Assessment of Efficient Dispatching in FreeRTOS

Florian Hagens and Kuan-Hsun Chen
Department of Computer Science, University of Twente, the Netherlands
f.hagens@student.utwente.nl, k.h.chen@utwente.nl

***Abstract*—This study investigates the efficiency of task dispatchers in real-world implementations. We focus on evaluating various task dispatching methods based on four distinct data structures and their impact on computation overhead and performance in FreeRTOS. By using a real-world setup, we analyze the merits and drawbacks of each data structure and corresponding task dispatcher implementation. Our preliminary findings suggest that task dispatcher efficiency highly depends on the task set size and their respective periods, with alternative dispatchers potentially outperforming the List-based implementation, which is presently utilized in FreeRTOS, in certain scenarios. Ultimately, this study seeks to provide valuable insights for system designers and developers, emphasizing the importance of tailoring task dispatchers to specific task sets for improved efficiency and reliability in real-time systems.**

***Index Terms*—Real-Time Operating Systems, Task Dispatchers**

I. Introduction

Real-time systems demand the efficient and timely execution of periodic tasks to guarantee system stability, responsiveness to time-sensitive events, and predictable behavior in their applications [1]. Although numerous studies have been conducted on scheduling algorithms, the task dispatcher, which plays a crucial role in initiating task execution and maintaining task periodicity, has not been as thoroughly investigated. This early work aims to investigate this gap by presenting a case study on FreeRTOS's task dispatcher, exploring various implementations to assess their respective operation overheads.

Prior work has studied task dispatcher optimization through hardware-based solutions and the development of efficient data structures [2]–[5]. These studies demonstrate the importance of task dispatcher optimization, and lay the groundwork for our current research. However, there is yet no definitive conclusion regarding the most suitable type of task dispatcher to implement in specific scenarios, emphasizing the need for further research and context-driven evaluations to establish best practices in task dispatcher design and implementation.

The task dispatcher plays a critical role in managing tasks with a specific data structure, as it is invoked in every system tick. Upon the occurrence of each system tick interrupt, the task dispatcher checks whether the current tick t is greater than or equal to the next unblock time B, which is defined by the earliest release job in the data structure. If this condition is met ($t \geq B$), the job with the earliest release time (R_{min}) is retrieved from the data structure. Then, the task dispatcher compares R_{min} with the current tick t. If $R_{min} > t$, the unblock time B is updated according to: $B = R_{min}$ However, if $R_{min} \leq t$, the task with the release time R_{min} is removed from the task dispatcher's data structure and placed in the ready queue. In FreeRTOS, the frequency of tick increments is defined by `configTICK_RATE_HZ` (e.g., set to 100, which corresponds to a tick increment every $10\,\mathrm{ms}$). In light of the description above, it is evident that the task dispatcher is invoked frequently, highlighting that even minor improvements in its overhead can lead to substantial reductions in the overall system overhead throughout its lifespan, which is primarily determined by the data structure.

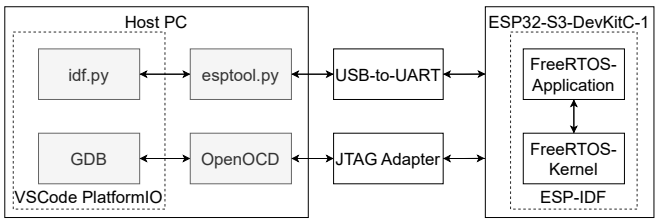


Fig. 1. Real-world measurement setup

Toward this, we evaluate the task dispatcher in FreeRTOS, which is one of the well-known RTOS, based on four data structures: List, Binary Search Tree (BST), Red-Black Tree (RBT), and Heap. BST is selected to strike a balance between time complexity and code simplicity, while Red-Black Trees and Heaps were employed for their superior time complexity performance. We implemented these data structures not only to compare the average computation overhead but also the "jitter" (the difference between worst and best-case performance) within a task set. The jitter here is crucial in ensuring the predictability of the system behavior (e.g., no unexpectedly long delay during the dispatching). Please note that, due to the considerable challenges arising from the design principles, e.g., memory footprint, the integration of a timing wheel has not been implemented [4].

Our Contributions: This paper presents a thorough assessment of each task dispatcher's effectiveness, evaluating computational overhead and offering valuable insights for developers and researchers in the field of embedded systems. A key aspect of our study is the use of real-world measurements on actual hardware, on which we performed CPU cycle measurements as a metric for determining overhead to ensure accurate and realistic behavior of the investigated dispatchers.

The codebase for the kernel, which includes the dispatcher implementations evaluated in this paper, is open for reference upon request to encourage transparency and foster collaboration within the academic community; however, it is not yet publicly available due to its work-in-progress status.

II. Real-World Measurement Setup

In Figure 1, we present the components of our real-world measurement setup, encompassing the software, hardware, and data collection methodologies employed for evaluating the performance of our target system.

A. Software Components

The underlying software platform is based on the ESP-IDF FreeRTOS kernel (FreeRTOS version V10.4.3 and ESP-IDF version 4.4.1), which offers a comprehensive development environment tailored for the ESP32 series of microcontrollers. We use `esptool.py` and `idf.py` as essential command-line utilities for handling firmware-related operations. The `esptool.py` allows us to flash firmware and interact with the ESP32 bootloader, while `idf.py` provides a range of build system and project management capabilities. In order to evaluate the changes made to the task dispatcher, we have developed a dummy FreeRTOS application that creates periodic tasks based on the desired task set, allowing for easy configuration of various parameters, such as periods and execution times.

B. Hardware and Debugging Tools

We employ the ESP32-S3-DevKitC-1 microcontroller as the target embedded device. The microcontroller is configured and managed using the PlatformIO structure, ensuring compatibility with multiple ESP32 devices and facilitating configuration adjustments. To gain insights into the measurement results, we rely on GDB and OpenOCD for debugging and profiling the embedded device. These tools enable effective examination of system performance and aid in understanding the intricacies of the measurement process.

C. Evaluation Methodology

The CPU cycle counter, based on the `ccount` register of the ESP32-S3-DevKitC-1, is employed to assess software system overhead. The CPU cycle counter offers various advantages, such as high-resolution time measurements, low overhead, independence from external factors, and consistency among systems. We track the average, best, and worst-case scenarios over multiple task executions.

To obtain accurate and reproducible results, we disable most compiler optimizations (optimization level -O0), making the measurement outcomes less reliant on the compiler or CPU architecture. This approach, although it may impact performance, provides valuable insights into the characteristics of the task dispatcher and its overhead.

To assess the performance of task dispatchers, we measured the CPU cycles for their three primary operations (i.e., task insertion, first task retrieval, and first task removal) and plotted them to derive visual insight. To ensure the behavioral correctness of the implementations, we restricted the task set to a maximum of 150 tasks.

The task model in our evaluation is strictly periodic, maintained through the use of the `vTaskDelayUntil()` function. Other task characteristics, such as priority and execution time, have no direct influence on the task dispatcher behavior.

III. Evaluation

Firstly, we examined the worst-case scenario of each implementation, ranging from 1 up to 150 tasks per implementation. Note that the worst-case scenario was enforced manually for a single execution of the primary operations, e.g., the longest path in tree-based structures. As shown in Figure 2, the characteristics of different implementations differ significantly.

Note that, for subsequent task set evaluations (Figure 3 and Figure 4), a graph's lower opacity area represents the computation overhead space. A single execution can have any value within this space, and the line between these bounds represents the average computation overhead. A smaller area with lower opacity indicates more consistent performance, while a larger area suggests more variable performance.

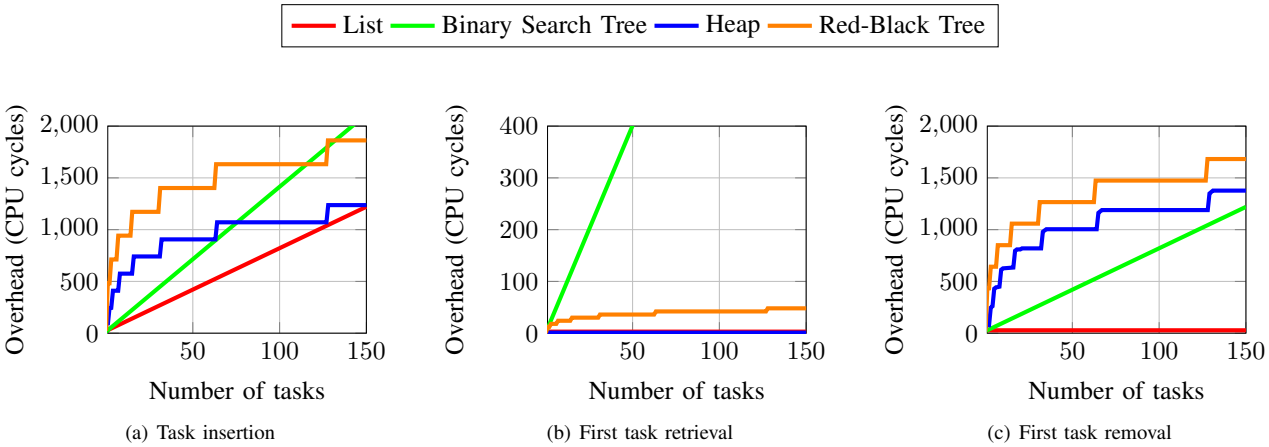
Secondly, we evaluated the homogeneous task sets, where every task had the same period. This aspect could be of interest in automotive industries, where substantial proportions of tasks operate within a limited number of periods [6]. Since the three primary operations exhibit a similar rate of invocation in such task sets, we can describe the computation overhead comparison fairly. As shown in Figure 3, the Heap-based dispatcher and the RBT-based dispatcher outperform the List-based dispatcher, at 25 tasks and 65 tasks, respectively.

Finally, we synthesized task sets, according to the automotive benchmark, provided by Kramer et al. [6]. This benchmark was chosen due to its significance, abstracted from real-world automotive applications, and its period variance. There was a total of 9 different task periods. For every period used in the benchmark, an equal amount of tasks was created. We evaluated 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 144 tasks. In Figure 4, the average performance of the RBT-based dispatcher is found to be comparable to that of the List-based dispatcher for larger task sets; however, the RBT-based dispatcher exhibits a substantially reduced difference, nearly half, between the best and worst performance outcomes.

IV. Conclusion

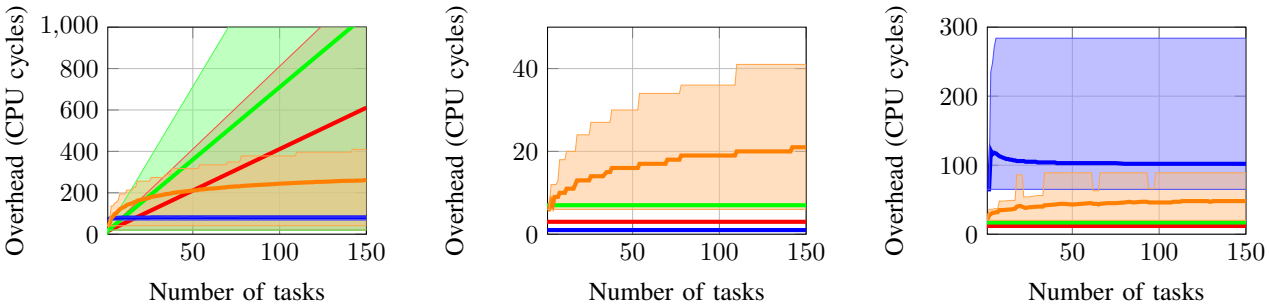
In this work-in-progress, we assess the efficiency of various task dispatchers in FreeRTOS. To examine the overhead incurred by different implementations, we deployed a real-world measurement setup via ESP32-S3-DevKitC-1 and examined the efficiency of different implementations under various configurations. The experimental results show that the performance highly depends on the size of the task sets and their respective periods. Interestingly, we found that RBT and Heap might perform better than the List-based task dispatcher, which is presently utilized in FreeRTOS, in specific scenarios.

Since the specification of real-time systems is often known offline, such as the number and the periods of tasks, we plan to leverage this information to derive tailored implementations automatically. Such specific solutions could be more applicable for industrial applications, as argued in [7].



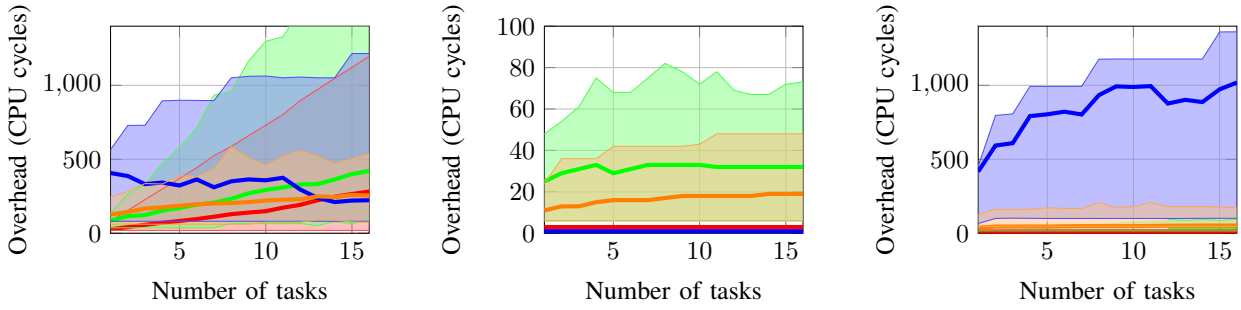
(a) Task insertion (b) First task retrieval (c) First task removal

Fig. 2. Worst-case computation overhead comparison of different task dispatcher implementations.



(a) Task insertion (b) First task retrieval (c) First task removal

Fig. 3. Overhead comparison of different task dispatcher implementations for homogeneous task sets.



(a) Task insertion (b) First task retrieval (c) First task removal

Fig. 4. Overhead comparison of different task dispatcher implementations for uniform distributed task sets.

REFERENCES

[1] 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 *IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 3–11.

[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 *33rd Real-Time Systems Symposium*, 2012, pp. 237–247.

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

[4] 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.

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

[6] 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*, vol. 130, 2015.

[7] G. v. der Brüggen, A. Burns, J.-J. Chen, R. I. Davis, and J. Reineke, "On the trade-offs between generalization and specialization in real-time systems," in *IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2022, pp. 148–159.