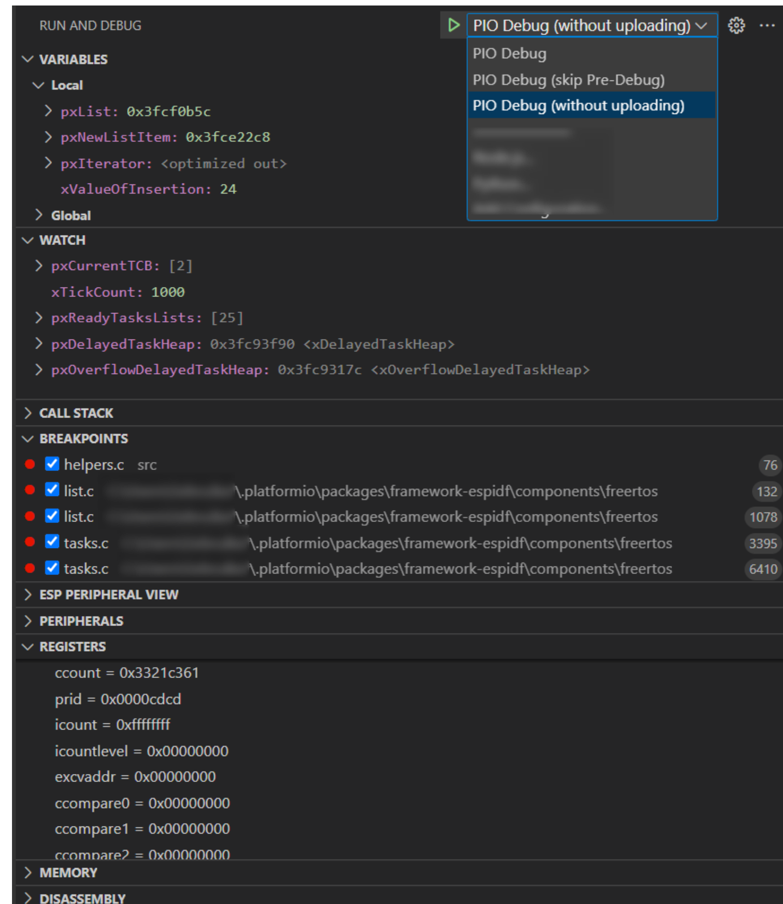

Figure 3.3: Debugging Options in GDB Visual Studio Code: An Overview of Key Features and Functionality.

Thirdly, the CPU cycle counter is independent of external factors that may interfere with other time measurement mechanisms, such as changes in the clock frequency. By measuring the clock cycle counter the interference of external factors is ruled out.

Finally, the CPU cycle counter provides a consistent measure of time across different systems, as it is based on the internal clock of the processor. This makes it a useful tool for comparing the performance and overhead of devices or dispatcher implementations, providing a standard measurement mechanism that can facilitate a fair comparison.

The use of the CPU cycle counter for measuring system overhead is not necessarily considered the best way in all circumstances. There are other methods and techniques for measuring system overhead that may be more appropriate depending on the specific context and requirements.

For example, if the goal is to measure the overhead of I/O operations or other system calls, then using the CPU cycle counter may not provide a complete picture, as it does not account for the time spent waiting for I/O operations to complete.

In such cases, techniques such as system call tracing or profiling may be more appropriate.

Additionally, the use of the CPU cycle counter for measuring system overhead can be impacted by factors such as power management and CPU frequency scaling, which can affect the number of clock cycles that have elapsed. In such cases, alternative methods that take these factors into account may be more appropriate.

In summary, the CPU cycle counter offers numerous advantages for measuring the overhead of a software system. Its high resolution, low overhead, independence from external factors, and consistency across systems make it an effective tool for providing accurate and reliable measurements of system overhead, enabling us to gain a good understanding of the performance of the implemented task dispatchers.

The CPU cycle counter overhead measurements were conducted by monitoring the run-time clock cycles based on the `ccount` register of the ESP32-S3-DevKitC-1. For the run-time measurements the average, best and worst-case scenarios were traced along a span of several tens of thousands of task executions up to several hundreds of thousands of task executions, for every single measured size for every task set, to ensure reliable results. Additionally, the debugger tool was deployed to track the `ccount` register in order to investigate and trigger specific cases and thoroughly examine function execution. The utilization of these techniques allowed us to give a good analysis of the CPU cycle counter overhead and thereby providing valuable insights into the characteristics of the task dispatcher.

For the measurements of the setup, we have disabled most compiler optimizations (so-called optimization level -O0) to prevent the compiler to optimize the performance. Enabling compiler optimization can significantly reduce the overhead of a system, making it faster and more efficient. However, the use of compiler optimizations will make it more difficult to accurately measure the overhead of a system, as the optimized code may be significantly different from the original code. By employing this approach, the measurement outcomes become less reliant on the compiler or CPU architecture, resulting in greater reproducibility and accuracy of the results.

Chapter 4

Methodology

In this chapter, we provide a comprehensive account of the research strategies, tools, and techniques employed to explore the effectiveness of task dispatcher implementations in embedded systems using FreeRTOS. The section is structured to present a coherent overview of our approach, with a focus on the data structures used, task dispatcher implementations, testing application, and limitations encountered during real-device testing.

Initially, we review various data structures found in literature that have been proposed for managing task dispatching. We examine their suitability for our research objectives, weighing the trade-offs between complexity, efficiency, and ease of implementation. This evaluation enables us to identify the most promising options for implementation in our study.

Subsequently, we discuss our implementation of the selected task dispatchers in FreeRTOS, providing an in-depth account of the design choices and some of the modifications made to the standard FreeRTOS kernel. We explain the rationale behind our decisions, highlighting the factors that influenced our choices and how they align with the goals of our research.

Additionally, we provide a concise description of the test application we developed for evaluating the implemented task dispatchers. We outline the key features of the application, its structure, and its functionality, including the metrics we used to assess the performance of the task dispatchers.

Lastly, we briefly address the limitations posed by memory constraints in real-device testing. Specifically, we discuss the challenges arising from the shortage of memory on the embedded device, which may impact the extent of our research and the scalability of our solutions.

4.1 Task Dispatching Data Structures

Finding useful data structures for task dispatchers in FreeRTOS is dependent on different factors. Since there are in reality three major functions required as can be seen in Section 2.4, task insertion, first task retrieval and first task removal. Comparing the number of executions for these functions is evident for insertion and removal. They must always become equal to ensure that tasks neither disappear nor spawn nondefined tasks ($n_{RemovalCalls} \approx n_{InsertionCalls}$), but the number of retrieval calls is not always equal to these. Looking at the implementation of the incrementation of ticks in FreeRTOS described in Figure 2.10 it can be seen that the execution of the retrieval of tasks will always be equal or higher than the removal (and thus also the insertion) of tasks.

While the number of task insertion and removal operations must be equal to maintain the integrity of the system, the number of retrieval operations can be equal to or greater than these. The specific ratio depends heavily on the nature of the task set, particularly the overlap in task release times. If task release times overlap significantly, i.e., if many tasks have the same period or multiples of each other's periods (e.g. for 1ms and 2ms period tasks, the 1ms task will have the same release time as the 2ms task 50% of the time), the number of retrieval operations will become closer to the number of removal operations.

However, if task release times do not overlap, this can result in a ratio that approaches two retrieval operations for each removal operation. This occurs because the system retrieves the next task, removes it, then retrieves the next task again, which is not yet ready to be removed.

The only scenario where the ratio of retrieval operations to removal operations would exceed 2 is if the global unblock time does not match the actual wake-up time of the next task, as can be derived from Figure 2.10. Thus, the ratio of retrieval calls to removal calls generally falls within the range of $1 \cdot n_{RemovalCalls} \leq n_{RetrievalCalls} \leq 2 \cdot n_{RemovalCalls}$, with the potential to exceed the upper limit in the case of a mismatch between the global unblock time and the actual task wake-up times. This ratio is a key factor influencing the overhead of the task dispatcher and should be taken into account when assessing performance.

In task dispatching related literature a few data structures are described to be viable for an RTOS [6] [29] [5]. In this section, we walk through some of these data structures with a short description and some advantages and disadvantages for these data structures.

Originally, most of the data structures mentioned in this subsection were not specifically designed as task dispatcher data structures. However, we describe them in this context to demonstrate how they can be adapted for this purpose.

Sorted List

The list structure is a basic data structure often utilized in computer science, as currently done in FreeRTOS. The structure consists of nodes that are interconnected via pointers. A sorted list, for example, can serve as the basis for a task dispatcher, where the list is ordered based on the relative release time. This way, retrieving and removing the first task to be released would have a time complexity of $O(1)$. However, inserting in a linked list has a time complexity of $O(n)$, as we need to traverse through the list to place the item in the correct location within the linked list.

Understanding the implications of time complexity, as we explore further in Subsection 4.1.1, is crucial in assessing the efficiency of an algorithm or operation. In this context, the disparity between the highly efficient retrieval operation ($O(1)$ complexity) and the less efficient insertion operation ($O(n)$ complexity) prompts a search for alternative data structures. These alternatives aim to strike a balance between improved time complexity and the simplicity of code that can be executed faster, thereby reducing overhead.

Bucket of Ignorance

The Bucket of Ignorance (BoI) is a data structure proposed by Ebbrecht et al., which is designed to optimize computational efficiency in task scheduling [29]. The underlying assumption of this data structure is that inserting an item with a high release time into a list structure potentially squanders computational time. This inefficiency arises because items with a long period (later release times) are added to the end of the list, and each insertion requires traversal through these items to place tasks with shorter periods (earlier release times) appropriately.

Ebbrecht et al., therefore, designed a data structure that keeps ”soon to be released” items in a sorted list and items are not due for soon release in an unsorted list [29]. This should result in the insertion having a relatively low overhead, on average, but can increase overhead at certain points in time when the two lists (ordered and unordered) have to be merged.

The goal of this approach is to get a lower average computation time overhead on the insertion of the data structure, compared to the list data structure, while maintaining the efficiency of removing and retrieving the head of the list, which remains $O(1)$. It is described that in the number of comparisons required by the BoI versus the ordered list structure, the BoI should outperform the list structure at around 32 tasks with log-uniform distribution with two orders of magnitude over the interval [1, 100] [29] [30].

Timing Wheel

The Timing Wheel is a data structure described by George Varghese and Anthony Lauck [5]. A Timing Wheel can conceptually be seen as a clock, where the current time stamp points to a task or a multitude of tasks. The paper describes three main types of timing wheels, normal, hashed and hierarchical timing wheels.

A normal timing wheel is a data structure that consists of an array of timer slots, each slot representing a unit of time, e.g. seconds or milliseconds. These slots are arranged in a circular fashion, and the current time is indicated by a pointer that points to the current time slot (hence the clock-like concept). When a release time is set, it is added to the slot that corresponds with this release time. When the timer advances to a successive slot, all the tasks in the current slot are examined, and those that have expired are removed from the timing wheel and added to the ready queue accordingly.

However, it is worth noting that the performance of the timing wheel can degrade if the number of active timers becomes very large, or if the timer intervals become very small. In such cases, more complex data structures, such as the hashed timing wheel, may be more appropriate.

The hashed timing wheel is a variant of the timing wheel that uses a hash function to directly map timers to slots in the wheel. This approach eliminates the need to scan all the timers in a slot when the wheel advances, making it more efficient in terms of time complexity than the standard timing wheel for systems with a large number of timers. However, this efficiency comes with a trade-off: the hash values of each timer need to be stored, which increases memory usage. This can introduce a significant memory overhead, especially when the number of timers is large.

The hashed timing wheel consists of an array of buckets, where each bucket contains a linked list of timers that hash to that bucket. When a timer is added, its hash value is used to determine the bucket it belongs to, and it is added to the linked list in that bucket.

When the wheel advances, only the timers in the buckets that correspond to the current slot are examined. This reduces the number of timers that need to be processed and improves the efficiency of the timer facility.

The main advantage of the hashed timing wheel is that it is more efficient than the standard timing wheel for systems with a large number of timers. The hashed timing wheel is however more complex than the standard timing wheel and requires a hash function to map timers to buckets. The hashed timing wheel may have higher memory overhead than the standard timing wheel, depending on the number of buckets used.

The hierarchical timing wheel is another variation introduced in the paper by George Varghese and Anthony Lauck [5]. In a hierarchical timing wheel, multiple

timing wheels are arranged in a hierarchical structure. Each wheel covers a range of time intervals and has a finer granularity than the one before it. E.g. the "highest" level wheel may have a granularity in terms of hours and the "lowest" level wheel may have a granularity in terms of milliseconds.

Adding a timer to the hierarchical timing wheel, it is placed in the correct granularity wheel based on its release time. When a timing wheel "ticks" all of the finer granularity wheels are checked (recursively until the finest granularity wheel).

A key advantage of the hierarchical timing wheel is the possibility to handle a large range of release times with fine granularity. However, the hierarchical timing wheel is a more complex timing wheel and may require more computation overhead due to its additional bookkeeping in order to maintain the recursive processing.

Differences between standard timing wheel, hashed timing wheel and hierarchical timing wheel mainly comes down to the task set. With a small range of release times, the standard timing wheel might be more beneficial due to its efficiency and simplicity. The hashed timing wheel might be preferred if the distribution of release times is more uneven and the number of release times is rather small. The hierarchical timing wheel is more flexible and should be more suited for a large range and a large amount of release times, but is more complex and might cause more computation overhead.

Binary Search Tree

A Binary Search Tree (BST) is a widely used data structure and is often described in literature [31] [32]. The BST can be used as a data structure for the task dispatcher to manage a collection of tasks based on their respective release time. A BST exists out of nodes, where a node N contains a $left$ child, $right$ child, parent and key (which is based on the release time of the task), such that $N(right, left, parent, key)$. By design, the left node should always point to a node that has a lower key value than the current node and the $right$ child should always point to a node that has a higher key value. It is important to note that only nodes at the bottom (so-called leaves) of the BST can have both $NULL$ children and only the root node can have a $NULL$ parent.

To insert a task into a BST, the key is checked and the node will move down the tree recursively (according to the previously described rules) until the appropriate location for this task is found. Once the location is identified the task is added as a new leaf node.

To retrieve the task with the lowest release time, you simply traverse through the tree by keep going left on every node until a node is reached that has a $left$ child that is $NULL$, this is the node with the minimal key-value / lowest release time.

Removing the lowest release time will start off with the same process as the retrieval of the lowest release time task. Now the node can be removed. If the removed node still had a right subtree, the $parent$ should be updated, where the right subtree of the removed node is now a child of the removed node's $parent$. If this is not the case, the $left$ child of the $parent$ node of the removed node should be set to $NULL$. This is a useful feature when only the lowest value should be removed and this has reduced overhead compared to removing random nodes in the BST.

One advantage of using a BST as the base of a task dispatcher is that the task insertion and retrieval of the first task operations can be performed in $O(log\ n)$ time complexity on average, where n is the number of tasks in the tree. This is because the BST can maintain a balanced tree structure, which allows for efficient search and insertion operations. Additionally, the removal of the first task can also be performed in $O(log\ n)$ time complexity in this average case.

However, one disadvantage of using a BST-based task dispatcher, is that the worst-case time complexity for the task insertion, first task retrieval and first task removal is $O(n)$, where n is the number of tasks in the tree. This can occur if the BST is heavily skewed (unbalanced), which can happen if tasks are added to the tree in sorted order. To address this issue, self-balancing binary search trees, such as the Red-Black Tree (RBT), can be used to maintain a balanced tree structure and ensure efficient operations in both the average and worst cases.

Red-black Tree

A Red-Black Tree (RBT) is a self-balancing BST that guarantees a balanced tree height as is given in Equation 4.1, where n is the number of nodes in the tree and h is the height of the tree [31].

$$\frac{h}{2} \leq log_2(n+1) \leq h \tag{4.1}$$

To ensure that the tree is balanced and the height of the tree is at most $2 \cdot log_2(n+1)$, the RBT must satisfy the following set of properties [31] [33]:

- Every node is either red or black.
- The root node is always black.
- Every leaf node (NULL node) is black.
- If a node is red, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

When these properties are fulfilled, the BST is an RBT. To keep these properties they must be held into account and adjusted accordingly in the insertion and removal functionalities of the task dispatcher. Some papers however also disregard or change the second and/or fourth given property [34] [32] [35].

Inserting an item into an RBT can be implemented in a similar fashion to the method presented for the BST. However, this insertion can cause a violation of one of the properties, which is where a so-called ”fixup” (or ”rebalance”) function comes into play.

If you only have to remove the earliest release time task, then you do not need to implement a fixup function for regular remove operations. The reason is that removing the earliest release time task only removes the smallest node from the tree, which is always a leaf node or a node with one child. Therefore, removing the lowest release time node can only violate the same properties as the task insertion function can and there is no need to implement additional fixup functions (which would be the case if a random item of the RBT would be removed). The fixup function of an RBT can increase the overhead due to increased instructions. However, it may also result in a decrease because of the lower time complexity. This would most likely be a consideration depending on specific task sets.

2-3-4 Tree

A 2-3-4 tree (sometimes called a (2,4)-tree or a 2-4 tree) is a data structure described in literature [36] [37] [34]. The 2-3-4 Tree can be used as the base of the task dispatcher to manage a collection of tasks based on their respective release time.

The 2-3-4 tree is a self-balancing search tree where a node can have two, three or four children. Each node stores one or two keys and the children are arranged in ascending order. The insertion function will look for the leaf where the task should be inserted based on the release time. If this leaf is full, the node is split into two new leaf nodes and the middle release time is moved up to the parent. If the parent node is now also full, this one will split the same way and this process is repeated recursively until the root is reached.

To retrieve the task with the earliest release time we traverse the leftmost path of the tree and there we find the earliest release time task.

If we want to remove this task we also traverse to this minimal value and remove the task from this node. If the node is now less than half full, it is either merged with one of the children nodes of its parent or one of these nodes is split to maintain the balance. This process is also repeated recursively until the root is reached.

An advantage of using a 2-3-4 tree for a task dispatcher is its efficient management of tasks with varying release times. The self-balancing nature of the tree ensures that the time complexity of operations remains relatively stable even as the tree grows in size. However, a disadvantage is that the tree requires more memory overhead compared to simpler data structures like a linked list or a binary search tree.

Heap

When we are talking about a "Heap" in this research without extra annotations we are talking about a "Min Binary Heap". Where "Min" means that the root of the heap will always hold the lowest value and "Binary" means that each node has 2 children. This means that in the context of the task dispatcher, the order of the tasks is based on their release time, where the first release time will always be the root of the heap. A Heap is often used in the field of computer science and is an efficient implementation for a priority queue [38] [31]. In an Array-based Heap, the parent of the node in position i is in position $\lceil \frac{i}{2} \rceil$, and the other way around children of the node will always be in position $2i$ and $2i + 1$.

To insert an item into a Heap structure we simply add the node as a new leaf and see if it violates the property that the parent should always be smaller than the children nodes. If this is the case we swap the violating child node with its parent recursively until the property holds again for the entire Heap. Retrieving the first to-release task is simply examining the root node of the Heap. Removing this node will mean that there will be a vacuum and this should be filled and the Heap property should be restored again. This step can be done in several ways, one of these ways is by putting a leaf node into the empty root position and recursively swapping it down until the Heap property is restored again. An advantage of the Heap structure in a task dispatcher is its efficiency, the insertion removal and retrieval can all be done efficiently. The earliest release time is always the root of the tree which makes it so that getting the task with the first release time can be done very quickly. A disadvantage of using a Heap structure is that it can require additional memory.

4.1.1 Time Complexities

In this subsection, we give an overview of the time complexities, average and worst-case of the aforementioned data structures with regards to their task insertion, first task retrieval and first task removal functionalities. The idea of complexity theory is classifying problems to their intrinsic computational difficulty. For this we use the Big O notation, the Big O notation is a mathematical notation used to describe the

growth rate of a function. More specifically, the upper bound of its growth rate. It is widely used in computer science, particularly in analyzing the performance of algorithms. Given two functions, $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there exist positive constants k and n_0 such that $f(n) \leq k \cdot g(n)$ for all $n \geq n_0$.

In this context, $f(n)$ represents the runtime or "complexity" of an algorithm, and $g(n)$ is a simpler function that captures the dominant growth behavior of $f(n)$. The constant k and n_0 help us ensure that the inequality holds for sufficiently large input sizes (n).

In other words, $f(n)$ is said to be "order of $g(n)$" or "Big O of $g(n)$" if, beyond a certain input size the growth rate of $f(n)$ is always bounded by a constant multiple of $g(n)$. This allows us to compare the efficiency of different algorithms by focusing on their dominant growth behavior as the input size increases.

The Big O notation serves as a useful tool for comparing algorithmic efficiency by characterizing their asymptotic growth rates. As a result, distinct functions sharing the same growth rate can be represented with identical O notation, simplifying the comparison process.

Functions can be described by means of this Big O notation by a small list of bounds. For the described data structures three of these complexities are relevant namely [39]:

- $O(1)$, Constant: there exists a constant k such that $f(n) \leq k$.
- $O(log\ n)$ Logarithmic: there exists a constant k such that $f(n) \leq k \cdot log\ n$
- $O(n)$, Linear: there exists a constant k such that $f(n) \leq k \cdot n$

Where f is the algorithm and n is the input size.

Constant time complexity, often denoted as $O(1)$, describes an algorithm whose execution time remains constant regardless of the size of the input, i.e. the algorithm is independent of the input size. This makes an algorithm with constant complexity highly efficient for large inputs, since the execution time does not increase as the input size grows.

Logarithmic time complexity, often denoted as $O(log\ n)$, describes an algorithm whose execution time increases logarithmically with the size of the input, i.e. as the time to execute an algorithm increases at a much slower rate compared to the input size. Algorithms with logarithmic complexity are algorithms that divide the task at hand into smaller sub-tasks.

Linear time complexity, often denoted as $O(n)$, describes an algorithm whose execution time increases linearly with the size of the input, i.e. the execution time of the function grows proportionally to the input size. Algorithms with linear complexity are algorithms that perform a fixed number of operations for every input element.

First, we present an overview of the worst-case execution time of the functions that are described in this research in Table 4.1. The worst-case execution time is the time complexity of the function in the worst-case scenario. The worst-case scenario is when the function is called with the worst possible input. For example, the worst-case scenario for the function "insertion" in a list structure is when you have to traverse through every node in the list to find the right position (i.e. the highest or lowest value dependent on implementation). The time complexity of the functions is given in n or m, where n is the number of items in the data structure and m is the number of arrays in the hierarchy (only applicable to the Hierarchical Timing Wheel), to find the right table to insert the timer into and to find the remaining time [5].

Secondly, we present an overview of the average case computation overhead in Table 4.2. As can be seen, there are some differences between the average and worst-case performance of the data structures, we will elaborate on why these differences occur.

The first case is the Hashed Timing Wheel where the retrieval of the first task has a worst-case of $O(n)$ and on the average case is $O(1)$. This is true if $n < tableSize$, and if the hash function (which is $TimerValuemodTableSize$) distributes timer values uniformly across the table [5].

The second case where there is a difference between the average and worst-case complexity, is for the BST. On average the BST is expected to have a complexity of $O(log\ n)$ compared to the $O(n)$ complexity in the worst case. This is because of the depth of the BST. If the BST is evenly distributed there is a complexity of $O(log\ n)$ but when e.g. all items are inserted in descending order, the BST will be fully unbalanced, essentially becoming a linked list where the lowest value is also retrieved at the last checked node, thus getting a complexity of $O(n)$.

Table 4.1: Worst-case time complexity of task dispatcher data structures, where n represents the number of items in the data structure and m represents the number of hierarchical arrays.

Implementation	Task Insertion	Retrieve First Task	Remove First Task
Sorted List	$O(n)$	$O(1)$	$O(1)$
Timing Wheel	$O(1)$	$O(1)$	$O(1)$
Hashed Timing Wheel	$O(1)$	$O(n)$	$O(1)$
Hierarchical Timing Wheel	$O(1)$	$O(m)$	$O(1)$
Bucket of Ignorance	$O(n)$	$O(1)$	$O(1)$
Binary Search Tree	$O(n)$	$O(n)$	$O(n)$
Heap	$O(log\ n)$	$O(log\ n)$	$O(1)$
Red-Black Tree	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$

Table 4.2: Average-case time complexity of task dispatcher data structures, where n represents the number of items in the data structure and m represents the number of hierarchical arrays.

Implementation	Task Insertion	Retrieve First Task	Remove First Task
Sorted List	$O(n)$	$O(1)$	$O(1)$
Timing Wheel	$O(1)$	$O(1)$	$O(1)$
Hashed Timing Wheel	$O(1)$	$O(1)$	$O(1)$
Hierarchical Timing Wheel	$O(1)$	$O(m)$	$O(1)$
Bucket of Ignorance	$O(n)$	$O(1)$	$O(1)$
Binary Search Tree	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$
Heap	$O(log\ n)$	$O(log\ n)$	$O(1)$
Red-Black Tree	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$

4.2 Task Dispatcher Implementations

In this research, several task dispatchers have been implemented within the FreeRTOS kernel, with each of these task dispatchers being based on a different data structure. Starting this section, there will also be a short description of the main differences between the implementation as is given in the description of the different data structures and the real FreeRTOS implementation.

The rest of the section is used to describe the different data structures, which are presented and explained in the fashion they are implemented in the kernel for this research. Here we only talk about the standard implementation of the task insertion, first task retrieval and first task removal functionalities outside of the kernel context. This means that extra information regarding the integration in the kernel is left out. All of the data structures are handled as if each data item in the structure is the same as the entire task and the wake time of the task, which is in reality not the case. This abstraction is made to make the described functionalities much less cluttered and more understandable for the reader.

As can be observed in this section, not all dispatcher implementations described in Section 4.1 are implemented. The Timing Wheel and 2-3-4 Tree are not described in this section and further evaluation sections. For Timing Wheels, there was a gap during implementation between the FreeRTOS kernel and the Timing Wheel implementation that resulted in large overheads and faulty implementations. The timing wheels are fundamentally different from the other implemented data structures in various ways, which makes it difficult to align them in the FreeRTOS kernel. One of these things is that the first task that should be released can not easily be obtained from the timing wheel. This is the case because the timing wheel is designed to only

check the location of the current time, whereas checking for the first to be released task resulted in a much larger overhead.

Thomas Gleixner, which is one of the top 30 developers of Linux, together with Ingo Molnar, tried to implement a timing wheel into the Linux kernel for a timer that has corresponding functionalities with the task dispatcher in FreeRTOS [40]. They unfortunately were also unsuccessful in doing so for various reasons [41].

For the 2-3-4 Tree, in the early stages of the development there was a larger overhead in the function calls compared to the RBT function calls. Since 2-3-4 trees are isomorphic to RBTs, meaning they are equivalent data structures (they can be mapped to each other) we have decided to only continue with the RBT. Similar considerations and decisions were also made with regard to other data structures as will be clarified later in this section for different specific dispatcher implementations.

4.2.1 Implementation to FreeRTOS

In FreeRTOS, each task is represented by a `ListItem_t` structure, as illustrated in Listing 4.1, which encapsulates essential information about the task.

Listing 4.1: ListItem_t FreeRTOS

```
struct xLIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
    configLIST_VOLATILE TickType_t xItemValue;
    struct xLIST_ITEM * configLIST_VOLATILE pxNext;
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
    void * pvOwner;
    struct xLIST * configLIST_VOLATILE pxContainer;
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE
};
typedef struct xLIST_ITEM ListItem_t;
```

In this structure, the `pvOwner` is the actual item of importance in the list structure. For the case of the task dispatcher, this would be the Task Control Block (TCB) which in terms contains all the properties of the task, such as the state of the task, priority, the stack of the task, etc. The delay (period) of the task however is in the `ListItem_t` under `xItemValue`. Moreover, the `pxContainer` is used to link list items (and by implication also the task) to a specific list that holds e.g. the number of items in that list. This list structure `List_t`, presented in Listing 4.2, is used in the kernel to keep track of what the state of a `ListItem_t` (and therefore the task) is at.

Listing 4.2: List_t FreeRTOS

```
typedef struct xLIST
{
    listFIRST_LIST_INTEGRITY_CHECK_VALUE
    volatile UBaseType_t uxNumberOfItems;
    ListItem_t * configLIST_VOLATILE pxIndex;
    MiniListItem_t xListEnd;
    listSECOND_LIST_INTEGRITY_CHECK_VALUE
} List_t;
```

The FreeRTOS kernel has the following of these `List_t` structures to track the tasks throughout the kernel:

- Ready task lists
- Pending ready task list
- Suspended task list
- Waiting termination task list
- Delayed task list
- Overflowed delayed task list

The ready task lists is an array of lists with a set priority, e.g. index [0] of the ready task list is a list of all the ready tasks with priority 0, and index [12] of the ready task list is a list of all the ready tasks with priority 12. These ready task lists are used to track through the kernel if the task is ready to be executed. The ready task lists are used by the scheduler to determine the highest-priority task that is ready to run, and to switch context to that task. When a task is created, it is added to the appropriate ready task list according to its priority. As tasks become ready to run (e.g. when they are unblocked or when a higher-priority task yields the processor), they are moved between the ready task lists.

By maintaining a separate list for each priority level, the scheduler can quickly determine which task to run next without having to search through all of the tasks in the system. This allows the scheduler to be highly efficient and responsive to changes in the system, even in large-scale real-time systems with many tasks.

When a task becomes ready to run, for example when a higher-priority task unblocks it, the FreeRTOS kernel adds the task to the pending ready task list. This list holds all the tasks that are ready to run but are currently blocked by tasks that are already running.

The pending ready task list is checked by the scheduler at the end of each task's time slice or when a higher-priority task becomes unblocked. If there are any tasks in the pending ready task list, the scheduler moves them to the ready list.

The purpose of the pending ready task list is to ensure that tasks that become ready to run are not overlooked by the scheduler. By keeping a separate list of tasks that are pending, FreeRTOS can ensure that all ready tasks get their fair share of CPU time, even if they cannot be scheduled immediately.

The suspended task list is a list used by the FreeRTOS kernel to hold a list of tasks that are currently suspended. Tasks can be suspended in FreeRTOS for a variety of reasons, e.g. when they are waiting for a semaphore or when they have been placed in the Blocked state by a call to a blocking API function.

When a task is suspended, it is removed from the ready list and placed into the suspended task list. This prevents the task from being scheduled by the FreeRTOS kernel until it is unsuspended.

The waiting termination task list is a list of tasks that have been deleted, but whose memory has not yet been freed. A task that is deleting itself can not be done within the task itself due to the need for a context switch to another task. This is why the task is placed in the termination list, later the idle task will check this list and free up any allocated memory allocated by the scheduler for the TCB and the stack of the removed task.

When a task is blocked on a delay, it is added to the delayed task list with its delay period specified. The task remains in this list until its delay period has elapsed, at which point it is moved to the appropriate ready task list based on its priority.

The delayed task list is updated on each system tick, which is a regular time interval set by the kernel configuration. When a tick interrupt occurs, the kernel checks the delayed task list to see if any tasks have completed their delay period. If so, the tasks are moved to the appropriate ready task list.

The delayed task list is used by FreeRTOS to implement software timers and other time-based features. By using the delayed task list to manage time-based events, FreeRTOS can provide accurate and efficient timing in real-time embedded systems.

The overflowed delayed task list is a list that essentially does the same as the normal delayed task list but it stores the tasks that have a delay that exceeds the maximum value of the timer. When the timer resets (when it reaches its maximum) the lists are swapped and so the ”normal” delayed task list becomes the overflowed delayed task list and vice versa, under the condition that the ”normal” delayed task list is empty, which should always be the case when the timer resets.

The delayed task list and the overflowed delay task list are essentially the core of the task dispatcher in FreeRTOS.

An "active" task is always part of one of these lists and therefore this should be taken into account when changing the implementation of the task dispatcher of FreeRTOS. When stepping away from this list structure entirely, not only the task dispatcher has to be reformed but a very large part of the kernel has to be recreated.

To work around this, for this research, the list structure is encapsulated in every implemented structure. As can be seen in the source code in the `list.c` and `list.h` files. E.g. the Heap can be implemented as an Array of `ListItem_t`'s (In Array because this in early stages of measurements resulted in less overhead as described in Subsection 4.2.5), the RBT has the `ListItem_t` of the task as data inside the node. With "Workarounds" such as these it was possible to implement new task dispatchers into FreeRTOS by changing as little as possible kernel behavior outside that of the task dispatcher.

In addition, every implemented data structure should be linked to the corresponding delayed task list or overflowed delayed task list defined by the `List_t` structure in Listing 4.2.

Besides the implementation of the data structure based on the existing `ListItem_t` and linking the data structures to the correct `List_t` some important changes have to be made to the kernel to get different data structures to work correctly.

Firstly, all data structures should be declared and initialized correctly in correspondence with the kernel. Some data structures need extra initialization to setup the data structure which can be handled in the `prvInitialiseTaskLists()` FreeRTOS function, which is called by the kernel on the very first task that is added to the ready task list.

Secondly, if a data structure makes use of more than one location to store the tasks, e.g. the BoI which makes use of two lists, one sorted and one unsorted, you have to make sure that all of the items in both lists are regarded as blocked tasks.

Lastly, it is crucial to manage the overflow of the task dispatcher's data structure in relation to its maximum release time. When the FreeRTOS timer overflows, tasks may surpass this maximum release time. To account for this, an overflowed task list is employed. Each data structure necessitates a corresponding mechanism — an 'overflow handler' — to ensure the task dispatcher continues to function correctly after the timer resets. This handler is managed in the `taskSWITCH_DELAYED_LISTS()` function. Here, the overflowed task data structure is swapped with the regular data structure, which should be empty at the time of the switch. This approach allows the system to operate continuously, even following an overflow event.

The aforementioned alterations constitute the most significant changes necessary within the kernel, external to the actual task dispatcher implementations. In the subsequent sections, we will present the data structures as they are currently implemented, with the assumption that these structural modifications have also been

incorporated. These changes are integral to ensuring the correct functioning of the data structures in the kernel's context.

4.2.2 List

The `list` data structure is currently being used in the FreeRTOS kernel for a multitude of functionalities, one of which is the bases for the task dispatcher. The means of this implementation is currently done via a circular doubly linked list as can be seen in Figure 4.1, which is a list that points to the next node as well as to the previous node. The ”last” node of the list points with its `next` pointer to the first item in the list, and the first item in the list points with the `previous` pointer to the last item in the list. In FreeRTOSes implementation the tail node of the linked list is pointed to by a `ListEnd` pointer. This tail node in the linked list of the task dispatcher contains the maximum possible value, meaning the tail node is always the same and is therefore used as a marker. In this case this would be the longest possible delay value of a task, which is defined by `portMAX_DELAY`.

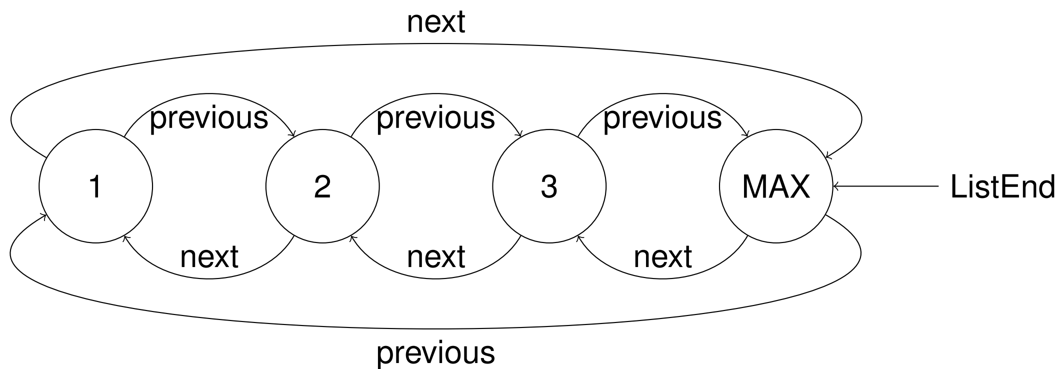

Figure 4.1: FreeRTOS list structure

In order to keep the list ordered in FreeRTOS we can insert an item as is given in Algorithm 1 in Section A.1. Where we iterate through the list and place the item in the list where the previous value is larger than the inserted value.

Since the doubly linked list is ordered in an increasing fashion (w.r.t. the wake time), the item that is first to be dispatched is at the head of the list. Which can be accessed quickly by taking the pointer to the `next` item of the `ListEnd` pointer.

Removing an item in the linked-list structure is also evident, by setting the `previous` pointer of the `next` item to the `previous` pointer of the item that will be removed and setting the `next` pointer of the `previous` item to the `next` pointer of the item that will be removed. This can also be seen in Algorithm 2 within Section A.1.

4.2.3 Bucket of Ignorance

The implementation of the BoI is equal to the list with regard to retrieval and removal of the first task from the data structure. Insertion is where the data structure differs from the list structure.

The implementation of the BoI is done using the existing list structure within the task dispatcher and therefore very few changes to the rest of the kernel had to be made outside of the transformation of the insertion function (i.e. little changes to the kernel outside of the mentioned changes in Subsection 4.2.1).

In Figure 4.2 we give an overview of the insertion function of the BoI, which is represented more extensively in Section A.2. But this image gives a good overview of how the BoI is ordered with regard to the release time. We regard a task only as its release time where τ_i is the task (i.e. the release time) to be inserted, L^o is the ordered list, and L^u is the unordered list of the BoI data structure. From the flow of the BoI insertion in Figure 4.2 it becomes clear that the new task is inserted into the unordered list when either this list is empty beforehand or when the release time of the newly inserted task τ_i is lower than the current head of the unordered list. If this is the case and the ordered list L^o is also empty this will require a refill of the unordered list L^u into the ordered list L^o, which can be found in Algorithm 4 in Section A.2. The new task is only inserted directly into the ordered list L^o when the unordered list L^u is not empty and the newly inserted task has a lower release time than the current head of this list.

There are different implementations of the BoI tested earlier in the development stage with a wide variety of task sets. As a result of these measurements, we found that the BoIList implementation gave the most optimal results with regard to computation overhead [42]. This is why we use this implementation from this point forward, discussing the BoI and only measurement results of this implementation are given in Chapter 5.

4.2.4 Binary Search Tree

For the BST we have created the `BSTItem_t` which is shown in Listing 4.3. As can be seen, we use the `ListItem_t` from Listing 4.1 as the data of the BST node. So also the information e.g. the release time stays within the `ListItem_t` and we do not save the data multiple times. We also do not use a pointer to the parent node because for the implementations required for the task dispatcher, this is not necessary.

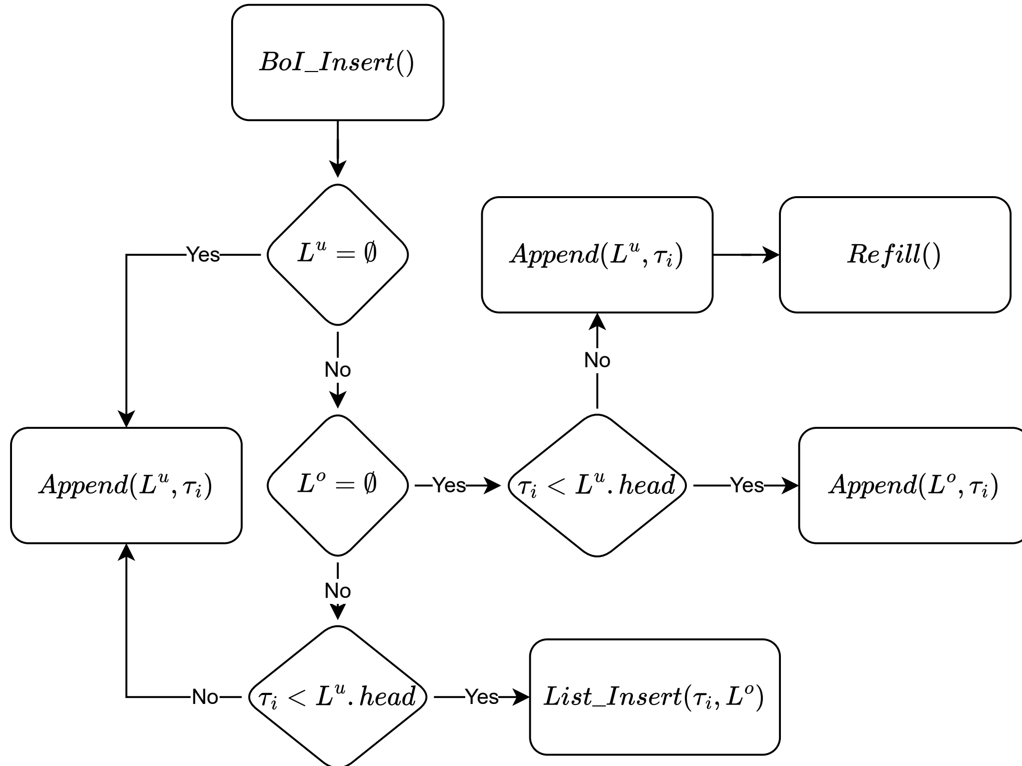

Figure 4.2: Flow diagram of the Bucket of Ignorance insertion function.

Listing 4.3: BSTItem_t Implementation

```
struct xBINARY_SEARCH_TREE_ITEM
{
        struct xLIST_ITEM * xItem;
        struct xBINARY_SEARCH_TREE_ITEM *pxLeft;
        struct xBINARY_SEARCH_TREE_ITEM *pxRight;
};
typedef struct xBINARY_SEARCH_TREE_ITEM BSTItem_t;
```

In Figure 4.3 we describe in a flow chart how the insertion of the BST is implemented. For simplicity, we only take into account the release time (as described earlier) in this description. Let B be the root of the BST, c and p as the pointer to the current node and its parent node respectively and τ_i be the task (release time) that is to be inserted into the BST.

If a BST is not yet established, the root is $NULL$, and the item for insertion becomes the root node, with its left and right children also being $NULL$. This is, of course, the case at the startup of the system, where the kernel sets the root of the BST to `PRIVILEGED_DATA static BSTItem_t *pxDelayedTaskBST = NULL;`. If a root exists, the current node c, utilized for traversing the tree, is set to the root B, with the parent p being set to $NULL$.

The subsequent step involves checking if the current node c is $NULL$. If this is the case, the parent node p is set to the current node c, after which we check if the task to be inserted has a lower release time than the parent node. If this holds true, we set the current node to the left child of the current node, if not we set it to its right node. This process repeats until we have traversed the entire current node. At which instance, the new release time should be inserted as either the left or the right child of the parent node, depending on if the release time of the to-be-inserted task is lower than its parent.

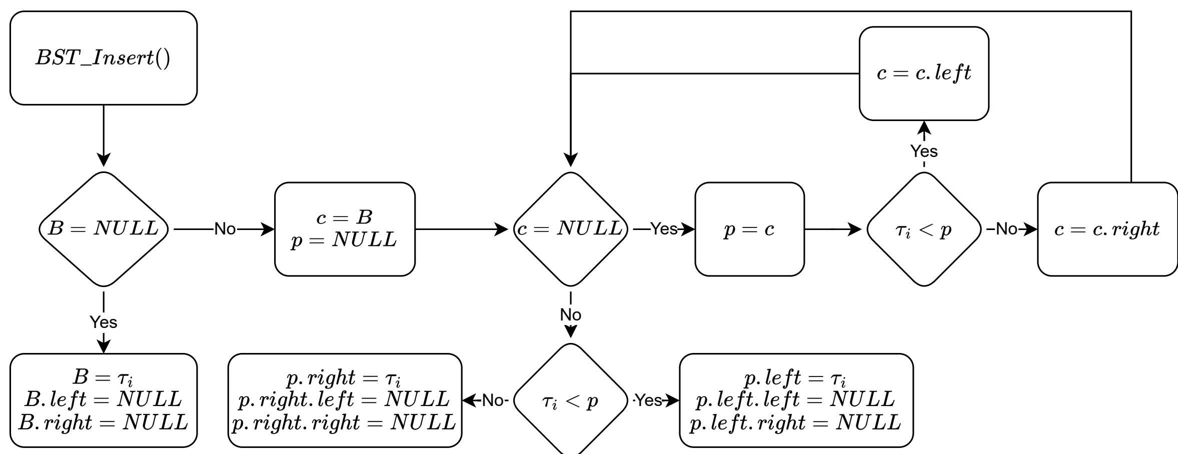

Figure 4.3: Flow diagram of the BST insertion function.

For the retrieval and removal of a node within the BST data structure in the task dispatcher the first task will always be located at the leftmost node in the tree. By consecutively traversing through the left side of the tree until there is a `NULL` node at the left of the current node. The current node would then be the lowest value. For the retrieval, we want to give the contents of this node to the kernel. For the removal, the node should be removed and the right child of the current node should be made the new left child of the parent node. This way the BST properties will hold and no nodes will be lost.

4.2.5 Heap

For the implementation of the Heap, we have considered (and tested out) two different implementations. We have the pointer-based Heap and the Array-based Heap. The first is far less common, an implementation of this however exists in e.g. libuv, which is a multi-platform software library that provides asynchronous I/O and event-driven programming capabilities for building high-performance network applications [43]. An advantage of this would be that the upper size would not have to be pre-defined, which is the case for an Array-based implementation. However,

since in FreeRTOS tasks are predefined, the upper bound of the Heap can also be predefined. Testing in early stages of the implementation of the Heap-based task dispatcher we measured computation overhead for both implementations and the Array-based Heap outperformed the pointer-based Heap. We therefore did not include the pointer-based Heap implementation when conducting the measurement results in Chapter 5.

Within the Array-based Heap implementation, there were also two implementations tested. A recursive implementation and an iterative implementation. In this case, the iterative implementation outperformed the recursive implementation on every task set we measured them for, this is the version that will be described in this chapter and that is measured in Chapter 5.

In Figure 4.4 a brief visual description is given of how the release time would be inserted into the implemented Array-based iterative Heap data structure. Let H be the Array-based Heap, τ_i the to-be-inserted task (release time) and n corresponds initially with the value of the number of items in the Heap.

Initially, the new task is assigned to the final slot in the Heap. Following this insertion, we determine whether the original Heap was not empty and whether the Heap item at index n is less than its parent (i.e., the item located at index $\frac{n-1}{2}$). If these conditions hold true, the items at these positions interchange, with an appropriate adjustment of n. This process is repeated until the Heap's properties are restored and maintained.

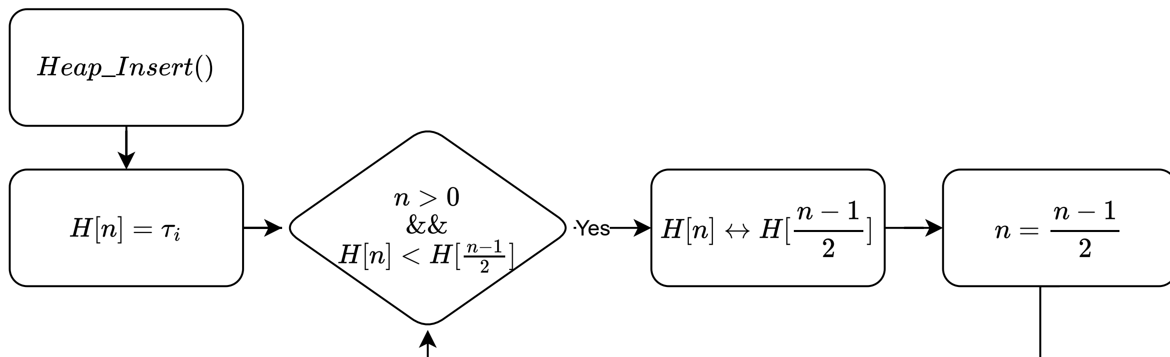

Figure 4.4: Flow diagram of the Array-based Heap insertion function.

For the retrieval of the first task, we can simply address the root of the Heap. However, to remove this we will have to make sure that we have a complete Heap and that the Heap property will hold. Our manner of implementation is described in Figure 4.5.

Let n initially be the number of tasks in the dispatcher, H be the Array-based Heap, c the current node that is under iteration (starting with 0) and r and l be the right and left children respectively. In the Array-based Heap, the right and left children are equal to the respective parent where $l = 2 \cdot parent + 1$ and $r = 2 \cdot parent + 2$.

We start off by decrementing the size of the Heap by one, followed by overwriting the root of the Heap by the last item in the Array of the Heap. This is followed by iteratively comparing the root node to its children and swapping them if necessary. This continues until the root node is smaller than both of its children. I.e. the first to be released task is replaced by the last node in the Heap and the Heap property is restored by iteratively swapping nodes in the Heap.

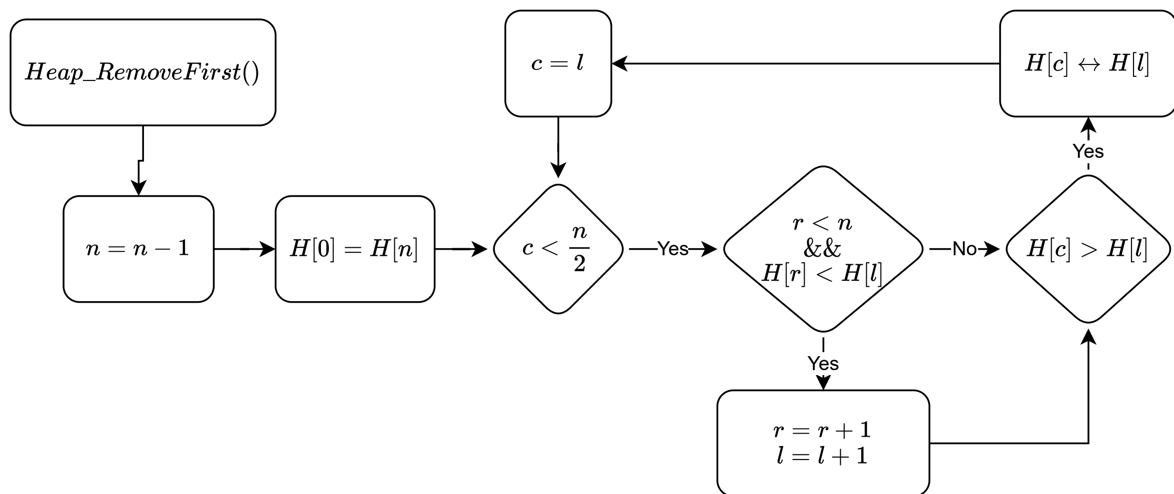

Figure 4.5: Flow diagram of the Array-based iterative first task removal function.

4.2.6 Red-Black Tree

Insertion into the RBT works almost similar to that of the BST in Subsection 4.2.4, as was presented in Section 4.1. The main difference is that the newly inserted node is initially marked as `RED` and after insertion the RBT is ”fixed up” (see Algorithm 6 in Section A.3), which is the algorithm that is used to make sure that after changes made to the RBT, the RBT will be made compliant the RBT properties.

Since only the lowest value node has to be removed from the RBT the removal function can be improved in terms of computation overhead compared to a normal RBT removal function (see Algorithm 7 in Section A.3). When removing the lowest value of the RBT there is another advantage that can be taken into account compared to a normal removal, only two violations of the RBT properties that are presented in Section 4.1 can be violated because of this. The two properties that can be violated are:

- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
- If a node is red, then both its children are black.

In the way the insertion is handled, these are also the only violations that can happen for insertion. This creates an opportunity for this research to use one `FixUp` function for both the insertion and the removal of the first task functions, which generally is not the case for arbitrary removal functions in RBTs

4.3 FreeRTOS Test Application

In order to evaluate changes made with regard to the task dispatcher in a real-world context, we need to have a FreeRTOS application that will create the tasks for the desired task set.

The test application contains three important files for its execution. For this, we do not take into account other important files with regards to e.g. the debugger, such as the `launch` or `ini` files. The

The `helpers.c` file primarily offers the implementation of the tasks for the test set. The `config.h` and `main.c` files allow the definition of each task using a structure, enabling the customization of various attributes such as periods or execution times. This flexibility significantly simplifies the creation of numerous tasks without the necessity of individually specifying and establishing each one.

The `main.c` file iteratively processes the tasks configured in `config.h`, generating the tasks accordingly. Through the use of defines, we can seamlessly obtain overhead measurements from the kernel. This information can then be transmitted via UART.

4.4 Limitations of Testing on Embedded Devices

Measuring software on a real device is an essential practice, providing a realistic and accurate assessment of the software's performance, behavior, and compatibility with different devices and configurations. This information aids developers in refining the software and ensuring it meets the needs and expectations of its users.

Despite its importance, conducting FreeRTOS kernel testing on an embedded device introduces certain constraints and limitations. While some of these limiting factors were mitigated using the tools described in Chapter 3, a significant constraint pertains to the testing of task dispatcher implementations.

This constraint is imposed by limited system resources, specifically memory. Embedded systems typically operate within a constrained environment, with limited memory capacity. As tasks in a real-time operating system like FreeRTOS are primarily stored in memory, the number of tasks that can be handled simultaneously is directly influenced by the available memory.

Memory constraints can cause complications such as slow system performance, task execution failures, or in severe cases, system crashes due to memory exhaustion. As a result, it is necessary to manage the task set size diligently to prevent such issues.

Through extensive testing with various task set sizes and by tracking the heap memory—which is stored in the RAM—we have determined the upper limit of safe operation. To prevent potential issues that could arise from exhausting the device's memory and to ensure the behavioral correctness of the implementations under test, we have set a cap on the task set size, restricting it to a maximum of 150 tasks. This limitation was established based on the observed memory usage and system behavior during our testing.

Chapter 5

Evaluation

In this chapter, we will evaluate the performance of the task dispatcher using the various task dispatcher implementations described in Chapter 4 to determine if the computation overhead of the task dispatcher is reduced using these new task dispatcher implementations and if they provide possibilities for optimization benefits for specific scenarios.

To achieve this, we have set up an evaluation process, as described in Chapter 3, by running esp-idf FreeRTOS with the customized task dispatcher implementations on an ESP32-S3-DevKitC-1 microcontroller. For the evaluation of the performance, two main methods of analysis were used, debugging toolset and run-time tracking of the clock cycles within the task dispatcher functions.

During the evaluation process, we measured the clock cycles within the task dispatcher for the three main functionalities (task insertion, first task retrieval and first task removal) for each implemented task dispatcher and plotted them to provide visual insight. The evaluation results will be presented and discussed in this chapter, providing insights into the performance of the different implemented task dispatchers. This assessment will aid in determining the appropriateness of various task dispatchers within FreeRTOS, while also providing insights into potential improvements across a wider range of RTOS task dispatchers concerning computational overhead

5.1 Task Set Synthesis

In this section, we discuss the task sets used to evaluate the performance of the various task dispatcher implementations. The task sets were based on the real-world automotive benchmarks given in a paper by Kramer et al. [1]. These benchmarks were chosen due to their significance as they were provided by Bosch, abstracted from real-world automotive applications, and demonstrate the importance of period

variance in task dispatcher performance. The task sets' variance in period is significant, as it enables a thorough assessment of the task dispatcher's adaptability and efficiency in managing tasks with differing timing requirements.

The automotive benchmark includes a task distribution shown in Table 5.1. It is important to highlight that the original distribution also includes 15% angle-synchronous (AS) tasks. These tasks, which are dependent on the speed of the crankshaft and the number of cylinders in the engine, can exhibit a high degree of variability. For the purpose of our measurements, these tasks were not included. Consequently, we adapted the distribution by proportionally distributing the remaining 85% across the different task periods to sum up to 100%, as shown in the table. In Table 5.2, we present the adjusted distribution, where each task period's share has been proportionally adjusted to account for the total of 100%. This normalized distribution was used in our analysis.

Table 5.1: Distribution of task periods in automotive benchmark [1].

Period (ms)	1	2	5	10	20	50	100	200	1000	AS
Share (%)	3	2	2	25	25	3	20	1	4	15

Table 5.2: Normalized distribution of tasks in automotive benchmark, excluding angle-synchronous tasks.

Period (ms)	1	2	5	10	20	50	100	200	1000
Share (%)	3.53	2.35	2.35	29.41	29.41	3.53	23.53	1.18	4.71

We evaluated four different sets of task sets to provide a comprehensive understanding of the performance of the task dispatchers under various scenarios. The first set of task sets involves evaluating the worst-case scenario for a single execution of the primary functionalities (i.e. task insertion, first task retrieval and first task removal). The primary objective of this set was to create an upper bound, providing a realistic view of the potential risks that may arise given a specific task dispatcher implementation. We evaluated this set for each implementation, ranging from one task per implementation up to 150 tasks per implementation.

The second set of task sets we evaluated was a homogeneous set, where every task had the same period. We evaluated this set for each implementation ranging from one task per implementation up to 150 tasks per implementation. We examine if there is a notable trend in the behavior of the task dispatcher functionalities in response to these sets of fully homogeneous period task sets. This aspect is of particular significance for industries such as the automotive sector, where substantial proportions of tasks operate within a limited number of periods. For instance, as

highlighted in the ”Real World Automotive Benchmark For Free” paper [1], merely two task periods account for $50\,\%$ of the tasks (rising to $59\,\%$ if angle-synchronous tasks are excluded).

The third set of task sets was based on a completely uniform task distribution of the task periods described in the paper by Kramer et al. [1]. For every period used in the benchmark, an equal amount of tasks was created. In this instance, there is a total of 9 different task periods as is presented in Table 5.1. We evaluated this set for each implementation ranging from a task set of one uniform distribution subset (existing of 9 tasks) up to a task set of 16 uniform distribution subsets, resulting in a range of 9 up to 144 tasks. This set of task sets has been selected due to its capacity to produce a well-distributed data structure within the task dispatcher. This approach allows for an assessment of the impact that the distribution of task periods has on the task dispatcher's overhead.

The fourth and final task set encompassed the automotive distribution. This set replicated the distribution as specified in the cited automotive benchmark paper, featuring one task for each percentage point in the non-normalized distribution. (For the normalized distribution, this distribution is unique in its ability to ensure tasks are expressed as integer values without altering the overall distribution.) In this set, we measured only one task set. This task set is included as it can provide valuable insights into the advantages that may be specific to this case. It is crucial to note, however, that the findings are not directly scalable to accommodate the desired quantity of tasks running concurrently.

Overall, by evaluating these four sets of task sets, we were able to obtain some understanding of the performance of the various implementations under different scenarios. Please note that the measurements in this chapter are done for specific task sets and cannot be applied universally. Instead, this thesis presents a selection of measurement results from specific task sets that provide valuable insight into the correlation between overhead and task set. Each measurement is described in detail to provide a clear understanding of the correlation.

While these benchmarks were specifically designed for automotive applications, the results we obtained are applicable to a wide range of real-time operating system (RTOS) applications due to the similarities between them.

5.2 Worst-Case Computation Overhead

Here we evaluate the computational overhead of the task dispatching implementations based on the five data structures described in Chapter 4. It is important to note that the worst case is based on a single execution worst-case. It is evident that triggering the worst-case computation overhead is different per data structure,

e.g. the worst-case insertion for the implementation of the List could be triggered by adding all tasks in descending order based on release time, where the last item that is inserted will have the worst-case computation overhead since it has to traverse through all the release times in the list. Which is not necessarily the worst-case execution time for other data structures/function calls.

5.2.1 List

We present the worst-case execution of the three different functionalities of the List-based task dispatcher.

The List-based task dispatcher task insertion function is performing a sorted insertion into a doubly linked list. The operation that consumes most of the time is the for loop that iterates over the list to find the correct position where the new item should be inserted. In the worst-case scenario, the for loop is iterating n times, therefore the time complexity of this function is $O(n)$ in the worst case. The worst-case execution is presented in Figure 5.1 from 1 up to 150 tasks. The linear relations between the number of tasks and the computation overhead is evident in this case, which is exactly as was expected.

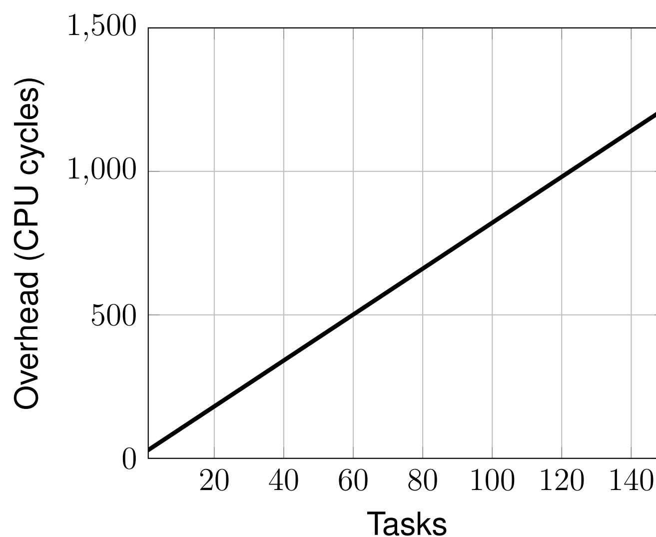

Figure 5.1: Worst-case computation overhead of List-based task dispatcher task insertion implementation per task set size.

For the retrieval of the first task, we can easily retrieve this by calling `( ( &( ( pxList )->xListEnd ) )->pxNext->pvOwner )`. The CPU clock cycle overhead of this call will remain the same and results in a constant of 3 CPU per call.

The function for removing the first item of the list (or any item for that matter) also performs a constant number of operations regardless of the inputs of the structure.

The function simply updates the pointer of the previous and the next. Which in turn gave us an overhead of 12 CPU cycles.

The results for the measurements done for the List-based task dispatcher, described in this subsection and graphically displayed in Figure 5.1, are summarized in Table 5.3, allowing for a more precise quantitative analysis of the relationship between the overhead and the number of tasks.

These formulas, which we will continue to use in this and following subsections, offer several advantages. They enhance precision, facilitate predictive analysis, simplify calculations, and provide insights into the underlying relationship between variables, without needing repeated explanations.

Table 5.3: Worst case computation overhead of List-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher.

Task Insertion	$f(n) = 21 + (8 \cdot n)$
First Task Retrieval	$f(n) = 3$
First Task Removal	$f(n) = 12$

5.2.2 Bucket of Ignorance

The worst-case execution of the BoI-based task dispatcher would be if the bucket would fill up the unsorted list, which in terms triggers the `Refill()` function for a large unsorted list. To order this again as is described in Subsection 4.2.3 causes a very large worst-case overhead. The worst-case was not deemed to be useful for the BoI since it is much larger than all the other data structures due to the aforementioned effect. We can see that, e.g. in later task set specific measurements the worst-case of specific task sets is already explosive, see Figure 5.14. Where in earlier unoptimized implementations could reach up to $100\,000$ CPU cycles, now it can still easily reach up to $10\,000$ CPU cycles. The goal of the BoI implementation is not to improve worst-case, but to improve average case computation overhead for task sets and this becomes very clear in worst-case analysis. Triggering worst cases for the BoI-based task dispatcher also caused noticeable jitter in the kernel, which can cause missing of deadlines due to synchronization issues.

The function calls for the retrieval and the removal of the first task are the same function calls done for the List based task dispatcher, presented in Subsection 5.2.1, and thus result in the same overhead.

5.2.3 Binary Search Tree

The worst computation overhead of the BST-based task dispatcher task insertion, retrieval of the first task and removal of the first task functionalities, presented in Figure 5.2 5.3 and 5.4 respectively, do all indeed have a linear complexity as was described in Subsection 4.2.4. As can be seen, the difference between the retrieval and the removal of the first task in Figure 5.3 and 5.4 respectively, is rather low. This is due to the fact that the only addition to the removal of the first task compared to the retrieval of the first task is the removal of the node, which can be done in constant time.

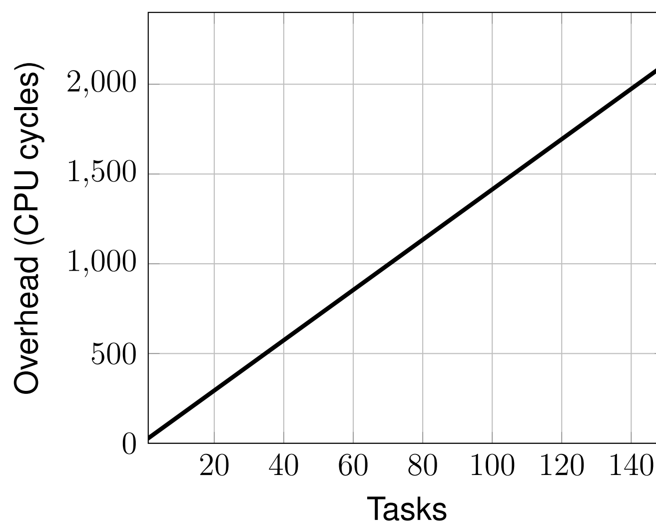

Figure 5.2: Worst-case computation overhead of BST-based task dispatcher task insertion implementation per task set size.

The graphical data presented in Figure 5.2, 5.3 and 5.4 was transformed into mathematical formulas seen in Table 5.4, allowing for a more precise quantitative analysis of the relationship between the overhead and the number of tasks.

Table 5.4: Worst-case computation overhead of BST-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher.

Task Insertion	$f(n) = 14 + (14 \cdot n)$
First Task Retrieval	$f(n) = 1 + (8 \cdot n)$
First Task Removal	$f(n) = 20 + (8 \cdot n)$

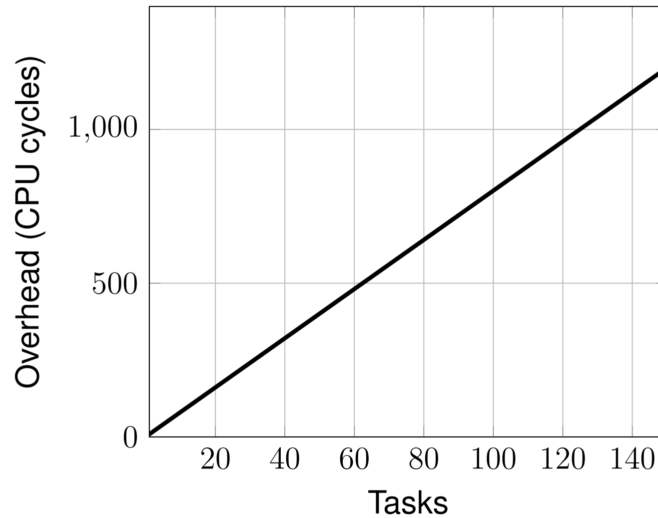

Figure 5.3: Worst-case computation overhead of BST-based task dispatcher first task retrieval implementation per task set size.

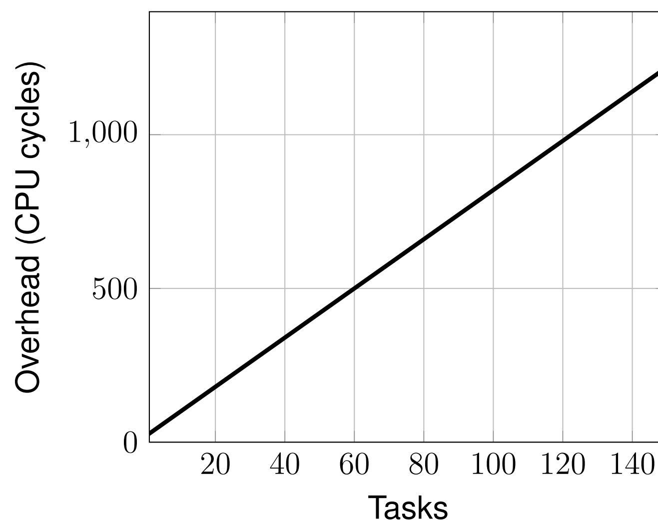

Figure 5.4: Worst-case computation overhead of BST-based task dispatcher first task removal implementation per task set size.

5.2.4 Heap

For the worst-case computation overhead measurements of the Heap-based task dispatcher task insertion presented in Figure 5.5, we can see the logarithmic relation of the function that was described in Subsection 4.2.5 very well. The declarations of values and steps that are always executed contribute to 77 CPU cycles and each iteration for the insertion loop costs at most 166 CPU cycles, resulting in a total calculated overhead of $(\lfloor(\log_2(n)\rfloor \cdot 166)\ + 77$. This corresponds with the measured values given in Figure 5.5. The next overhead increment should e.g. be at 256 tasks where then the total would come down to 1405 if this were possible on the device.

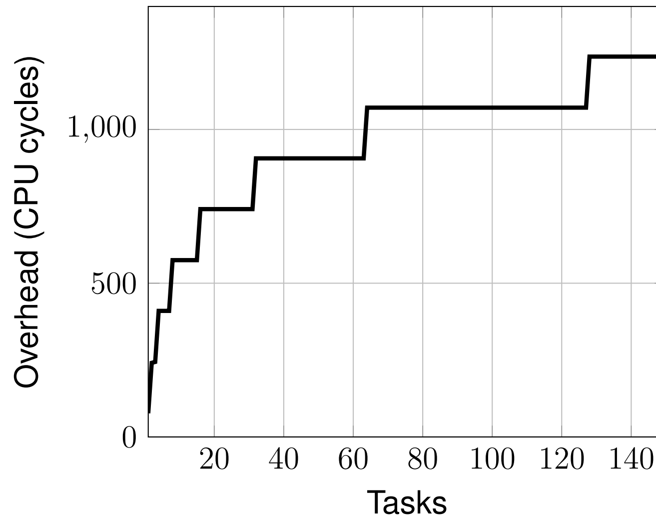

Figure 5.5: Worst-case computation overhead of Heap-based task dispatcher task insertion implementation per task set size.

Retrieving the first task has a constant time complexity and will only take one (1) CPU cycle.

The worst-case time complexity of the first task removal function also has a time complexity of O(log n). This is because the function uses a min Heap data structure, which is a complete binary tree where each parent node has a value less than or equal to its child nodes. The "heapify" equivalent, which is used to maintain the Heap property of the tree, takes O(log n). Also for this function, it can be seen in Figure 5.6 that, in the measurements, there is a logarithmic relationship between the function and the number of tasks. There is however, as can also be seen if you look closely near the places where $\log_2(n-1)$ would reach a new integer value that it is not instantly a stable value. This is because of the second if statement inside the loop, which makes the first two values of the new increase a little lower than the ones following.

The CPU cycles of the removal of the first task of the Heap based task dispatcher can be fairly well estimated as $f(n) \approx \begin{cases} 77 & \text{if } n \leq 2 \\ \lfloor \log_2(n-1) \rfloor \cdot 185 + 82 & \text{otherwise} \end{cases}$.

The graphical data presented in Figure 5.5 and 5.6 was transformed into mathematical formulas seen in Table 5.5, allowing for a more precise quantitative analysis of the relationship between the overhead and the number of tasks.

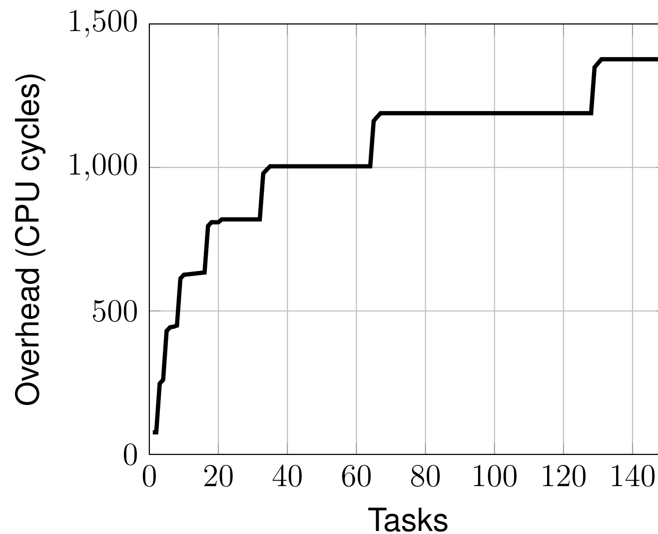

Figure 5.6: Worst-case computation overhead of Heap-based task dispatcher first task removal implementation per task set size.

Table 5.5: Worst-case computation overhead of Heap-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher.

Task Insertion	$f(n) = (\lfloor \log_2(n) \rfloor \cdot 166) + 77$
First Task Retrieval	$f(n) = 1$
First Task Removal	$f(n) \approx \begin{cases} 77 & \text{if } n \leq 2 \\ (\lfloor \log_2(n-1) \rfloor) \cdot 185) + 82 & \text{otherwise} \end{cases}$

5.2.5 Red-Black Tree

The time complexity of the insertion function is $O(log\ n)$ worst-case, where n is the number of elements in the tree. This is because it performs a search operation in the tree, which has a time complexity of $O(log\ n)$, and then performs additional operations to fix up the red-black tree properties, which also have a time complexity of $O(log\ n)$.

As was outlined in Section 4.1, it can be proven that the maximum height of an RBT is $2 \cdot log(n+1)$, where n is the number of tasks in the RBT. However, in the RBT, triggering the full scope of worst-case measurements, particularly a large imbalance, was not achieved. For this reason, we calculated the worst-case based on the maximum number of iterations of the while loop for finding the appropriate node, along with the maximum possible number of ”fixup” calls (worst-case). These calculations underpin the findings presented in this subsection; note, however, they are not entirely corroborated by measurements due to the aforementioned limitations.

The insertion results are composed of a static 22 CPU cycles before and after the loop and fixup function. The loop constitutes 17 CPU cycles per iteration and the fixup function to 98 CPU cycles per execution at max. Since both of these can in theory be called $2 \cdot log(n+1)$ times, this gives a total of $f(n) = 22 + (230 \cdot \lceil \log_2(n+1) \rceil)$ as given in Figure 5.7

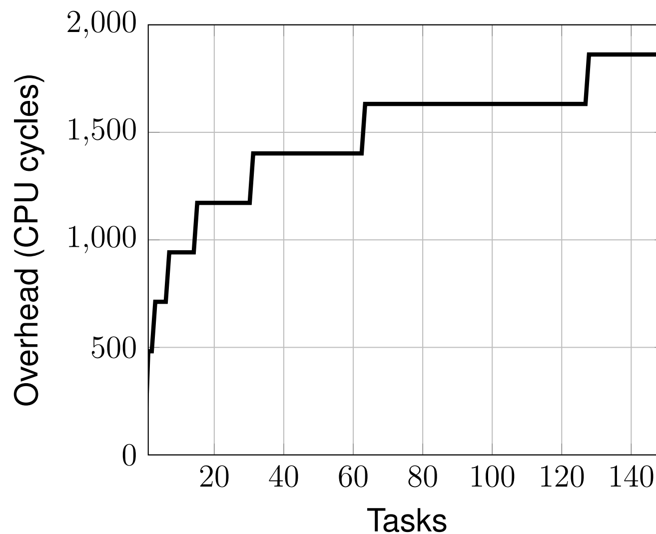

Figure 5.7: Worst-case computation overhead of RBT-based task dispatcher task insertion implementation per task set size.

The retrieval of the first task consists only of a while loop to the leftmost node and will in turn result in a total overhead of 6 CPU cycles per loop iteration, resulting in a worst-case overhead of $f(n) = 12 \cdot \lceil \log_2(n+1) \rceil$, visually displayed in Figure 5.8

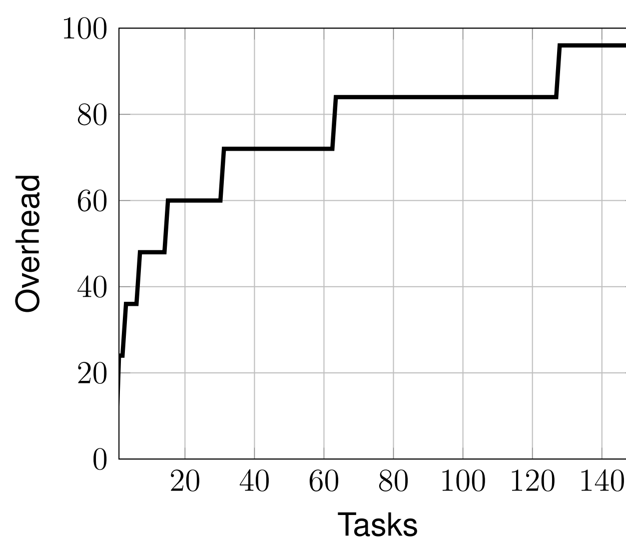

Figure 5.8: Worst-case computation overhead of RBT-based task dispatcher first task retrieval implementation per task set size.

The removal of the first task is built up in a similar fashion, where a total CPU cycle overhead of 18 is static, the loop constitutes 6 CPU cycles per iteration and the fixup function again constitutes 98 CPU cycles per call at a maximum. This results in a total worst-case computation overhead of $f(n) = 18 + (208 \cdot \lceil \log_2(n+1) \rceil)$. Which is graphically presented in Figure 5.9.

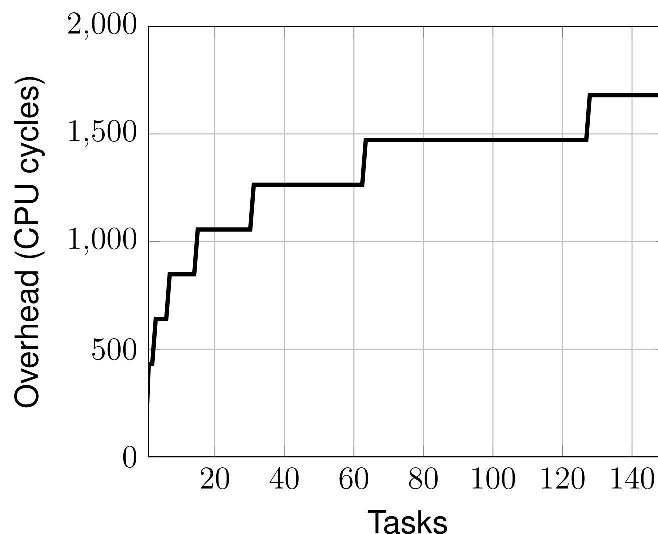

Figure 5.9: Worst-case computation overhead of RBT-based task dispatcher first task removal implementation per task set size.

The graphical data illustrated in Figures Figure 5.7, Figure 5.8 and Figure 5.9 has been translated into mathematical formulas, as seen in Table 5.6. This transformation enables a more detailed quantitative examination of the relationship between computational overhead and the number of tasks.

Table 5.6: Worst case computation overhead of RBT-based task dispatcher functions in FreeRTOS in CPU cycles, where n is the number of tasks in the dispatcher.

Task Insertion	$f(n) = 22 + (230 \cdot \lceil \log_2(n+1) \rceil)$
First Task Retrieval	$f(n) = 12 \cdot \lceil \log_2(n+1) \rceil$
First Task Removal	$f(n) = 12 \cdot \lceil \log_2(n+1) \rceil$

5.2.6 Comparison of Worst-Case Computation Overhead

The evaluation of the worst-case computation overhead for different task dispatcher implementations based on five data structures - List, BoI, BST, RBT and Heap- provides several key insights. By comparing the overhead of insertion, first task retrieval and first task removal functions presented in Figure 5.10, 5.11 and 5.12 respectively, we can better understand the worst-case performance of these implementations under various task set sizes.

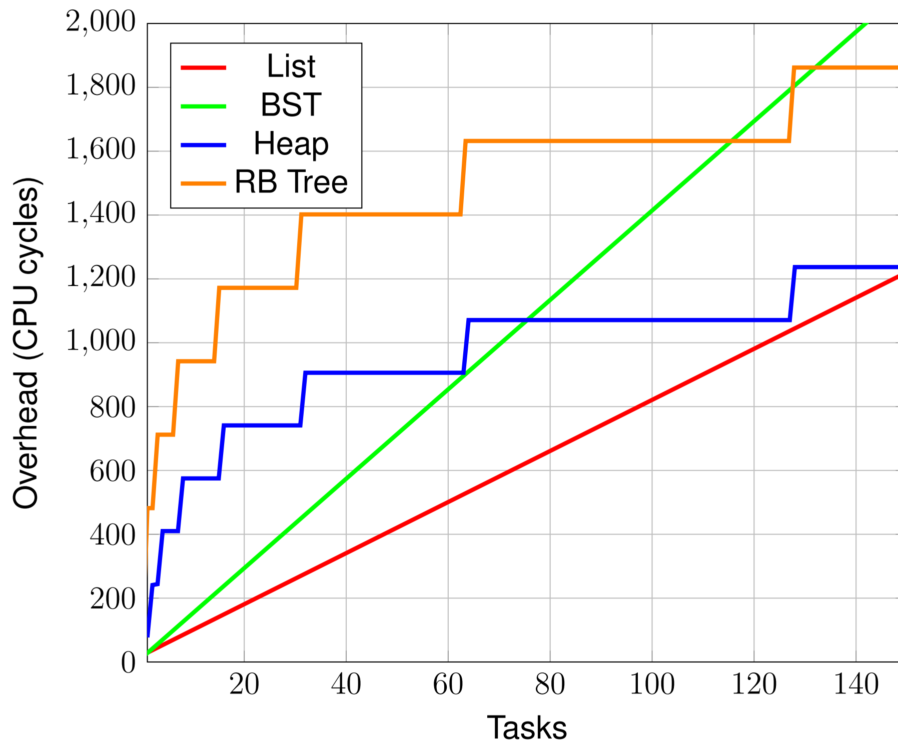

Figure 5.10: Worst-case computation overhead comparison of different task dispatcher task insertion implementations.

Up to 150 tasks (our maximum), the List-based task dispatcher demonstrates superior worst-case performance compared to the other implementations, not only due to the constant removal and retrieval functions but also the lowest overhead for task insertion. Based on the evident trend in the presented figures, if this trend persists, we can state that the worst-case performance of the List-based task dispatcher will be outperformed by the Heap-based task dispatcher at approximately 370 tasks. Which includes the performance of task insertion, first task retrieval and first task removal, due to their respective $log(n)$ complexity.

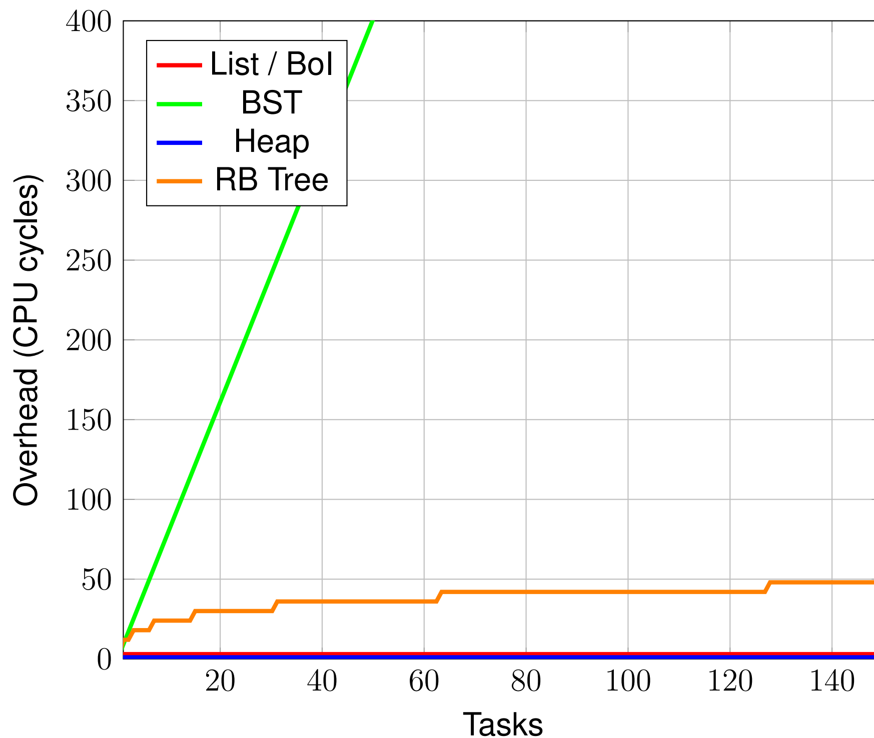

Figure 5.11: Worst-case computation overhead comparison of different task dispatcher first task retrieval implementations.

In contrast, the BoI-based implementation performs significantly worse than its counterparts, even in some cases resulting in noticeable jitter inside the RTOS when the task set is large. It is essential to note that the worst-case values presented are derived from a single execution comparison, it is impossible for every function execution to be equal to the worst-case scenario, which is where the following subsections will give a better indication of what to expect in reality. It is however good to know the upper bound of the execution in order to make a well-balanced decision. Therefore, these worst-case values should be considered an indication of the maximum jitter that can be expected in the kernel when implementing these different task dispatchers.

It is evident that the task dispatcher implementations do comply with the worst-case complexity discussed in Subsection 4.1.1 and presented in Table 4.1. It is also a good illustration that better time complexity does not always translate into lower overhead. This is especially true in lower input sizes due to the simplicity of code for data structures such as lists that outweigh the time complexity of data structures with a better time complexity than lists.

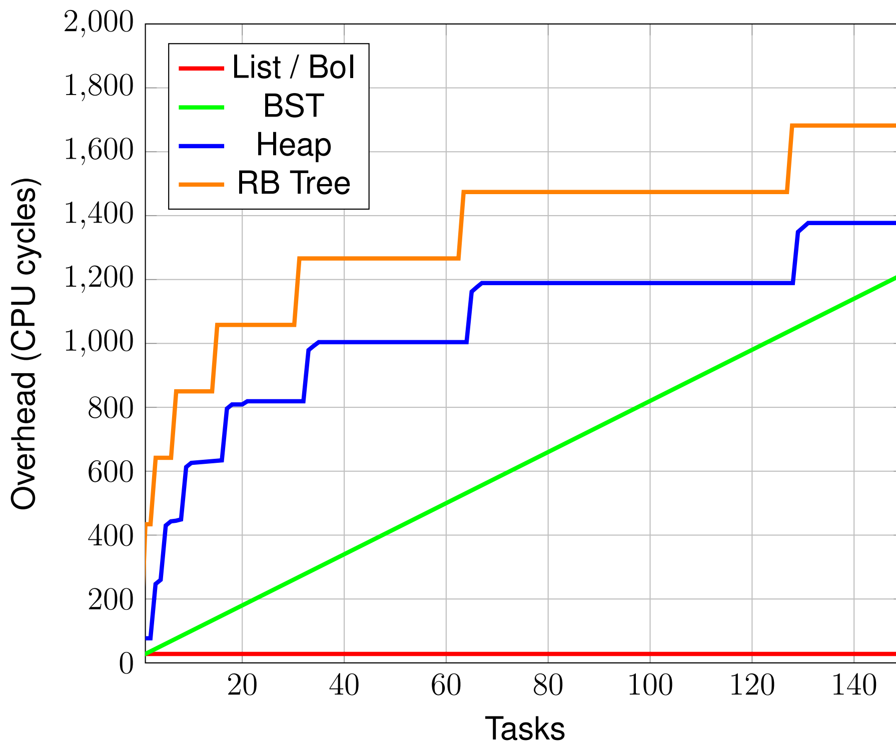

Figure 5.12: Worst-case computation overhead comparison of different task dispatcher first task removal implementations.

5.3 Task Sets with Homogeneous Period Distribution

The purpose of this measurement is to see how the different task dispatchers respond to a task set that exists of n tasks that all have the same period x. For these measurements, it does not matter what value is used for x as long as the schedule is viable (i.e. deadlines are met) and the x is equal in all tasks.

In the evaluation of these results, it is important to note that there is an area with lower opacity around the data points in the graph. This lower opacity area indicates the range between the best- and worst-case performance for that specific task set. It is worth noting that a single execution of the task set can have any value between the lower and upper bound. The line in between these bounds represents the average value of the execution times.

By observing the area with lower opacity around the data points, we can gain a better understanding of the performance of the task dispatchers under different scenarios. For instance, a smaller area with lower opacity would suggest that the task dispatcher's performance is more consistent and reliable, while a larger area with lower opacity would suggest that the task dispatcher's performance is more variable and unpredictable.

5.3.1 List

In this section, we examine the list-based task dispatcher response to cases where all tasks have an equal period as presented in Figure 5.13. Our analysis reveals that the worst-case computation overhead (the upper bound) corresponds with the analysis of Subsection 5.2.1. This is expected behavior since the task list will be filled completely, meaning that there have to be insertions at both ends of the list determining the worst and best-case overhead. The best case is constant for every number of tasks because of the optimal insertion position. The worst and the average case increment by 8 and 4 CPU cycles per additional task respectively, meaning that the average case for inserting a task into a List-based task dispatcher with homogeneous periods will always be approximately half of its worst case.

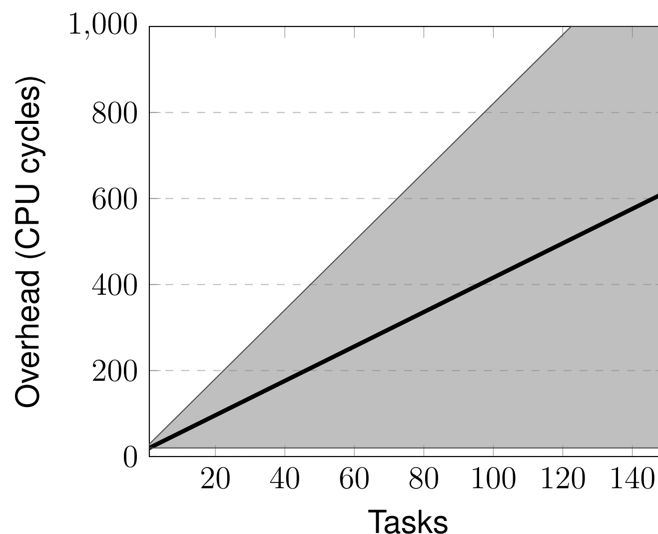

Figure 5.13: Overhead measurements for List-based task dispatcher task insertion implementation for homogeneous task sets.

Retrieval and removal of the first task to be dispatched remain 3 and 12 clock cycles respectively, as presented in Subsection 5.2.1 it remains the same execution independent on the task set.

5.3.2 Bucket of Ignorance

Examining the measurement results of the BoI insertion presented in Figure 5.14, a unique characteristic stands out compared to other dispatching methodologies: the upper bound, or worst-case computation overhead, becomes quite substantial. This increase is clearly visible in the graph as a distinct increase in the lower opacity area representing the possible computation overheads. As a reference around $n = 14$ the CPU cycles in the worst-case for this task set exceed 1000 CPU cycles and at $n = 90$ it already exceeds 9000 CPU cycles in overhead. However, its average is not increasing as much (201 and 534 respectively). This is due to its design choice to only sort the "bucket" when needed. This in terms gives a high computation overhead when the "bucket" is at a high capacity and the ordered list is almost empty. The average case seems to be rising with approximately 4 CPU cycles per task. Retrieval and removal of the first task remain equal to the List-based task dispatcher implementation i.e. 3 and 12 clock cycles respectively.

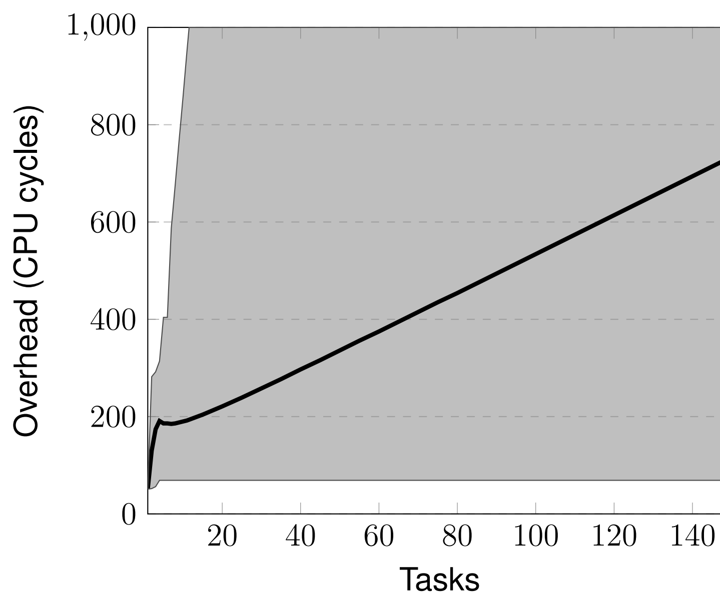

Figure 5.14: Overhead measurements for BoI-based task dispatcher task insertion implementation for homogeneous task sets.

5.3.3 Binary Search Tree

The set of homogeneous period task sets for the BST-based task dispatcher provides the same worst-case computation overhead as the overall worst-case overhead $(14 + (14 \cdot n)$ for the insertion. The best-case computation overhead takes 20 clock cycles for insertion in the BST-based task dispatcher. When looking at the average measurements it gives a clear linear relation which can be described as $10 + 7 \cdot n$. Which in terms of overhead increment per task is half of the worst-case computation overhead.

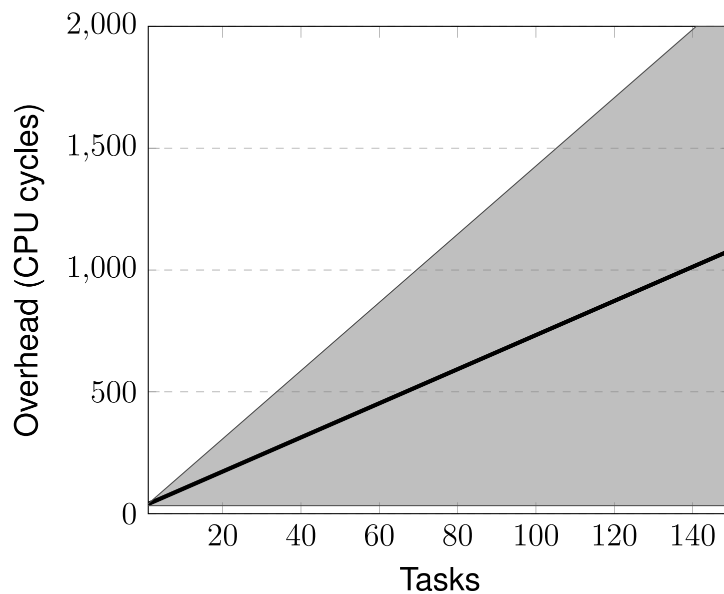

Figure 5.15: Overhead measurements for BST-based task dispatcher task insertion implementation for homogeneous task sets.

Looking at the retrieval and removal functionalities of the BST-based task dispatcher, the computation overhead takes a constant amount of clock cycles, which are 7 and 17 CPU cycles respectively.

Examining the measurement results, it becomes apparent that the BST-based implementation in the case where all tasks have equal release times becomes a fully leaning BST, essentially becoming a linked list. This supports the notion that the retrieval and removal of the first task are indeed constant.

5.3.4 Heap

Inspecting the results of the Heap-based task dispatcher for the set of homogeneous period task sets it is indicated that for the task insertion presented in Figure 5.16, the worst, best and average case have little to no change when the task set is increased. The insertion algorithm follows the standard approach of adding the new item to the last position of the Heap and then moving it up through the Heap until the Heap property is restored. Interestingly, when the input tasks have the same release time, the overhead of the insertion operation does not increase as the number of the inserted tasks grows. When a new item is inserted into a Heap, the Heap's structure must be adjusted to maintain the Heap property, which specifies that parent nodes must always be smaller than or equal to their children. This adjustment process involves swapping the new item with its parent node repeatedly until the Heap property is satisfied. In a scenario where the same release times are being inserted repeatedly into the Heap, the structure of the Heap remains unchanged with respect to the release times for each new insertion. As a result, there are no swaps or adjustments required, and the time it takes to insert a new item into the Heap remains constant. This results in a constant time complexity for the Heap insertion operation. While this is a specific scenario, it highlights the importance of testing for various input scenarios to determine how a system or algorithm performs under different conditions.

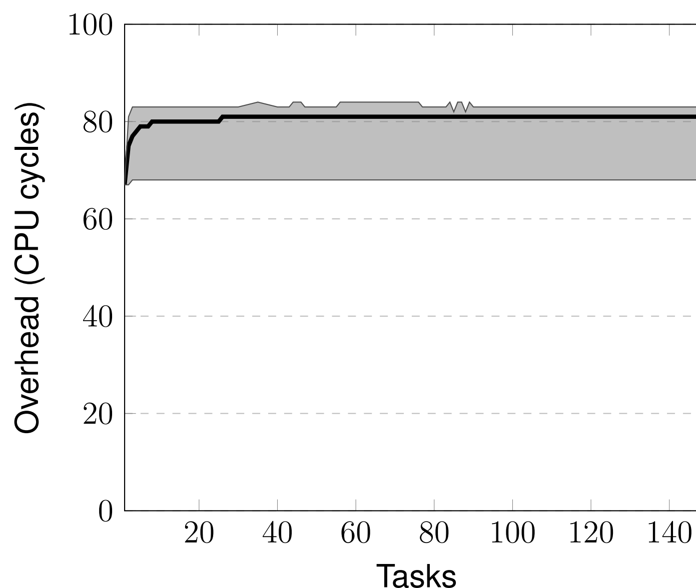

Figure 5.16: Overhead measurements for Heap-based task dispatcher task insertion implementation for homogeneous task sets.

The retrieval of the first task will stay constant at only 1 CPU cycle to get this task.

For the removal of the first task presented in Figure 5.17 we see a similar phenomenon as was seen for the insertion, where the overhead becomes constant at some point. When the root of a Heap is removed, the last element in the Heap is moved to the root position. This creates a temporary violation of the Heap property, as the new root may not be greater than or equal to its child nodes. To resolve this, the Heap's structure is adjusted by repeatedly swapping the new root with its child nodes until the Heap property is satisfied.

However, if the Heap contains the same release times, the Heap's structure remains unchanged after each root removal, as the new root will have the same release time as the previous root. Therefore, there are no swaps or adjustments required, and the time it takes to remove the root remains constant. This is because the Heap's structure is already optimal for the given set of release times, and removing the root does not change the structure in terms of release time order.

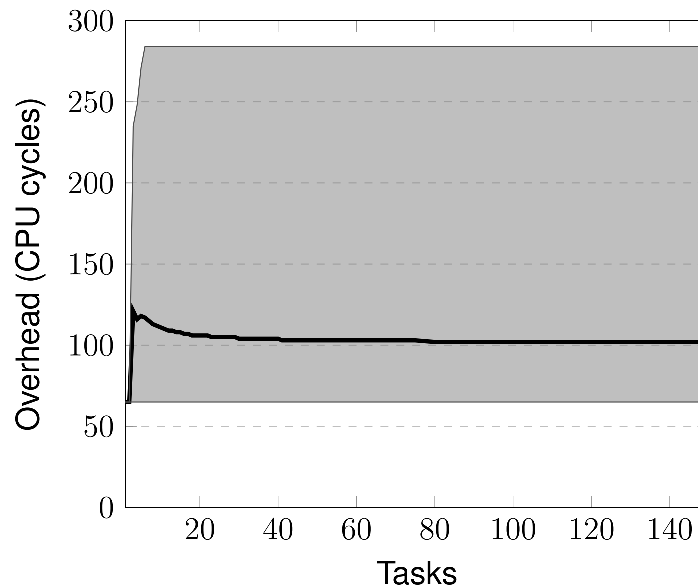

Figure 5.17: Overhead measurements for Heap-based task dispatcher first task removal implementation for homogeneous task sets.

5.3.5 Red-black tree

Examining the computational overhead of the RBT-based task dispatcher w.r.t. the set of homogeneous period task sets, all three of the functionalities (task insertion, first task retrieval and first task removal) result in a clear logarithmic relation between the number of tasks and the computation overhead.

For the insertion, presented in Figure 5.18, there is a clear smooth logarithmic scale for the average and a more logarithmic staircase relation for the worst case. Which is in line with the logarithmic time complexity we expect for the RBT insertion. The smooth logarithmic relation for the average case indicates that the RBTree structure is efficient in maintaining on average a balanced structure as more data is added to the tree. This leads to a continuous and smooth increase in overhead as more tasks are added. However, the staircase-like logarithmic relation for the upper cases suggests that the RBT structure has insertions at larger heights of the tree compared to the average. This can lead to sudden jumps in the overhead when inserting tasks into the RBT-based task dispatcher

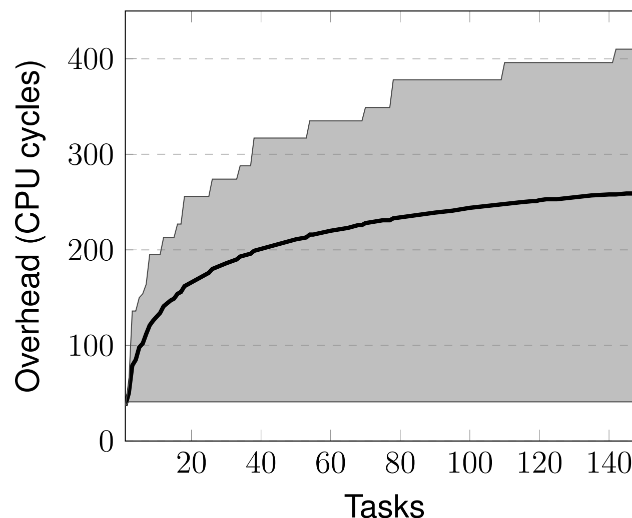

Figure 5.18: Overhead measurements for RBT-based task dispatcher task insertion implementation for homogeneous task sets.

The function for the retrieval of the first task, presented in Figure 5.19, is also given in a more logarithmic staircase relation. This is due to the fact that it is only dependent on the height of the tree on the left side of the RBT. When this height (on the left side) increases the retrieval of the first task also increases. Examining the removal of the first task, presented in Figure 5.20, Also a logarithmic scale can be seen with some more clear outliers. These outliers remain throughout very large execution times and are not incidental.

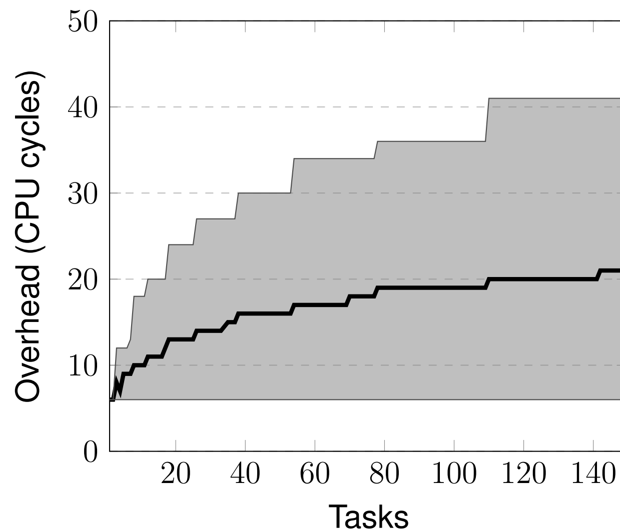

Figure 5.19: Overhead measurements for RBT-based task dispatcher first task retrieval implementation for homogeneous task sets.

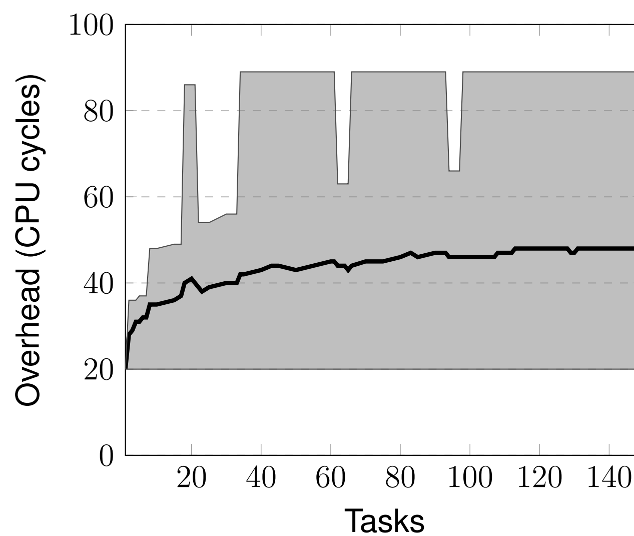

Figure 5.20: Overhead measurements for RBT-based task dispatcher first task removal implementation for homogeneous task sets.

5.3.6 Comparison of Homogeneous Period Task Sets Overhead

In this subsection, we analyze the computation overhead of the different task dispatcher implementations based on the List, BoI, BST, RBT and Heap data structures for a set of task sets with homogeneous periods (every period is equal for a set with a size of 1 up to 150). The objective is to compare the performance of these task dispatcher implementations by considering task insertion, first task retrieval and first task removal operations.

With respect to task insertion, presented in Figure 5.21, the Heap-based task dispatcher excels (after approximately 18 tasks, over the list) converging to constant time due to the implementation which favors homogeneous task periods. Comparing the RBT-based task dispatcher to that of the List-based, we can see that in the average case, the RBT-based task dispatcher outperforms the List-based task dispatcher at around 50 tasks. The BST-based task dispatcher remains underwhelming, as it essentially behaves like a linked list with additional overhead. Interestingly, the average performance of the List and BoI are quite similar, exhibiting the same slope in Figure 5.21 and of course utilizing the same functions for first task retrieval and removal. However, the worst-case performance of the BoI for this set of task sets is notably poor and increases significantly, while its average performance remains acceptable. The performance of both the RBT and Heap-based task dispatcher implementations exhibits lower variability, as evidenced by the closer proximity of their best and worst-case execution times. This contrast is particularly notable when compared to the more substantial disparities observed in the computation overhead of the List-based task dispatcher's insertion functionality.

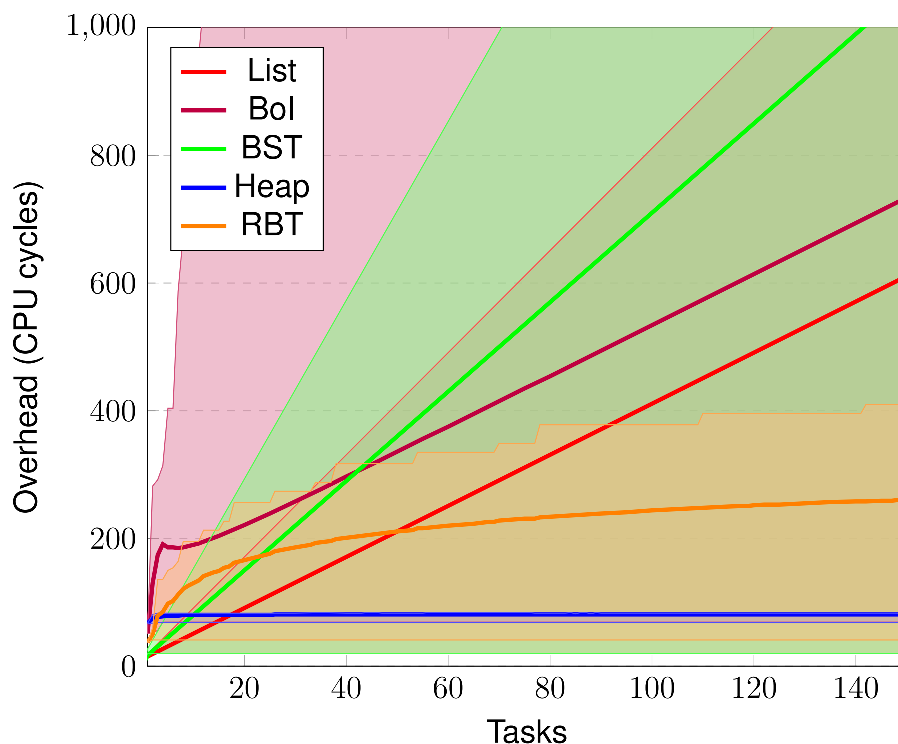

Figure 5.21: Overhead comparison of different task dispatcher task insertion implementations for homogeneous task sets.

In terms of the first task retrieval, presented in Figure 5.22, all task dispatcher implementations, except for the RBT-based task dispatcher, exhibit constant-time behavior, which would be optimal considering the frequency of this operation compared to the insertion and first task removal executions. However, in the case of the fully homogeneous task set, the overlap is 100%, resulting in an almost equal ratio as is described in Section 4.1 ($n_{RemovalCalls} \approx n_{InsertionCalls} \approx n_{RetrievalCalls}$). The significance of this observation is that we can leverage these insights to estimate the total computational overhead when these three functions are combined. By understanding the performance characteristics of each individual function we can better predict the overall system performance, thereby enabling more informed decisions for system design and optimization.

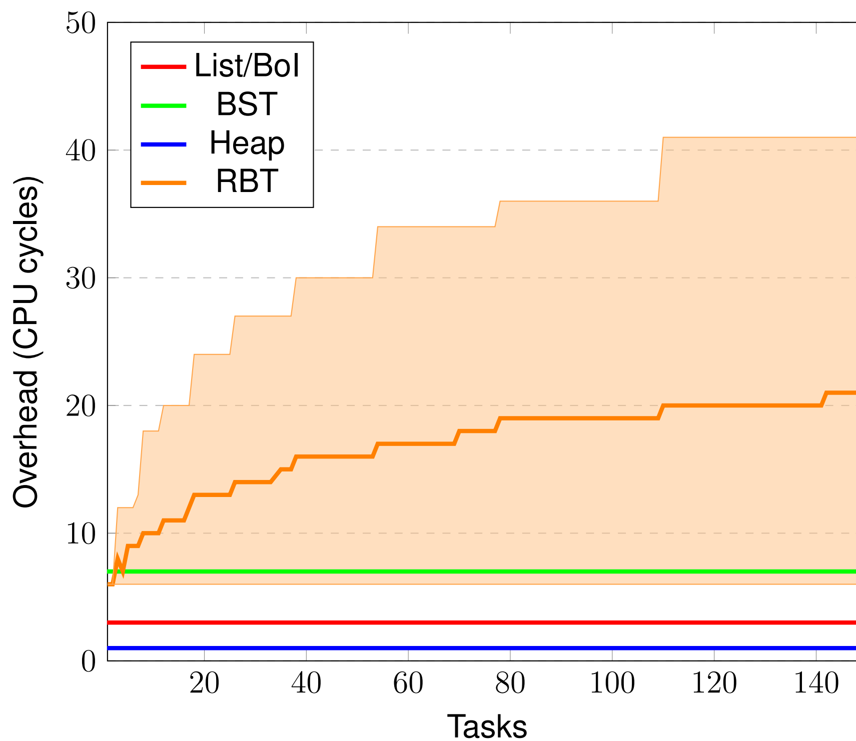

Figure 5.22: Overhead comparison of different task dispatcher first task retrieval implementations for homogeneous task sets.

For first task removal, presented in Figure 5.23, all implementations behave on average in constant time or converge to constant time for homogeneous task sets, again except for the RBT-based task dispatcher. This increment is however a logarithmic function with low incrementation, with an average increment of only 2 CPU cycles for the last 50 task sets.

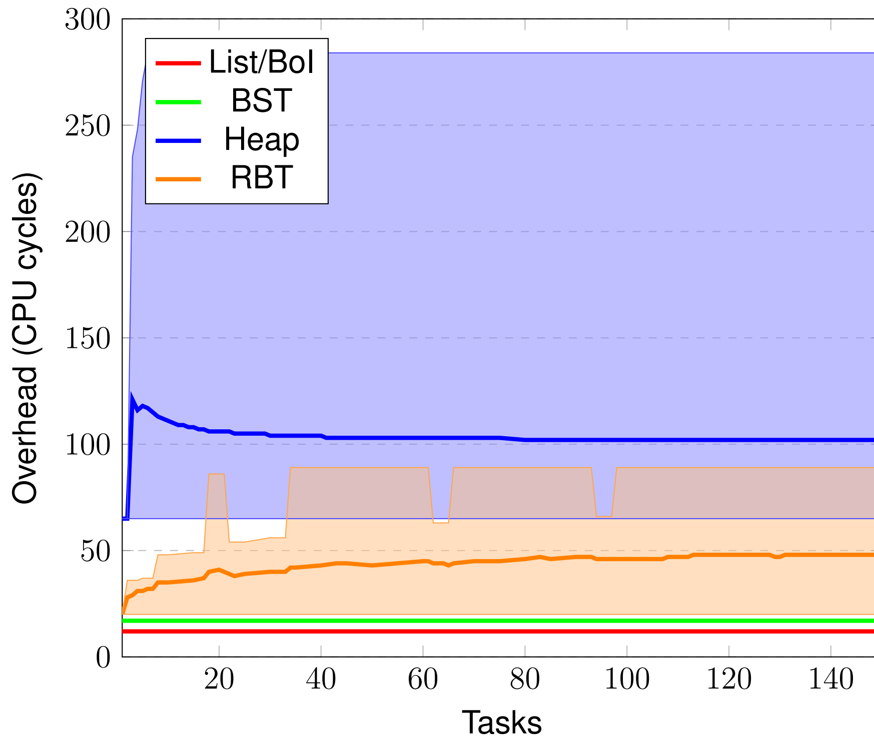

Figure 5.23: Overhead comparison of different task dispatcher first task removal implementations for homogeneous task sets.

Combining all operations and taking into account that insertion, first task retrieval and first task removal occur approximately equally often in a fully homogeneous task set ($n_{RemovalCalls} \approx n_{InsertionCalls} \approx n_{RetrievalCalls}$), we can compare the overall performance of these task dispatchers for this case. As a result, the Heap-based task dispatcher outperforms the List-based task dispatcher in terms of computation overhead at approximately 25 tasks. Similarly, the RBT-based task dispatcher surpasses the List-based task dispatcher in computation overhead at approximately 65 tasks for these specific task sets. Other implementations do not outperform the List-based task dispatcher implementation in terms of computation overhead for a fully homogeneous task set.

5.4 Task Sets with Uniform Period Distribution

The following results are more heavily based on the different task periods that are given in the Real World Automotive Benchmark by Kramer et al. [1]. The benchmark gives 9 different periods used in the automotive industry presented in Table 5.2. Due to the limitation of the embedded device described in Chapter 4, we can only range the uniform distribution from 1 to 16 tasks per period.

In this task set evaluation, it's important to remember that the graph's lower opacity area around data points represents the range between the best and worst performance for the specific task set, with the line in between indicating the average execution times. As discussed previously, the size of this area provides insight into the consistency and predictability of the task dispatcher's performance.

To provide a more comprehensive representation of the data points, markers have been included in addition to the plot line. This is particularly important given the limited number of data points available, with only 16 data points per implementation. To prevent over-cluttering of the graph, markers have only been used for the average case data points and not for the outliers (i.e., the best and worst case data points).

The result of the task insertion for the set of uniform period distribution task sets is presented in Figure 5.24. Here we can clearly see that for task insertion the List-based task dispatcher, for a lower number of tasks remains the optimal implementation compared to the other implementations on average. We can however also see that for a higher number of tasks per period, the performance of the List-based task dispatcher diminishes comparing the average overhead to the BoI, Heap and RBT-based task dispatchers.

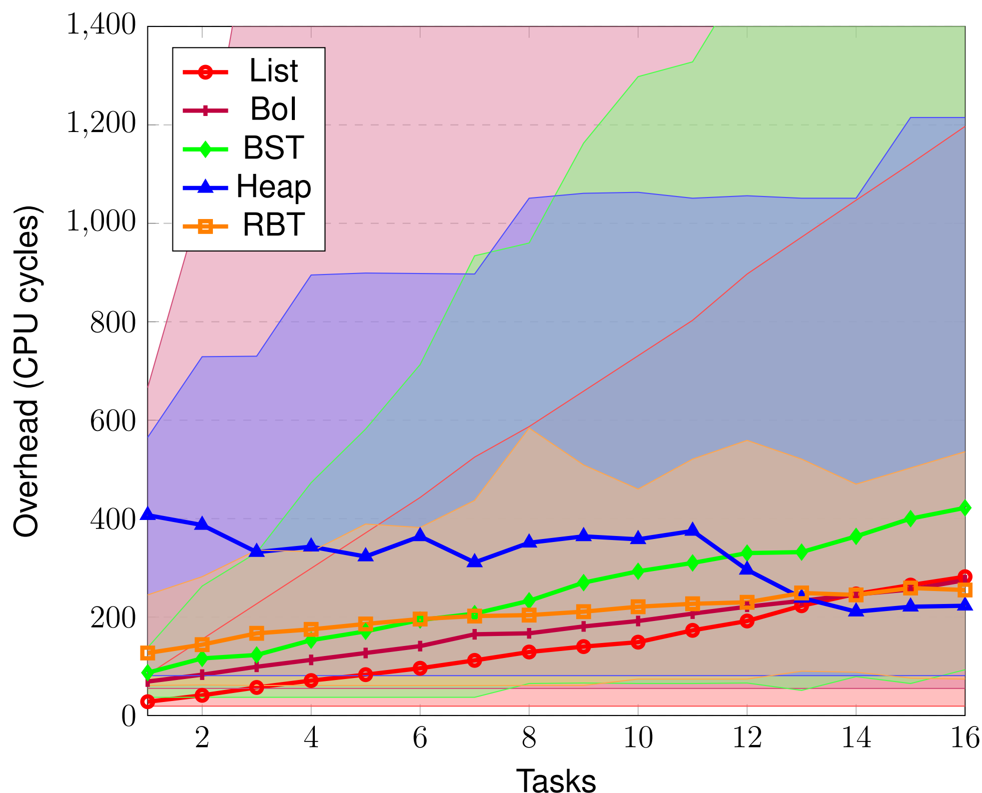

Figure 5.24: Overhead comparison of different task dispatcher task insertion implementations for uniform task sets.

Moreover, when examining the worst-case overhead of the RBT-based implementation, it was observed that the overhead was more consistent with the average overhead compared to the other implementations. This suggests that the RBT implementation was less affected by outliers and had more stable performance.

The retrieval of the first task from the various task dispatchers for the uniform period task distribution is presented in Figure 5.25. The overhead of the retrieval functionalities is still rather low compared to the insertion and removal for the task dispatchers. For the List / BoI and Heap-based dispatchers they remain constant, while the RBT and BST implementations slightly increase over time with respect to their average and worst-case computation overhead.

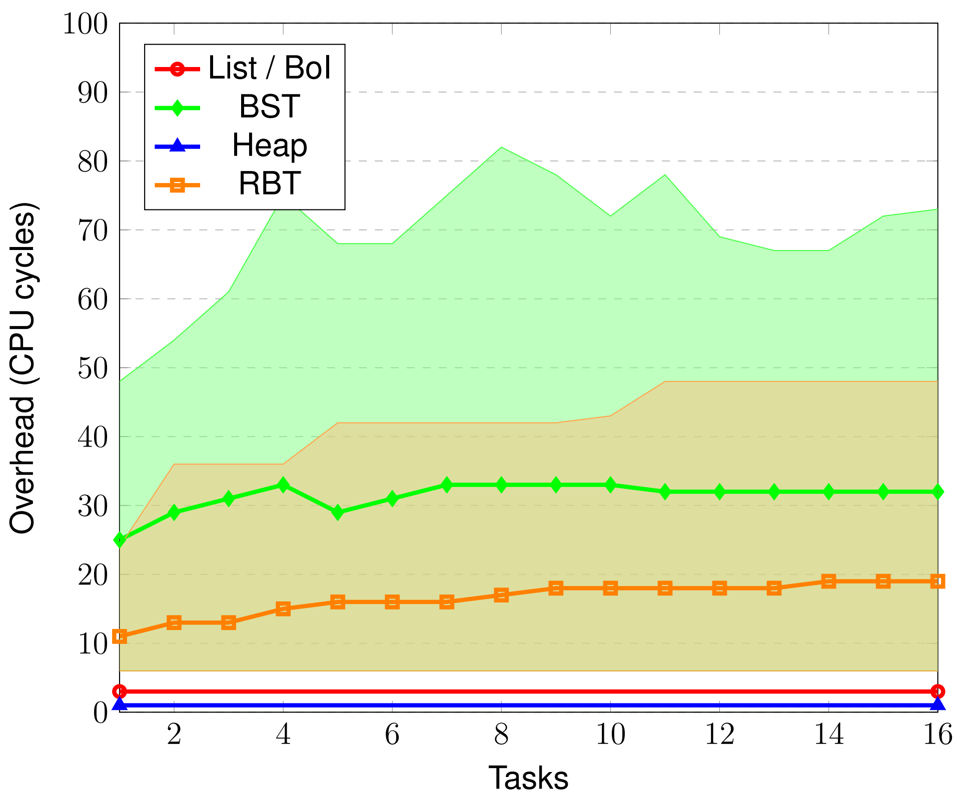

Figure 5.25: Overhead comparison of different task dispatcher first task retrieval implementations for uniform task sets.

For the removal of the first task from the dispatcher based on the different data structures, presented in Figure 5.26, all dispatcher implementations remain low in overhead, except for the Heap based dispatcher. This is due to the reordering of the Heap while preserving the Heap property. The List and BoI implementations remain constant of course and the RBT and BST increase slightly over time because of their logarithmic scale if the height of the tree is preserved supported by the fact that the overhead for each iteration is rather low.

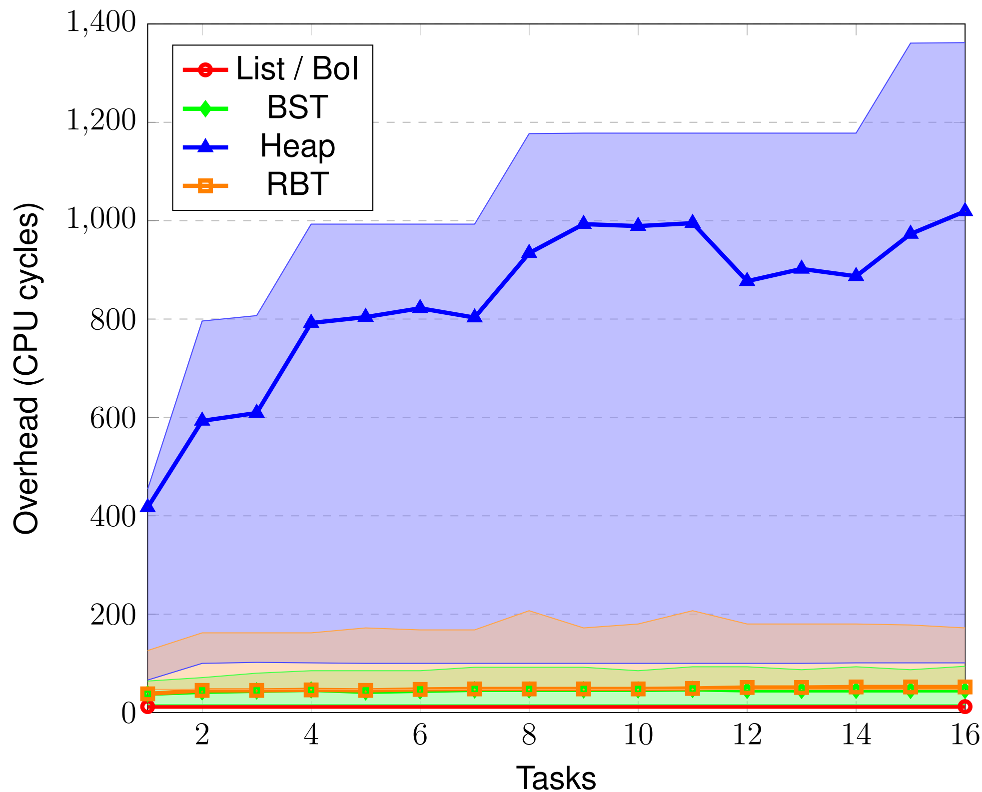

Figure 5.26: Overhead comparison of different task dispatcher first task removal implementations for uniform task sets.

Comparing the results for all of the task dispatchers for the uniform period task distribution, we can state that for 16 tasks per period (144 total tasks), the List and RBT-based task dispatchers have almost equal performance, since the ratio of retrieval of the first task is slightly higher, the List-based task dispatcher does keep the best average performance for this size, by just a few CPU cycles. However, if we look at the difference between the best and worst performance (jitter), we can see that, even though for the List-based task dispatcher first task retrieval and removal are constant, the RBT-based task implementation, combining all the functionalities, has a lower difference (almost half).

5.5 Automotive Benchmark Period Distribution

Given the constraints of the embedded device (ESP32-S3-DevKitC-1), there is a limit to the tasks that can be run simultaneously. This limitation means that the real automotive benchmark distribution of periods in Table 5.1 can only allocate one (1) task per percentage point. If we want to increase this to a higher value where the outcome would be an integer value for each task, the embedded device used in this research would not suffice in terms of memory. This constraint results in one measurement result per task dispatcher implementation for this task distribution, which is given in Table 5.7.

During task insertion, the List-based task dispatcher is revealed to be the most efficient in terms of average computation overhead (181), outperforming the BoI- (226), BST- (277), Heap- (344), and RBT-based task dispatchers (219). In the worst-case scenario, however, the RBT-based task dispatcher exhibits the lowest overhead (474), thereby suggesting more stability compared to the other methods. Furthermore, the BoI-based task dispatcher resulted in the highest worst-case scenario overhead (6543), while indicating a comparatively good average-case performance (especially when considering first task retrieval (3) and first task removal (12) overheads into account). Both the stable performance of the RBT-based task dispatcher and the unstable performance of the BoI-based task dispatcher, again underline the importance of evaluating not only the average-case performance but also considering the implementation's worst-case performance.

The respective overheads for first task retrieval remain comparatively low to those of the task insertion. With the List- (3), BoI- (3) and Heap-based task dispatchers (1) demonstrating their $O(1)$ time complexity, remaining static in terms of overhead.

Regarding first task removal, again both List- and BoI-based task dispatchers showcase their $O(1)$ complexity stability. In stark contrast, the Heap-based task dispatcher exhibits substantial overhead, peaking at 1176 in the worst-case scenario and averaging at 946.

In light of the obtained results from the scaled-down automotive benchmark period distribution, it can be concluded that for this specific task set, the List-based task dispatcher tends to be the most efficient in terms of the average-case computation overhead. However, shifting the focus towards minimizing worst-case overhead paints a slightly different picture.

Upon reviewing the worst-case performance of the RBT-based task dispatcher and taking into account the weighing of task insertion, first task retrieval, and first task removal calls described in Section 4.1 ($n_{RemovalCalls} \approx n_{InsertionCalls}$ and $1 \cdot n_{RemovalCalls} \leq n_{RetrievalCalls} \leq 2 \cdot n_{RemovalCalls}$), we observe a computation range of between 699 and 747 CPU cycles. This range is marginally less than the List-based task dispatcher's range, which lies between 747 and 750 CPU cycles.

Such observations lead us to an intriguing insight into the stability of single execution performance, wherein the RBT-based task dispatcher exhibits superior stability compared to its List-based counterpart. Therefore, while the average-case performance would suggest a preference for the List-based dispatcher, consideration of the worst-case scenario and execution stability leans more favorably toward the RBT-based task dispatcher.

Table 5.7: Scaled down automotive distribution measurement results given in CPU cycles [1].

	Task Insertion			First Task Retrieval			First Task Removal		
	Best	Worst	Average	Best	Worst	Average	Best	Worst	Average
List	19	732	181	3	3	3	12	12	12
BoI	55	6543	226	3	3	3	12	12	12
BST	20	1200	277	6	72	28	16	92	40
Heap	81	1064	344	1	1	1	100	1176	946
RBT	61	474	219	6	48	17	47	177	54

Furthermore, considering a device with greater memory, our results would be even more beneficial. A higher-capacity device would allow us to increase the number of tasks per percentage point, leading to a more granular task distribution. The increase in tasks per percentage point would ideally stretch to 1500 tasks. This would offer us a more precise understanding of the task dispatcher performance for the real automotive benchmark [1]. For example, we could explore how their efficiency evolves as the number of tasks increases.

Chapter 6

Conclusion

In this thesis, we have examined the impact of task set characteristics on the efficiency of our implemented task dispatchers in real-time operating systems, focusing on FreeRTOS. Our research compared List, BST RBT, Heap, and BoI-based task dispatchers, identifying their respective strengths and weaknesses across various scenarios. This highlights the critical role of task set properties in determining dispatcher performance and underscores the importance of tailoring task dispatchers to specific system requirements.

We addressed a key limitation of previous research by utilizing a real-world experimental setup. This enabled us to analyze task dispatcher behavior under practical constraints and provide valuable insights for system designers and developers, while guiding future research in the field. Our findings contribute to existing knowledge about task dispatcher performance and can help develop more efficient real-time operating systems.

Key Findings of our research revealed several important insights into the performance of various task dispatchers in FreeRTOS. The currently used List-based task dispatcher performs well on average for smaller task sets, exhibiting lower overhead compared to other implementations. However, as the task set size increases, its efficiency diminishes. In contrast, the RBT-based task dispatcher demonstrates comparatively good performance for larger task sets and, notably, exhibits a very low difference between the worst and best case (jitter) for the sets of task sets examined in our study. This highlights the RBT's consistency and suitability. Additionally, the Heap-based task dispatcher excels in homogeneous task sets, further emphasizing the importance of selecting the appropriate task dispatcher based on the specific characteristics of the task set at hand. Our research findings also suggest that as the task set size increases beyond those tested in this study, the performance advantage of the RBT-based task dispatcher over the List-based implementation is likely to become even more pronounced. This projected trend is particularly relevant

for real-world applications, such as the benchmark that was discussed and used for our evaluation, where task sets may range from approximately 1000 to 1500 tasks.

Limitations of our research include the inability to measure larger task sets, due to the constraints of the small embedded device used in our experiments, and the lack of diversity in task set periods. These limitations may impact the generalizability of our findings and warrant further research with larger task sets and more diverse period distributions. Testing more diverse task sets might reveal other factors that contribute to the preferred dispatcher implementation and shed light on the specific conditions under which each method is best suited.

Real-world implications of our research emphasize the potential benefits of optimizing task dispatchers in real-time systems. Improved task dispatcher performance can lead to better overall performance in specific scenarios, including higher tick frequencies and enhanced deadline guarantees. This can impact system designers, developers, and vendors in the field of real-time systems, and encourage FreeRTOS and other vendors to investigate further optimization of task dispatchers.

Future work should address our research limitations by increasing task set sizes e.g. to 1000-1500 tasks, which is more representative of real-world automotive applications. This can be achieved by using a more powerful embedded device with a larger memory capacity, enabling more accurate performance assessments of task dispatching implementations in realistic systems. Integrating a timing wheel into the FreeRTOS kernel has proven challenging due to fundamental differences in design. This would involve understanding the existing codebase, implementing dynamic insertion and removal of tasks, efficient memory use, and rigorous testing and verification. Fundamentally different approaches to task dispatching may offer increased performance, e.g. one approach involves using a per-task counter in the dispatcher and consolidating tasks with identical periods. This could transfer the overhead from dispatch checks to the actual process and potentially enable multiple dispatches in a single period

In summary, our research adds value to the field by offering a deeper understanding of the impact of task dispatchers on real-time system performance and promoting further exploration and optimization of task dispatcher designs. We hope that our findings will encourage continued research and development in this area, ultimately leading to better and more efficient real-time systems.

Bibliography

[1] S. Kramer, D. Ziegenbein, and A. Hamann, “Real world automotive benchmarks for free,” in *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, vol. 130, 2015.

[2] W. Hofer, D. Danner, R. Müller, F. Scheler, W. Schröder-Preikschat, and D. Lohmann, “Sloth on time: Efficient hardware-based scheduling for time-triggered rtos,” in *2012 IEEE 33rd Real-Time Systems Symposium*. IEEE, 2012, pp. 237–247.

[3] P. Kohout, B. Ganesh, and B. Jacob, “Hardware support for real-time operating systems,” in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003, pp. 45–51.

[4] Y. Li, X. Liu, Y.-f. Ding, H.-x. Cui, Y.-b. Du, and Y. Li, “An improvement of task scheduling algorithms and hardware scheduler of real-time operating system,” *International Journal of Hybrid Information Technology*, vol. 7, no. 3, pp. 337–344, 2014.

[5] G. Varghese and A. Lauck, “Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility,” *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 824–834, 1997.

[6] M. Short, “Improved task management techniques for enforcing edf scheduling on recurring tasks,” in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2010, pp. 56–65.

[7] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[8] Q. Li and C. Yao, *Real-time concepts for embedded systems*. CRC press, 2003.

[9] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, no. 1, p. 46–61, jan 1973. [Online]. Available: https://doi.org/10.1145/321738.321743

[10] A. Phillip, J. L. Seppo *et al.*, "Real-time systems design and analysis: Tools for the practitioner," *ISBN: 978–0470768648*, 2012.

[11] A. McPherson, B. Proffitt, and R. Hale-Evans, "Estimating the total development cost of a linux distribution," *The Linux Foundation*, vol. 198, no. 198, p. 198, 2008.

[12] Torvalds, "Linux/timerqueue.c," Apr 2020. [Online]. Available: https://github.com/torvalds/linux/blob/master/lib/timerqueue.c

[13] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 3–11.

[14] "Rtems/rtems: Realtime smp kernel, networking, file-systems, drivers, bsps, samples, and testsuite." [Online]. Available: https://github.com/RTEMS/rtems

[15] "Kernel timing," Jun 2022. [Online]. Available: https://docs.zephyrproject.org/3.1.0/kernel/services/timing/clocks.html

[16] F. Guan, L. Peng, L. Perneel, and M. Timmerman, "Open source freertos as a case study in real-time operating system evolution," *Journal of Systems and Software*, vol. 118, pp. 19–35, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300383

[17] "Market leading rtos (real time operating system) for embedded systems with internet of things extensions," Dec 2022. [Online]. Available: https://www.freertos.org/

[18] R. Kase, "Efficient scheduling library for freertos," 2016.

[19] "Microcontrollers and compiler tool chains supported by freertos," Jan 2022. [Online]. Available: https://www.freertos.org/RTOS_ports.html

[20] "Freertos demo applications," Jan 2022. [Online]. Available: https://www.freertos.org/a00102.html

[21] "Github freertos-kernel." [Online]. Available: https://github.com/FreeRTOS/FreeRTOS-Kernel

[22] R. Barry. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

[23] "Esp-idf, official iot development framework." [Online]. Available: https://www.espressif.com/en/products/sdks/esp-idf#:~:text=ESP%2DIDF%20is%20Espressif's%20official,as%20C%20and%20C%2B%2B.

[24] "Freertos (overview)." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html

[25] PlatformIO, "A professional collaborative platform for embedded development." [Online]. Available: https://platformio.org/

[26] H. Högl and D. Rath, "Open on-chip debugger–openocd–," *Fakultat fur Informatik, Tech. Rep*, 2006.

[27] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 675, 1988.

[28] "Jtag debugging." [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-guides/jtag-debugging/index.html

[29] M. Ebbrecht, K.-H. Chen, and J.-J. Chen, "Bucket of ignorance: A hybrid data structure for timing mechanism in real-time operating systems."

[30] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010, pp. 6–11.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[32] R. Sedgewick and K. Wayne, *Algorithms*. Pearson Education, 2011.

[33] R. Wiener, "Generic red-black tree and its c# implementation." *J. Object Technol.*, vol. 4, no. 2, pp. 59–80, 2005.

[34] K. Mehlhorn, P. Sanders, and P. Sanders, *Algorithms and data structures: The basic toolbox*. Springer, 2008, vol. 55.

[35] J. Morris, *8.2 Red-Black Trees*. John Morris, 1998.

[36] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data structures and algorithms in Java*. John wiley & sons, 2014.

[37] R. Sedgewick, "Left-leaning red-black trees," in *Dagstuhl Workshop on Data Structures*, vol. 17, 2008.

[38] N. Deo and S. Prasad, "Parallel heap: An optimal parallel priority queue," *The Journal of Supercomputing*, vol. 6, pp. 87–98, 1992.

[39] D. S. JOHNSON, "Chapter 2 - a catalog of complexity classes," in *Algorithms and Complexity*, ser. Handbook of Theoretical Computer Science, J. VAN LEEUWEN, Ed. Amsterdam: Elsevier, 1990, pp. 67–161. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444880710500072

[40] T. L. Foundation, "Linux kernel developer: Thomas gleixner," Sep 2022. [Online]. Available: https://www.linuxfoundation.org/blog/blog/linux-kernel-developer-thomas-gleixner

[41] T. Gleixner and I. Molnar, "The linux kernel." [Online]. Available: https://docs.kernel.org/next/timers/hrtimers.html

[42] M. Ebbrecht, "Benchmarking timer mechanisms in real-time operating systems," Ph.D. dissertation, Technical University Dortmund, 2022.

[43] Libuv, "Libuv/heap-inl.h," May 2014. [Online]. Available: https://github.com/libuv/libuv/blob/v1.x/src/heap-inl.h

Appendix A

Pseudo code implemented data structure functions

A.1 List

A.1.1 Insertion

Algorithm 1 Insert a node into a sorted doubly linked list

Require: A pointer to the tail of a sorted doubly linked list z and a pointer to the node to be inserted x.

Ensure: The node pointed to by x has been inserted into the list in sorted order.

procedure LISTINSERT(x, z)
  $y \leftarrow z.previous$
  while $y \neq z$ and $y > x$ **do**
    $y \leftarrow y.previous$
  end while
  $x.next \leftarrow y.next$
  $x.next.previous \leftarrow x$
  $x.previous \leftarrow y$
  $y.next \leftarrow x$
end procedure

A.1.2 Remove First Task

Algorithm 2 Remove a node from a doubly linked list

Require: A pointer to the node to be removed x

Ensure: The node pointed to by x has been removed from the list and the list remains sorted.

```
procedure LISTREMOVE(x)
    x.next.previous ← x.previous
    x.previous.next ← x.next
end procedure
```

A.2 Bucket of Ignorance

Algorithm 3 Insert a node into the Bucket of Ignorance [29] [42]

Require: A pointer to the node to be inserted x, a pointer to the (tail of the) sorted linked list y and a pointer to the unsorted bucket z.

Ensure: The node pointed to by x has been inserted in the list y or z, where y is ordered with the first expiry times compared to y.

```
procedure BOIINSERT(x, y, z)
    if z is empty then
        ListInsertEnd(x, z)
    else
        if y is empty then
            if x < z.head then
                ListInsert(x, y)                    ▷ Algorithm 1
            else
                ListInsertEnd(x, z)                 ▷ Algorithm 1 but to the end of the list
                Refill(y, z)                        ▷ Algorithm 4
            end if
        else
            if x < z.head then
                ListInsert(x, y)                    ▷ Algorithm 1
            else
                ListInsertEnd(x, z)                 ▷ Algorithm 1 but to the end of the list
            end if
        end if
    end if
end procedure
```

Algorithm 4 Refilling the Bucket of Ignorance [29] [42]

Require: A pointer to the (tail of the) sorted linked list y and a pointer to the unsorted bucket z.

Ensure: The sorted list y is refilled with the unsorted list z,based on the splitting point s

```
procedure REFILL(y, z)
    MergeSort(z)
    s ← SizeOf(z)/2
    for i = 0, i < s, i++ do
        e_min ← z.Head
        ListInsertEnd(z.head, y)          ▷ Algorithm 1 but to the end of the list
        ListRemove(z.head)                ▷ Algorithm 2
    end for
    for i = 0, i < s, i++ do
        if e_min == z.head then
            ListInsertEnd(z.head, y)      ▷ Algorithm 1 but to the end of the list
            ListRemove(z.head)            ▷ Algorithm 2
        else
            break
        end if
    end for
end procedure
```

A.3 Red-black Tree

Algorithm 5 Left-Rotate operation in a Red-Black tree

Require: A pointer to the root of a Red-Black tree $root$ and a pointer to the node that it should be rotated left around.

Ensure: The node pointed to by x has been rotated to the left.

procedure ROTATELEFT($root, x$)
  $y \leftarrow x.rightChild$
  $x.rightChild \leftarrow y.leftChild$
  if $y.leftChild \neq NULL$ **then**
    $y.leftChild.parent \leftarrow x$
  end if
  $y.parent \leftarrow x.parent$
  if $x.parent = null$ **then**
    $root \leftarrow y$
  else if $x = x.parent.leftChild$ **then**
    $x.parent.leftChild \leftarrow y$
  else
    $x.parent.rightChild \leftarrow y$
  end if
  $y.leftChild \leftarrow x$
  $x.parent \leftarrow y$
end procedure

Algorithm 6 Fix up a Red-Black tree after insertion

Require: A pointer to the root of a Red-Black tree $root$ and a pointer to a newly inserted node z.

Ensure: The Red-Black tree is fixed up so that it satisfies the Red-Black tree properties.

procedure FIXUP($root, z$)
  while $z.parent \neq null$ and $z.parent.color = red$ **do**
    if $z.parent$ is the left child of its grandparent **then**
      $y \leftarrow$ the right child of z's grandparent
      if y is non-null and y is red **then**
        $z.parent.color \leftarrow black$
        $y.color \leftarrow black$
        $z.grandparent.color \leftarrow red$
        $z \leftarrow z.grandparent$
      else
        if $z = z.parent.rightChild$ **then**
          $z \leftarrow z.parent$
          ROTATELEFT($root, z$) ▷ Algorithm 5
        end if
        $z.parent.color \leftarrow black$
        $z.grandparent.color \leftarrow red$
        ROTATERIGHT($root, z.grandparent$) ▷ Algorithm 5 but to the right
      end if
    else
      $y \leftarrow z.grandparent.leftChild$
      if $y \neq null$ and $y.color = red$ **then**
        $z.parent.color \leftarrow black$
        $y.color \leftarrow black$
        $z.grandparent.color \leftarrow red$
        $z \leftarrow z.grandparent$
      else
        if $z = z.parent.leftChild$ **then**
          $z \leftarrow z.parent$
          ROTATERIGHT($root, z$) ▷ Algorithm 5 but to the right
        end if
        $z.parent.color \leftarrow black$
        $z.grandparent.color \leftarrow red$
        ROTATELEFT($root, z.grandparent$) ▷ Algorithm 5
      end if
    end if
  end while
  $root.color \leftarrow black$
end procedure

Algorithm 7 Remove the minimum node from a Red-Black tree

Require: A pointer to the root of a Red-Black tree $root$.
Ensure: The minimum node in the Red-Black tree has been removed.

```
procedure RBTreeRemoveMin(root)
    x ← root
    while x.leftChild ≠ null do
        x ← x.leftChild
    end while
    y ← x.parent
    z ← x.rightChild
    if z ≠ null then
        z.parent ← y
    end if
    if y = null then
        root ← z
    else if x = y.leftChild then
        y.leftChild ← z
    else
        y.rightChild ← z
    end if
    if z ≠ null then
        FixUp(root, z)                                  ▷ Algorithm 6
    end if
end procedure
```