PROCEEDINGS OF

OSPERT 2023

The 17th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 11th, 2023 in Vienna, Austria

in conjunction with

ECRTS

The 35th Euromicro Conference on Real-Time Systems
July 11–14, 2023, Vienna, Austria

Editors:
Renato Mancuso
Alexander Zuepke

Contents

Message from the Chairs

Welcome to OSPERT'23, the 17th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. This year, OSPERT will provide a combined program with the RT-Cloud workshop. We invite you to join us in participating in a workshop of lively discussions, exchanging ideas about systems issues related to real-time and embedded systems.

The workshop will open with a keynote by Ulrich Drepper, discussing co-development of hardware and software at the Red Hat CoDes lab at Boston University. We will have a second keynote from the RT-Cloud workshop in the afternoon.

OSPERT'23 received four submissions from which all were selected by the program commitee to be presented at the workshop. Each paper received three individual reviews. Our special thanks go to the program committee, a team of nine experts for volunteering their time and effort to provide useful feedback to the authors, and of course to all the authors for their contributions and hard work.

OSPERT'23 would not have been possible without the support of many people. The first thanks are due to Alessandro Papadopoulos, Peter Puschner, and the whole ECRTS organizing team for entrusting us with organizing OSPERT, and for their continued support of the workshop. We would also like to thank the chairs of prior editions of the workshop who shaped OSPERT and let it grow into the successful event that it is today.

Last, but not least, we thank you, the audience, for your participation. Through your stimulating questions and lively interest you help to define and improve OSPERT. We hope you will enjoy this day.

The Workshop Chairs,

Renato Mancuso	Alexander Zuepke
Boston University	Technische Universität München
USA	*Germany*

Program Committee

Harini Ramaprasad *UNC Charlotte*
Bryan Ward *Vanderbilt University*
Daniel Casini *Scuola Superiore Sant'Anna*
Francesco Restuccia *Northeastern University*
Christian Dietrich *Technical University of Hamburg*
Gedare Bloom *University of Colorado at Colorado Springs*
Arpan Gujarati *University of British Columbia*
Catherine Nemitz *Davidson College*
Marine Sauze-Kadar *CEA-Leti*

Keynote Talk

Co-Developing Hardware and Software

Ulrich Drepper
Distinguished Engineer, Red Hat Research

The granularity at which hardware SKUs are available usually means that one uses a more-or-less general development platform. On top of this, at best, a customized real-time OS is deployed, which due to portability, is written with compromises in the HAL and user API.

In the Red Hat CoDes lab at Boston University, we are developing solutions that allow specifying the exact needs of the program(s) to run and which creates from this specification and the program code everything from the exact specification of the hardware (how many cores, what ISA, what extensions, what interfaces like DDR, Ethernet, PCIe, SPI, GPIO, etc) to the bootloader, debugger interfaces, OS with standard interfaces and further on to an integrated development environment for the software developer.

The goal is to create an efficient platform to deploy the application with complexity a developer already handles fine on hardware based on FPGAs, which allows a common hardware design used in many situations. All this, of course, with fleet management, documents, and design security models.

This talk will give a short overview of the current state and what we hope to achieve.

Ulrich Drepper joined Red Hat again in 2017 after a seven-year hiatus when he worked for Goldman Sachs. He works in the Red Hat Research group, which is part of the office of the CTO. As part of his job, he looks after the ongoing projects the group is working on. He specifically concentrates on developing new technologies for efficient computing for high-performance needs or limited energy budgets with the help of customized hardware/software solutions.

In his last position at Goldman Sachs, he worked on various areas, such as stochastic algorithms to aid in operation, consulting internally on high-performance and low-latency development, and teaching classes around computing fundamentals and machine learning.

His previous stint at Red Hat lasted 14 years. The last position was as a member of the office of the CTO to collect and disseminate information relevant to the Red Hat Enterprise Linux product, predominantly in the high-performs area. During this time and back to the earliest days of Linux, he developed the basic runtime and development tools still in use today.

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.

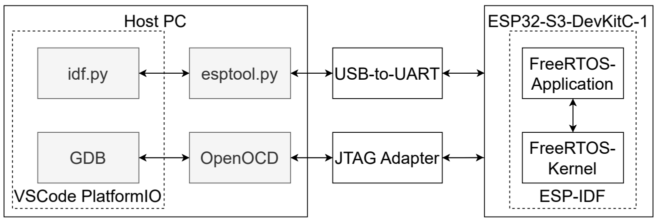


Fig. 1. Real-world measurement setup

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.

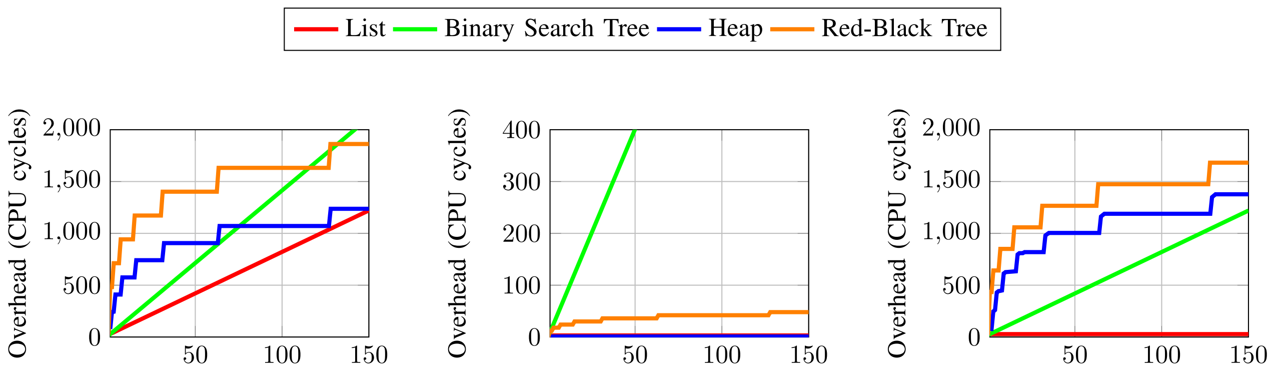
Secondly, we evaluated the homogeneous task sets, where every task had the same period. This aspect could be of interest in automotive industries, where substantial proportions of tasks operate within a limited number of periods [6]. Since the three primary operations exhibit a similar rate of invocation 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

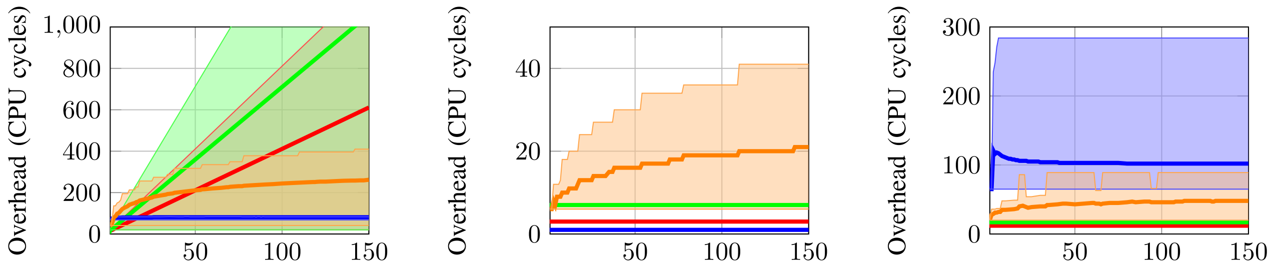
In this work-in-progress, we assess the efficiency of various task dispatchers in FreeRTOS. To examine the overhead incurred by different implementations, we deployed a real-world measurement setup via ESP32-S3-DevKitC-1 and examined the efficiency of different implementations under various configurations. 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].



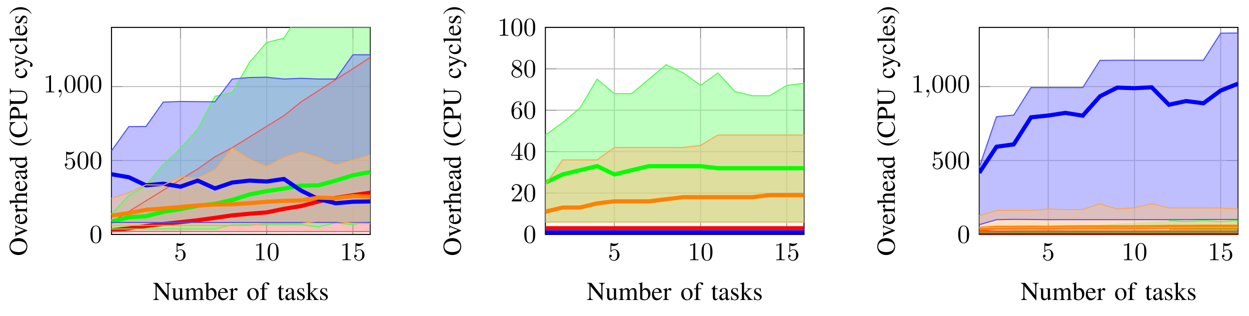
(a) Task insertion (b) First task retrieval (c) First task removal

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.

ResourceGauge: Enabling Resource-Aware Software Components

Andreas Schmidt[1] [0000-0002-7113-7376] Luis Gerhorst[2,3] [0000-0002-3401-430X]
Kai Vogelgesang[1] [0000-0002-7633-1880] Timo Hönig[3] [0000-0002-1818-0869]

[1] Saarland Informatics Campus (SIC), Germany, {andreas.schmidt,vogelgesang}@cs.uni-saarland.de
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany, gerhorst@cs.fau.de
[3] Ruhr-Universität Bochum (RUB), Germany, timo.hoenig@rub.de

***Abstract*—*Software Engineering* arose from the need to scale up and decompose the development of large software systems [1]. While the discipline was successful in finding *functional* interfaces (i.e., data formats and callee/caller relationships), the support to formulate *non-functional* interfaces [2] for resource budgets is still lacking. Without resource awareness, designing system software—especially for cyber-physical systems—remains a difficult endeavor [3]. Additionally, resource awareness has become a critical aspect for system software, as we need to design and build CO_2-efficient systems today.**

In this paper, we present ResourceGauge, an approach to ergonomically manage physical resources—such as time or energy—using a stock Rust compiler toolchain. With ResourceGauge, data and/or control-flow units (e.g., functions) become resource-aware and adequate resource use is checked for compositions of multiple software components. By this, expected resource allocation failures are dependably detected at compile time (if static annotations do not comply) as well as runtime (if full compliance cannot be checked statically).

***Index Terms*—Performance Interfaces, Time, Energy**

I. Introduction

Engineering system software is challenging—especially for cyber-physical systems (CPSs) [4]. Due to the complexity of systems interacting with the environment and computing optimal control actions, developing CPSs becomes a highly distributed task. Still, individual software components have, due to their CPS nature, resource limits in the amount of time and energy they can request and use. At integration time, such limits must be composed and checked for compatibility. Ultimately, limits must be enforced statically (i.e., at build-time) and dynamically (i.e., at run-time), for instance, using custom logic or leveraging `_timeout(...)` function variants.

To overcome this, we consider *performance interfaces* [2], the non-functional equivalent to established software interfaces (e.g., call conventions, data structure and type). These new interfaces are models mapping from inputs/configurations to non-functional properties [2], [5]. We focus on constraints, for example, resource limits per function call. Industry-grade programming systems do not directly allow specifying and enforcing these resource limits [3], [4]—despite developers' demand [6]. This is a challenge, as developers understand the application better [7] and resource optimizations are difficult, if not impossible to accomplish without resource awareness [8].

The contribution of this paper is threefold:

- We revisit the state of the art in dependable resource-aware software tools and programming support.
- We propose ResourceGauge[1], an approach enabling developers to make components resource-aware, with low overhead and changes to the functionality.
- We present `resourcegauge-rs`, a ResourceGauge implementation, using an unmodified Rust compiler. This demonstrates how ownership systems, being initially designed for memory management, can be reused for physical resource management.

The rest of the paper is structured as follows: Sec. II describes use cases and requirements for resource-aware software components. Background and related work is described in Sec. III. Sec. IV presents the usage and implementation of `resourcegauge-rs`. Sec. V discusses the approach, Sec. VI gives an outlook, and Sec. VII concludes this paper.

II. Requirements & Use Cases

To allow for resource-aware software components (RASCs), the following requirements must be met:

Ergonomic Working with resource limits feels natural to the developer. This is fulfilled if the solution is declarative/functional ("What are the resource limits?") instead of imperative ("This function has this exact timeout."). We use the term in line with the Rust project [16].

Dependable Resource limit exceedance is always detected and handled. We distinguish a) *soft* when exceedance may only be detected after occurring and b) *hard* when exceedance is always anticipated and handled before happening. Whenever possible, ResourceGauge realizes *hard* but falls back to *soft*, if necessary.

Efficient Resource limit exceedance is handled as early as possible, avoiding unnecessary computations. Further, resource management must be lightweight.

Composable When multiple RASCs are combined, their awareness should not interfere—instead, the integration should become more aware thanks to the individual components being aware.

[1]We use *Gauge* figuratively, i.e., we add fill-level- or rate-indicating gauges to language-level items—to improve resource-awareness.

Source	Flow	Data	Compile	Run	Unmodified Compiler	Focus
Green [9]	✓	–	✓	✓	–	energy-saving by approximation
EnerJ [10]	✓	✓	✓	✓	–	energy-saving by approximation
ET [7]	✓	✓	✓	✓	–	awareness
Liu et al. [11]	✓	✓	–	✓	✓	application-level energy management
Carbonneaux et al. [12]	✓	–	✓	–	–	worst-case resource analysis
Eco [13]	✓	✓	✓	✓	–	energy- and temperature-awareness
Dehesa et al. [14]	✓	✓	✓	✓	✓	constant resource usage for side-channel security
Brown et al. [15]	✓	✓	✓	✓	–	proofs for resource contracts
ResourceGauge (this paper)	✓	✓	✓	✓	✓	enforcing resource limits; designed to be easily expandable

TABLE I
RELATED WORK COMPARED BY *Target* (FLOW VS. DATA), *Time* (COMPILE VS. RUN), USAGE OF A MODIFIED COMPILER, AND *Focus*.

ResourceGauge is motivated by several real-world use cases found in both user applications and operating system (OS) kernels. Using RASCs, developers discover and prevent code patterns that traditionally would have resulted in resource bugs at execution time. Further, RASCs significantly simplify code that (already) employs manual resource management. In kernel-space, for example, interrupts often have implicit resource limits that must be enforced. Doing so manually frequently leads to resource bugs [17]. Here, an automated approach like ResourceGauge greatly helps. Further, in user-space, distributed server-applications deal with concurrent, time-critical requests. ResourceGauge supports developers in reducing tail latency statically and improve timely failover using dynamic methods. To summarize, RASCs are highly relevant for various kernel and user code bases. We guide the design and implementation of ResourceGauge aligned to these use cases.

III. BACKGROUND

We focus on making physical resources (time and energy) first-class elements of the programming environment (i.e., the programming language, the compiler, and libraries).

Time as a Resource

There is a significant body of work on time in (distributed) programming systems [18], [19]. A core issue is that, in today's programs, *timing is emergent rather than specified* [20]. Lee et al. [3] argue that time should be a concept of all programs, not just real-time programs—where worst-case execution time (WCET) analysis is common [21], [22]. They propose *alignment*, *precedence*, *simultaneity*, and *consistency*, as concepts that are not (yet) part of common programming environments. Concepts such as XC [23] and OASIS [24] extend the C programming language to directly handle timing aspects. Our work focuses on the synchronization of logical system execution and the physical passing of time.

Energy as a Resource

As energy is different from time in many ways [8] (e.g., time progression is universal and constant), the state of the art is even further away from establishing thorough energy-awareness. However, there exist promising approaches to account for system/component energy consumption to applications. For example, PowerScope [25] allows consumed energy to be attributed to individual processes and procedures using time-based statistical sampling of the power consumption. Similarly, Power Sandbox [26], a more recent approach to accounting, isolates applications in their vertical hardware and software stack. To accommodate this, RASCs should be orthogonal to the OS-level accounting approach.

Rust Language Ecosystem

Despite being young[2], the Rust language and ecosystem have established a reputation for providing functionality that is ergonomic [16], [27], dependable [28], and composable. Further, the toolchain already leads to resource-*efficient* results [29] in comparison to most other programming languages. `resourcegauge-rs` relies on the following features of the upstream Rust compiler: a) ownership-based information flow, b) generic and zero-sized types, c) compile-time constant evaluation, and d) metaprogramming using macros. If the relevant features are implemented in other languages [30], [31], our approach can be implemented in those as well.

Related Work

In Table I, we compare alternative approaches to provide resource-awareness. First, we indicate whether these approaches deal with *(Information) Flow (or Code)* analysis, i.e., whether the behaviour is taken into account. Second, we indicate if the approach deals with specific *Data (Structures)*, i.e., the data is taken into account. We further highlight whether the approaches work at *Compile*- and/or *Run*-time, whether they use an *Unmodified Compiler*, and their intended *Focus*. While this is a coarse categorization, it is evident that our approach is among those that consider both resource implications of flow and data—looking at both the compile time as well as the runtime. `resourcegauge-rs` is also one of the few that work with an unmodified compiler.

Looking at the compile time approaches, we can distinguish approaches based on WCET analysis [12], [15] and approaches based on making resources explicit in the *type system* [7], [10]. `resourcegauge-rs` takes the latter approach. If we look in detail at the runtime approaches, we see that different strategies are possible: [9] and [10] use approximate computing, trading accuracy of computed results for a lower energy demand. In contrast, [7], [11], and [13] adapt the processor's

[2]Initial project by Graydon Hoare since 2006, version 1.0 in 2015.

speed to turn residual latency into an energy saving. The security-focussed [14] applies *padding*, i.e., it makes functions constant in their resource usage. Conceptually, our approach currently only focuses on enforcing the resource budgets—but could be used to build the approaches listed above.

IV. resourcegauge-rs

resourcegauge-rs provides ergonomic and dependable management of physical resources without using custom compiler extensions to an industry-grade systems programming language (Rust). Ergonomics are achieved by making it straightforward to turn a resource-agnostic piece of software into a resource-aware solution. Hence, the library introduces a) a set of macros to enable resource management, b) supportive algorithms and data structures to deal with resources, c) code transformations, d) static analysis, and e) run-time monitoring. Lst. 1 is an example where resource-agnostic existing code is extended by a few additional language elements—transforming it into a resource-aware function.

A. Macros

At the core of resourcegauge-rs is a procedural attribute macro `#[resourcegauge]` (cf. Lst. 1). This macro can receive multiple attributes, i.e., the `max_latency="2s"` or energy limits. The procedural attribute macro completely processes the function it is associated with, allowing for arbitrary code transformations. A second component of the API are function-like macro calls, such as:

`check!()`: Introduce a runtime check for resources. As resource checks impose overheads, it is up to the developer to decide at which locations within a function these checks should be made. Additionally, implicit checks are performed at function exit.

`remaining_latency!()`: Return the remaining time before a deadline-exceedance will be inevitable. This macro can be used to parameterize a subroutine with an automatically computed deadline. Here, previous measurements of code blocks are considered, together with the overall function resource budget. An equivalent for energy is `remaining_energy!()`.

Apart from these macros, functions that are put under `#[resourcegauge]` are modified as minimally as possible. We refrain from introducing language constructs that are not part of the Rust language.

B. Algorithms & Data Structures

A type `ResourceFailure` is introduced that implements `Error`. The type has variants `Latency` and `Energy`, both of which carry a `FailureMode` enum: a) `BudgetExceeded`, which also reports how much resource was *overspent* or b) `BudgetExceedanceExpected` with the remaining budget at the time of failure.

Furthermore, the two types `ExpiringAt<T>` and `ExpiringIn<T, const D: Duration>` are introduced that allow data to be made resource-aware (as opposed to full functions as before). One approach is to create an `ExpiringAt` with a deadline, after which reading it causes a `BudgetExceeded` failure. Ideally, this type would act as a smart pointer (implementing `Deref`), allowing it to be used like normal, but with added resource checks. However, if we obtain a reference (`&T`), a developer can pass it around without requiring another deadline check. Instead, the current design provides several methods to get access to `&T` in a resource-aware manner (see Lst. 2 for the full signature):

`take` is used to unpack `ExpiringAt<T>`, after which the developer is responsible for the "last mile" between taking and actually using the data.

`map` accepts a closure accessing `&T`, whose result is wrapped into another `ExpiringAt<T>` instance with the original deadline. As in `Result::map`, the closure is only executed if the data is still valid—otherwise we short-circuit to `ResourceFailure`.

`with` accepts a closure and chains `map` and `take` together. If the result is `Ok`, the closure was able to completely process the expiring data while it was valid.

Additionally, we implement common traits (e.g., `Copy`, `Clone`, `Send`, `std::ops::Add`) for `ExpiringAt<T>` if `T` itself supports the respective trait. The result of the trait functions is again an `ExpiringAt<T>`, and its validity is the intersection of the operands. Thereby, computations of invalid data can be made—the system remains safe, as the result must eventually go through the fallible `take()`. We do not implement these traits for `ExpiringIn<T,D>` as it contains the static budget `D`. Instead, a developer must convert `ExpiringIn<T,D>` into `ExpiringAt<T>`, explicitly dropping the type-level latency budget.

resourcegauge-rs intends to provide the following algorithms: 1) Measurements with statistical filtering, 2) Budget calculations accounting for both static and dynamic data, and 3) Budget checks based on static data only, allowing resourcegauge-rs to reject programs with incompatible budget requirements (cf. Sec. IV-E).

C. Code Transformation

The expansion of the `#[resourcegauge]` macro performs two modifications to implement resource management (cf. Lst. 3). First, we turn the function signature's return value from `T` to a `Result` with `ResourceFailure` as well as the remaining `ResourceLatency`. Then, we modify all return expressions to wrap the original result properly. Finally, runtime monitoring code is added (cf. Sec. IV-D).

D. Runtime Monitoring

As discussed before, the nature of physical resources and data-dependent operation semantics implies that we cannot enforce all resource limits at compile time. Instead, we semi-automatically add resource checking code. Semi, because we do so automatically at function exit, but allow the developer to introduce more checks—improving the performance, as checks are more fine-granular and functions exit earlier.

For each monitored function, we introduce a global, thread-safe storage for measurements. At the function start, we

```rust
 1
 2
 3
 4 fn receive () -> T {
 5
 6   let s = socket();
 7
 8
 9
10   let d = s.recv();
11
12
13
14
15   let r = d[0..256].to_upper();
16   T(r)
17 }
```

a) Agnostic Code

```rust
 1 use resourcegauge::prelude::*;     // imports check!(),
 2                                    // ResourceFailure, ...
 3 #[resourcegauge(max_latency="10sec")]
 4 fn receive () -> T {
 5                        // ^ becomes Result<T, ResourceFailure>
 6   let s = socket();   // amenable to measurement (dynamic)
 7   check!();           // detects ResourceFailure,
 8                       // e.g. BudgetExceeded
 9
10   let d = s.recv_timeout(remaining_latency!());
11                            // ^ derived timeout
12
13   check!();  // may detect e.g. BudgetExceedanceExpected
14
15   let r = d[0..256].to_upper(); // WCET amenable (static)
16   T(r)
17 }                                 // implicit check!()
```

b) Small scale changes (lines 1, 3, 7, 10, 13) to make it aware code

Listing 1: Semi-automatic resource management using `resourcegauge-rs`.

```rust
impl<T> ExpiringAt<T> {

  fn take(self) -> Result<T, ResourceFailure> { ... }

  fn map<U>(&self, closure: impl FnOnce(&T) -> U) -> ExpiringAt<U> { ... }

  fn with<U>(&self, closure: impl FnOnce(&T) -> U) -> Result<U, ResourceFailure> { ... }
}
```

Listing 2: Synopsis of methods on `ExpiringAt` data types.

```rust
struct ResourceLatency<const L:
    std::time::Duration> {}

// Before:
#[resourcegauge(max_latency="2s")]
fn f(in: In) -> Out {
    // ...
}

// After:
const LATENCY_f: Duration =
    Duration::from_secs(2);
fn f<const L: Duration>(latency:
    ResourceLatency<L>, in: In)
  -> Result<(Out, ResourceLatency<
    {L.checked_sub(LATENCY_f).unwrap()}>
  ), ResourceFailure> {
    // ...
    Ok((out, latency.shorten::<{
      L.checked_sub(LATENCY_f).unwrap()}>()
    ))
}
```

Listing 3: Static analysis using zero-sized types (`ResourceLatency`) and const generics (`L`).

check the leftover budget and set limits accordingly. At intermediate check locations, we look backwards (resource limit exceeded?) and forwards (will the resource limit be exceeded with high confidence?), failing the function if appropriate. At `remaining_` locations, we compute the remaining budget for the end of the current block (based on measurements of future blocks and the overall budget). At the function end, we check again if a resource limit was exceeded.

`resourcegauge-rs` allows for platform-specific measurement methods. For latency measurements, every system comes with the `std::time::Instant` API and more accurate measurement on systems with cycle counters (e.g., on x86 the `rdtsc` instruction and time interpolation can be used [32]) are possible. Energy consumption can be measured or estimated using processor performance counters, external measurement devices, or energy-models. For Intel x86 systems, *Running Average Power Limit* (RAPL) [33] is readily available on recent processors. To efficiently use external measurement devices, interrupts based on energy budgets can be used to efficiently limit energy consumption [34]. If implemented at the system-level, code annotated with `resourcegauge-rs` can be interrupted by an energy-out signal once the OS has detected a budget violation using the hardware interrupt. Finally, energy models based on other performance counters can be used as a fall-back if no direct energy measurements are supported by the platform.

E. Static Analysis

In our static analysis, if a program is accepted (compiled), the `resourcegauge-rs` limits are not in conflict, although run-time failures are still possible. However, these failures are now *function-local* and not due to unsatisfiable composition. While elaborate static analysis is possible with dedicated tools, `resourcegauge-rs` only uses existing Rust features (some are nightly, cf. Sec. V) and hooks into the common compile chain. At the core of the analysis are *zero-sized* [35], *const generic* [36] types, e.g., `ResourceLatency` in Lst. 3. Being zero-sized, the type is dropped after compilation—enabling zero runtime cost. As the type is const generic, we can use actual constant values (e.g., a duration in this case) to tell different types apart. As Rust supports constant evaluation, these values can also be leveraged to compute values at compile time—again incurring no runtime cost.

In Lst. 3, we see that an extension of the macro introduces additional generic parameters. The function becomes const generic with respect to a concrete latency value. `latency` becomes a zero-sized call parameter with *move semantics* (i.e., the function takes ownership of the latency). At the end of the function, the latency is shortened, i.e., the returned type constant is the input constant minus the annotated latency.

At the call site, a caller must now construct a budget `ResourceLatency<Duration::from_secs(5)>` and pass it to `f`. Down the call stack, if a function is called without a sufficient budget, the compilation fails. While calling a function with a usage of X and a budget of Y, the result is a budget of (Y-X). If this becomes negative the compilation fails as well because `checked_sub` panics.

V. Discussion

Overheads

`resourcegauge-rs` introduces compile- and runtime overheads which we will evaluate thoroughly in future work. At runtime, there is additional monitoring for both latency and energy. However, the end user controls this overhead as they can pick how often the resource is checked—at the same time picking how early resource violations are detected. At compile time, there are two classes of overheads. First, due to Rust's monomorphization of generic code, every combination of latencies creates code duplicates, thereby increasing the binary size. Second, the compilation time increases due to a) procedural macro execution and b) constant evaluation. To summarize, `resourcegauge-rs` gives the user control over the introduced overheads whenever possible. This allows them to balance convenience and efficiency.

Legacy Code / Graceful Improvement

As we are using a nightly Rust compiler version (we do not use a fork or 2.0 version of Rust) our solution is in tight sync with the main Rust project and ecosystem. Thanks to the backwards compatibility guarantees of Rust, this allows `resourcegauge-rs` to be used in existing code bases with minimal friction. `resourcegauge-rs`is a library that operates on functions and data types, thus a developer can gradually introduce resource-awareness—without requiring major rewrites. Hence, existing resource-agnostic code can be upgraded step-by-step to become resource-aware.

Error Messages

The Rust ecosystem has a reputation for providing well-designed and helpful error messages to developers. However, macros as well as const evaluation and const generic types can lead to situations where code annotated with `#[resourcegauge()]` causes confusing error messages. We currently investigate ways to improve this situation and consider contacting the Rust team with respect to custom error messages for violations of const generic constraints. One way is the (also not yet stable) `Diagnostic` API [37].

Nightly Compiler Features

`resourcegauge-rs` requires a *nightly* compiler version[3]. The reason for this is the use of constant evaluation, of which only a minimal subset is stabilized so far and some functionality is still missing. The relevant features flags are: `adt_const_params` [38], `const_option` [39], `generic_const_exprs` [40], and `inline_const` [41]. This presents the danger that `resourcegauge-rs` breaks in the future. Such a break would be technical, but most likely not conceptual as the Rust project is committed to stabilising these features.

Execution-Time Computations

The `const Budget::<B1>::shorten::<B2>()` method has the purpose of statically turning a larger budget B1 into a smaller budget B2, if B1 is larger. As these are constant generic parameters on zero-sized types, this ideally happens at compile time. However, with the current compiler this cannot be achieved, hence we check at execution-time if B1 is smaller B2. As this happens in each function invocation, a violation would be caught through any test that executes the function at least once, i.e., it is not a corner-case that could go undetected.

Availability

`resourcegauge-rs` is MIT licensed and available at https://doi.org/10.5281/zenodo.8026935.

VI. Future Work

`resourcegauge-rs` is a first working prototype for resource-aware Rust components, based on which we can explore several research directions.

The annotated maximum latencies are currently found in an iterative process, i.e., the developer adding a constraint (probably derived from system design) and then testing the system. The sublatencies of code blocks are instead gathered by measurement, even though some blocks are purely computational and amenable to WCET analysis. While we are targeting systems for which a strict WCET analysis is not possible, future work can leverage hybrid WCET approaches (e.g., [21], [22]) to validate the annotations/sublatencies or automatically infer them.

Conceptually, lifetimes in Rust and the resource budgets on data structures are closely related. Hence, leveraging lifetimes, the ownership system, and a borrow checker such as

[3]To be precise, we used rustc 1.67.0-nightly (edf018221 2022-11-02)

`polonius` [42] could make the detection of more static resource violations possible. Leveraging a recent prototype of liquid types in Rust, called Flux [27], provides further potential for static resource analysis.

So far, we have established dependable resource limit management (i.e., compiler fails with static violations, runtime yields `Result` that must be used). A different line of investigation is dynamic optimization, i.e., exploiting excess resources at runtime. The natural approach would be to trade time for energy, by slowing down the system—but more advanced schemes can be envisioned.

Currently, we deal with comparably continuous treatment of energy (limited by system precision). A direction worth investigating is the phase and mode approach to energy from *Energy Types* [7]. Potentially, it is possible to use `resourcegauge-rs`'s annotations to infer appropriate phases and modes.

As discussed in Sec. III, we focus on sequential programs and their resource usage. Modern cyber-physical systems are concurrent, distributed systems, where a more global view is required [3], [20]. Future work will hence investigate how the annotations can be extended from functions to tasks (i.e., processes), for example, with periodic behaviour.

Finally, `resourcegauge-rs` is not just targeting new-to-be-developed software, but can also be used to refactor Rust programs with custom resource-awareness. In the systems community, Coccinelle [43] is a well-established tool to search for and transform systems code (e.g., in the Linux kernel). By leveraging Coccinelle to search for custom resource-aware code and replacing it, `resourcegauge-rs` will be used to discover and fix resource bugs early.

VII. Conclusion

We presented `resourcegauge-rs`, an implementation of ergonomic and dependable resource-aware components in Rust. `resourcegauge-rs` enables developers to declaratively annotate resource usages such as time and energy. This allows a) the application to dependably react to execution–time violations and b) checking compositions of annotated functions and/or expiring data types for their compatibility at compile time. Using an upstream version of the Rust compiler, we show that this functionality can be achieved in Rust itself, without designing a custom language or compiler extensions—allowing for widespread use in the future.

Acknowledgment

This work was funded by the German Research Foundation (DFG) grant 389792660 as part of TRR 248 – CPEC (see https://perspicuous-computing.science), within the Priority Program 2377 (Disruptive Memory Technologies, project numbers 502228341 "Memento" and 465958100 "NEON"), and within the Priority Program 2378 (Resilient Worlds, project number 502615015 "ResPECT"), and by the Bundesministerium für Bildung and Forschung (BMBF, Federal Ministry of Education and Research) for the project AI-NET-ANTILLAS 16KIS1315. Luis Gerhorst's contribution to this work was made prior to his employment at Amazon.

References

[1] Peter Naur and Brian Randell. Software Engineering: Report of a Conference sponsored by the NATO Science Committee, Garmisch, Germany. 1969.

[2] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *USENIX NSDI*, pages 567–584, 2022.

[3] Edward A Lee and Marten Lohstroh. Time for all programs, not just real-time programs. In *International Symposium on Leveraging Applications of Formal Methods*, pages 213–232. Springer, 2021.

[4] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *IEEE ISORC*, 2008.

[5] Norbert Siegmund, Johannes Dorn, Max Weber, Christian Kaltenecker, and Sven Apel. Green configuration: Can artificial intelligence help reduce energy consumption of configurable software systems? *Computer*, 55(3):74–81, 2022.

[6] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proc. of the 11th Working Conference on Mining Software Repositories*, pages 22–31, 2014.

[7] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *Proc. of the ACM international conference on Object oriented programming systems languages and applications*, pages 831–850, 2012.

[8] Vaastav Anand, Zhiqiang Xie, Matheus Stolet, Roberta De Viti, Thomas Davidson, Reyhaneh Karimipour, Safya Alzayat, and Jonathan Mace. The Odd One Out: Energy is not like Other Metrics. In *HotCarbon 2022: 1st Workshop on Sustainable Computer Systems Design and Implementation*, 2022.

[9] Woongki Baek and Trishul M Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proc. of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 198–209, 2010.

[10] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. *ACM SIGPLAN Notices*, 46(6):164–174, 2011.

[11] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-oriented characterization of application-level energy optimization. In *Intl. Conf. on Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2015.

[12] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 467–478, 2015.

[13] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *37th International Conference on Software Engineering*, volume 1, pages 767–777. IEEE, 2015.

[14] Mario Dehesa-Azuara, Matthew Fredrikson, Jan Hoffmann, et al. Verifying and synthesizing constant-resource implementations with types. In *Symposium on Security and Privacy (SP)*, pages 710–728. IEEE, 2017.

[15] Christopher Brown, Adam D Barwell, Yoann Marquer, Céline Minh, and Olivier Zendra. Type-driven verification of non-functional properties. In *Proc. of the 21st International Symposium on Principles and Practice of Declarative Programming*, pages 1–15, 2019.

[16] Aaron Turon. Rust's language ergonomics initiative. https://blog.rust-lang.org/2017/03/02/lang-ergonomics.html, 2017. Accessed: Jan. 20, 2023.

[17] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. An incremental path towards a safer os kernel. In *Proc. of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 183–190, 2021.

[18] Bo Wan, Haizhao Luo, Kaiqi Zhou, Xi Li, Chao Wang, Xianglan Chen, and Xuehai Zhou. A time-aware programming framework for constructing predictable real-time systems. In *Intl. Conf. on High Performance Computing and Communications; Intl. Conf. on Smart City; Intl. Conf. on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 578–585. IEEE, 2017.

[19] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Rohan Tabish, Hokeun Kim, Shaokai Lin, Christian Menard, Cong Liu, and Edward A Lee. Xronos: Predictable coordination for safety-critical distributed embedded systems. *arXiv preprint arXiv:2207.09555*, 2022.

[20] Marten Lohstroh, Martin Schoeberl, Andres Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A Lee. Invited: Actors revisited for time-critical systems. In *Design Automation Conference (DAC)*, 2019.

[21] Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. Timeweaver: A tool for hybrid worst-case execution time analysis. In *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[22] Abderaouf N Amalou, Isabelle Puaut, and Gilles Muller. We-hml: hybrid wcet estimation using machine learning for architectures with caches. In *27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 31–40. IEEE, 2021.

[23] Douglas Watt. *Programming XC on XMOS devices*. XMOS Limited, 2009.

[24] Stéphane Louise, Matthieu Lemerre, Christophe Aussagues, and Vincent David. The oasis kernel: A framework for high dependability real-time systems. In *13th International Symposium on High-Assurance Systems Engineering*, pages 95–103. IEEE, 2011.

[25] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Proc. IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 2–10, 1999.

[26] Liwei Guo, Tiantu Xu, Mengwei Xu, Xuanzhe Liu, and Felix Xiaozhu Lin. Power sandbox: Power awareness redefined. In *Proc. of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Nico Lehmann, Adam Geller, Gilles Barthe, Niki Vazou, and Ranjit Jhala. Flux: Liquid Types for Rust. *preprint arXiv:2207.04034*, 2022.

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe Systems Programming in Rust. *Communications of the ACM*, 64(4):144–152, 2021.

[29] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[30] Microsoft. Project Verona. https://github.com/microsoft/verona/tree/e4341833624bb39e8e3a290b204c348fe85da5c4, 2022. Accessed: Jan. 18, 2023.

[31] The kernel development community. Ownership-like reference tracking for bpf_ringbuf_reserve/submit() - Debian Bullseye bpf-helpers(7) Manpage. https://manpages.debian.org/bullseye/manpages/bpf-helpers.7.en.html#void~5, 2022. Accessed: Jan. 18, 2023.

[32] Stefan Reif, Andreas Schmidt, Timo Hönig, Thorsten Herfet, and Wolfgang Schröder-Preikschat. X-lap: A systems approach for cross-layer profiling and latency analysis for cyber-physical networks. In *Proc. of the 15th International Workshop on Real-Time Networks (ECRTS RTN)*, RTN, Dubrovnik, Croatia, June 2017.

[33] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[34] Luis Gerhorst, Stefan Reif, Benedict Herzog, and Timo Hönig. Energy-budgets: Integrating physical energy measurement devices into systems software. In *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8, 2020.

[35] Rust community. Exotically Sized Types - The Rustonomicon. https://doc.rust-lang.org/nomicon/exotic-sizes.html#zero-sized-types-zsts, 2022. Accessed: Jan. 2, 2023.

[36] The Rust const generics project group. Const generics MVP hits beta! https://blog.rust-lang.org/2021/02/26/const-generics-mvp-beta.html, 2021. Accessed: Jan. 2, 2023.

[37] Rust community. Diagnostic in proc_macro - Rust. https://doc.rust-lang.org/stable/proc_macro/struct.Diagnostic.html, 2022. Accessed: Jan. 2, 2023.

[38] Rust community. Tracking Issue for more complex const parameter types: feature(adt_const_params) - Issue #95174 - rust-lang/rust. https://github.com/rust-lang/rust/issues/95174, 2022. Accessed: Jan. 2, 2023.

[39] Rust community. Tracking issue for const Option functions - Issue #67441 - rust-lang/rust. https://github.com/rust-lang/rust/issues/67441, 2019. Accessed: Jan. 2, 2023.

[40] Rust community. Tracking Issue for complex generic constants: feature(generic_const_exprs) - #76560 - rust-lang/rust. https://github.com/rust-lang/rust/issues/76560, 2020. Accessed: Jan. 2, 2023.

[41] Rust community. Tracking Issue for inline const expressions and patterns (RFC 2920) - #76001 - rust-lang/rust. https://github.com/rust-lang/rust/issues/76001, 2020. Accessed: Jan. 2, 2023.

[42] Rémy Rakic, Niko Matsakis, and contributors. rust-lang/polonius: Defines the Rust borrow checker. https://github.com/rust-lang/polonius/tree/0a754a9e1916c0e7d9ba23668ea33249c7a7b59e, 2022. Accessed: Jan. 2, 2023.

[43] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *Annual Technical Conference (USENIX ATC 18)*, pages 601–614, 2018.

Arm MUCH: Full-spectrum hardware-event-based Armv8 application profiler

Andrea Misuraca
Technical University of Munich
andrea.misuraca@tum.de

Andrea Bastoni
Technical University of Munich
andrea.bastoni@tum.de

***Abstract*—Profiling on Arm architecture using the Performance Monitoring Unit (PMU) gives developers low-overhead access to Hardware Events Monitors (HEM). These events are available on every Armv8-based platform and provide detailed information on the execution of applications, including multicore-related interference and usage of shared hardware resources. Prompt access to such information is fundamental for mixed-criticality systems in order to manage and regulate interference. Despite board configuration providing dozens of different HEMs (typically 30 to 50), the PMU only allows the simultaneous monitoring of a limited number of them (generally between 6 and 8). A simultaneous full-spectrum hardware profiling based on HEMs is therefore not possible. Recently, a methodology (MUltiCorrelation HEM reading and merging approach *MUCH*) has been proposed to statistically reconstruct a coherent HEM-based execution context from multiple application runs. MUCH has been validated only in a bare-metal environment for an NXP T2080 PowerPC platform. Given the widespread adoption of the Arm architecture for embedded systems, in this paper we implemented the MUCH approach for the Armv8 architecture on Linux to assess the viability of such an approach for complex systems. Our results on a Raspberry Pi 3 Model B confirm the practicality of the approach on complex Armv8-based systems. Furthermore, we explored how to leverage the obtained full statistical profile to derive properties of the analyzed application, for example, the sufficient set of HEMs to simultaneously monitor at runtime with minimal information loss, and machine learning-based models for HEMs prediction on a future set of applications.**

***Index Terms*—Arm, profiling, PMU, HEM, pairwise-correlation**

I. Introduction

Software profiling is a dynamic program analysis technique where a program's behavior is modeled using data monitored at runtime. Hardware-based software profiling enables developers to better understand the execution of a program, monitoring how the hardware layer reacts to the application. When integrating applications with different criticalities on complex multi-processor systems, monitoring of hardware performance counters is the basic mechanism used by several techniques (*e.g.*, [14], [23]) to manage hardware-related interference (especially in the memory subsystems) and to achieve higher isolation among otherwise independent applications.

The Arm architecture exposes Performance Monitor Units (PMUs) as architecture-specific, on-chip solution for low-overhead hardware event-based profiling. In this context, Hardware Event Monitors (HEMs) constitute the hardware events to be observed, and Performance Monitor Counters (PMCs) represent the actual architectural counter where the monitored information is stored. The Armv8 architecture defines a standard, common set of HEMs that should be present in any implementation, and a set of HEMs that could be optionally implemented by manufacturers [6].

Compared to other software profiling and debugging solutions (*e.g.*, Arm CoreSight [8]), Arm hardware-event-based profiling has low overhead and generates less execution interference in terms of *e.g.*, resource usage and bus accesses. Unfortunately, extensive application profiling and monitoring using HEMs is limited by the low number of PMCs that are typically available on Arm boards. In fact, PMCs are often a magnitude order fewer than the HEMs that could possibly be monitored. Therefore, despite the standard availability of PMCs and HEMs on Arm-based platforms, monitoring the *full hardware execution context* observing one single run of an application (or a benchmark) is not possible.

MUltiCorrelation HEM reading and merging (MUCH) is a recent approach [20] that relies on statistical analysis to reconstruct the full-hardware application context across multiple runs of an application or benchmark. The approach has been developed to target explicitly complex multiprocessor systems-on-chip (MPSoC), where HEM monitoring and profiling is becoming progressively more important to master interference among mixed-criticality (real-time) applications [15]. In [20], the approach has been validated in a bare-metal setup (without operating system) on a PowerPC NXP T2080. Given the increasing relevance of the Arm architecture for complex MPSoC used in safety critical automotive, industrial, and avionics domains, this paper proposes Arm MUCH, an application profiler for the Armv8 architecture that adopts the MUCH-approach and runs on a complex operating systems such as Linux. With Arm MUCH we:

- Validate whether the MUCH approach can be applied to real-world Arm architectures together with a complex operating system such as Linux. Our results on a Raspberry PI 3 with Linux 5.9.93 confirm the applicability of the MUCH methodology to the Arm architecture even with a complex operating system.
- Contribute a framework for implementing MUCH on Armv8. In particular, we automated the HEM allocation

Andrea Bastoni was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

at application runtime, the gathering of the retrieved data and performing the statistical MUCH methodology.

- Interface the Arm MUCH framework with multiple profiling technologies for Arm Linux, namely *perf*, *eBPF*, and inline assembly for manual HEMs allocation.
- Explore how to derive the minimal set of HEMs that best characterize an application. This enables confidence in capturing the key properties of an application despite the limited number of monitored PMCs.
- Implement AI-based HEM-prediction systems that use the statistical data retrieved by MUCH to reconstruct the *complete set of HEMs inside one benchmark execution* by forecasting all non-monitored HEMs.

The rest of this paper is organized as follows. Section II introduces the concepts of MUCH and discusses previous work. Section III presents the architecture of Arm-MUCH, while Section IV discusses its implementation and our experimental results. Section V concludes.

II. Background and Related Work

A. The MUCH Approach

The core concept underlying the MUCH approach [20] is to *rearrange and merge individual and independent HEM readings* into one single coherent dataset as if each all HEMs were all measured in the same run. MUCH employs Multi-Variate Gaussian Distributions *(MVGD)* to preserve all pairwise HEM correlations simultaneously. The goal is to use measured data to generate full-spectrum HEM vectors (merged from multiple runs) of size nh in accordance with the MVGD model $X \sim N_{nh}(\hat{\mu}, \hat{\Sigma})$, where $\hat{\mu}$ and $\hat{\Sigma}$ are the empirical expected value and empirical covariance matrix obtained in multiple runs of the experiments for the same HEM h^i. The empirical covariance matrix $\hat{\Sigma}$ can be obtained from the empirical *correlation* matrix $\hat{S}$ that expresses the correlation $\hat{\rho_{ij}}$ for each pair of empirical HEMs vectors $(\{h^i\}, \{h^j\})$. Each empirical covariance value $\hat{\sigma_{ij}}$ in $\hat{\Sigma}$ is computed as: $\hat{\sigma_{ij}} = \hat{\rho_{ij}} \cdot \hat{\sigma_i} \cdot \hat{\sigma_j}$, where $\hat{\sigma_i}$ is the variance associated with an empirical vector of HEMs $\{h_i\}$.

In order to use Multi-Variate Gaussian Distributions, the correlation between two different HEMs must be correctly evaluated. Each HEM needs to be measured in the same sub-experiment, namely a *benchmark run*, at least once with every other HEM present in the machine. Each HEM's "true" value is statistically modeled with a *Gaussian distribution*. Therefore, by the *central limit theorem*, each sub-experiments should run multiple times for a sufficient amount of time. We run each sub-experiment for at least 10 seconds and repeat every experiment 50 times.

From a mathematical point of view, the challenge is to reorder the grouped sample vector $\{h_i\}$ for each HEM h_i, such that for each pair of HEMs $\{h_i\}$ and $\{h_j\}$, the empirical correlation $\hat{\rho_{ij}}$ of the grouped samples is close to ρ_{ij}, namely the Pearson's empirical correlation between $\{h_i\}$ and $\{h_j\}$, calculated in the sub experiment in which they are both allocated.

B. Previous Work

Despite the risks of non-precise and context-dependent event accounting, the low overhead and widespread availability of HEMs have been exploited in a host of tools to gain insights into application's behavior [10]. In the embedded real-time field, the usage of HEMs have been leveraged in multiple works to monitor and regulate MPSoC interference at both cache [13], [14], interconnect [19], [23], and DDR level [17], [22]. Recent architecture-level features such as Arm MPAM [7] or Intel RDT [12] extend the concept of HEMs to provide quality of service throughout the memory subsystem. The usage of HEMs for profiling purposes in the GPU and accelerators domain is a standard practice supported *e.g.*, by tools such as the NVIDIA Nsight Systems [16]. The Arm v8.2 architecture also introduced a dedicated infrastructure for statistical profiling [9], but its usage and adoption is still limited. The Arm Coresight [8] infrastructure can deliver both hardware assisted application profiling and debugging capabilities. Its overheads for data collection and exporting hinder nonetheless its usage for low-overhead hardware profiling.

Alternative approaches to MUCH (*e.g.*, [11], [18], [21]) to merge HEMs have been shown [20] to be not applicable to HEMs or inferior to the MUCH approach.

III. Architecture

Our Arm MUCH framework is divided in two main components:

- **Profiler Middleware** that will access and manage the run-time progress of the profiled application. Currently, Arm MUCH supports both *perf subsystem* and *kprobes* in order to evaluate and obtain HEMs data at run-time from the profiled application. We note that any profiler compatible with the perf output format could be used for the data acquisition.
- **Data Analysis Framework** that performs the data processing. This includes data collection, loading and writing benchmark sessions to disk, and the statistical evaluations associated with MUCH.

The framework has been split into two components to provide a wider set of profiling tools that the end-user can leverage to validate the data collected. The implementation abstracts away details of the *e.g.*, Linux *perf* API and provides a unified interface for data processing that facilitated the validation of the experimental data.

A. Profiling Middleware

The profiling layer supports two ways to allocate at run time the selected set of HEMs to be monitored.

The first approach uses Linux *perf* to select the HEMs from bash's command line. This approach is the easiest one, as we do simply need to spawn a correct instance of *perf* with the application we choose to monitor and the right sets of *HEMs* right after the *-e argument*. Despite its easiness of use, with this approach we can only monitor the entire application from start to end. Specifically, we cannot monitor particular sections of the application or have warm-up phases *e.g.*, to avoid measuring memory allocation and initialization.

The second supported method for selecting HEMs is a lightweight API for code instrumentation. The API can directly allocate, enable, and disable *hardware events*. The API wraps the `perf_event_open()` syscall to access PMUs for writing and reading purposes. This method provides better integration and function-level granularity with the trade-off of rebuilding the application against the profiling middleware, inserting the right calls for PMU activation, and reading the data through a file descriptor.

B. Data Analysis Framework

The data analysis framework processes the data generated by the profiler—any profiler could be used as long as the traces are in perf-format—and supports the statistical MUCH analysis. The framework consists in a main application that supports different modes, with regard to what the inputs and outputs should be, including *e.g.*, processing previous benchmark iterations using the *-l* flag, or exporting benchmark sessions to disk using *-w* argument. The statistical evaluation is performed in *python* using *numpy* [1], *scipy* [3], and *sklearn* [4]. Plots and graphical evaluation are visualized using *matplotlib*. HEMs vector results are exported as binary file or printed on *stdout*.

The application implements different phases to performs the MUCH analysis of event counters.

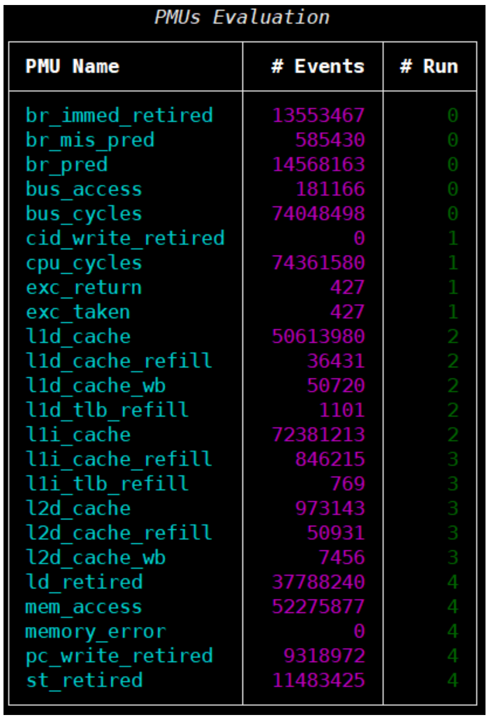
1) **Empirical correlation matrix:** $\hat{S}(i,j)$ is calculated between all pair of HEMs h^i and h^j. Each cell of the matrix is defined as *Pearson correlation coefficient* between HEMs at a given column and row. Hence, this matrix will be symmetrical with unitary values on the main diagonal.

$$\hat{S}(i,j) = \begin{Bmatrix} 1, & \text{for } i = j \\ \hat{\rho}_{ij}, & \text{for } i \neq j \end{Bmatrix}$$

2) **MVGD mapping:** Using *copula theory* and *Percent Point Function*, each *HEM counter measurement* $\{h^i\}$ is mapped to its relative value as it was part of normally distributed linear space.
3) **MVGD-mapped covariance matrix:** $\hat{\Sigma}_0(i,j)$ is calculated between all pair of HEMs, using *MVGD values* with covariance and correlation values from the experiments. Each cell of the matrix is defined as follows:

$$\hat{\Sigma}_0(i,j) = \begin{Bmatrix} \hat{\sigma}_i^2, & \text{for } i = j \\ \hat{\sigma}_i \times \hat{\sigma}_j \times \hat{\rho}_{ij}, & \text{for } i \neq j \end{Bmatrix}$$

4) **Random samples extraction:** This step reconstructs empirical data order statistics from the *MVGD-mapped covariance matrix*, using the multivariate-normal function.
5) **MVGD-based correlation matrix:** This step calculates the correlation matrix between all between all pairs of HEMs using the sorting order defined by the previous steps.



```
                 PMUs Evaluation
┌──────────────────────┬────────────┬───────┐
│ PMU Name             │   # Events │ # Run │
├──────────────────────┼────────────┼───────┤
│ br_immed_retired     │   13553467 │     0 │
│ br_mis_pred          │     585430 │     0 │
│ br_pred              │   14568163 │     0 │
│ bus_access           │     181166 │     0 │
│ bus_cycles           │   74048498 │     0 │
│ cid_write_retired    │          0 │     1 │
│ cpu_cycles           │   74361580 │     1 │
│ exc_return           │        427 │     1 │
│ exc_taken            │        427 │     1 │
│ l1d_cache            │   50613980 │     2 │
│ l1d_cache_refill     │      36431 │     2 │
│ l1d_cache_wb         │      50720 │     2 │
│ l1d_tlb_refill       │       1101 │     2 │
│ l1i_cache            │   72381213 │     2 │
│ l1i_cache_refill     │     846215 │     3 │
│ l1i_tlb_refill       │        769 │     3 │
│ l2d_cache            │     973143 │     3 │
│ l2d_cache_refill     │      50931 │     3 │
│ l2d_cache_wb         │       7456 │     3 │
│ ld_retired           │   37788240 │     4 │
│ mem_access           │   52275877 │     4 │
│ memory_error         │          0 │     4 │
│ pc_write_retired     │    9318972 │     4 │
│ st_retired           │   11483425 │     4 │
└──────────────────────┴────────────┴───────┘
```

Fig. 1: Output example of the PMU statistical evaluation.

6) **Mean Squared Error (MSE):** Since multiple sorting orders could exist, this step selects the best reordering—among multiple iterations—that minimizes the *mean squared error* between the *empirical correlation matrix* and the *MVGD-based correlation matrix*.

In addition to the standard MUCH analysis, the framework can automatically: i) *cluster* HEMs to identify the smallest set of HEMs that could reflect the key characteristics of the profiler application, and ii) make prediction on values of HEMs that were not allocated in a specific run.

Since the maximum number of PMUs that can be traced simultaneously in one run is limited (*e.g.*, the Arm Cortex-A53 supports simultaneous tracing from only six PMUs), clustering the most "meaningful" HEMs helps reducing the number of runs to identify application behaviors. Clustering of n HEMs is determined by maximizing the sum of the *maximum absolute correlation values* with regards to every other HEMs within one experiment. Correlation values are filtered using the *python Pandas Dataframe* [2].

Furthermore, the framework implements three machine learning algorithms (Linear Regression, Multi-layer Perceptron, and Random Forest) to predict values for HEMs that were not allocated in a specific run. The models are trained on the full set of all HEMs.

IV. Implementation and Evaluation

The framework, comprising the profiling middleware and the data analysis component, has been developed in Python and C. The profiler middleware automates the activities of profiling applications and benchmarks using both *perf* and our

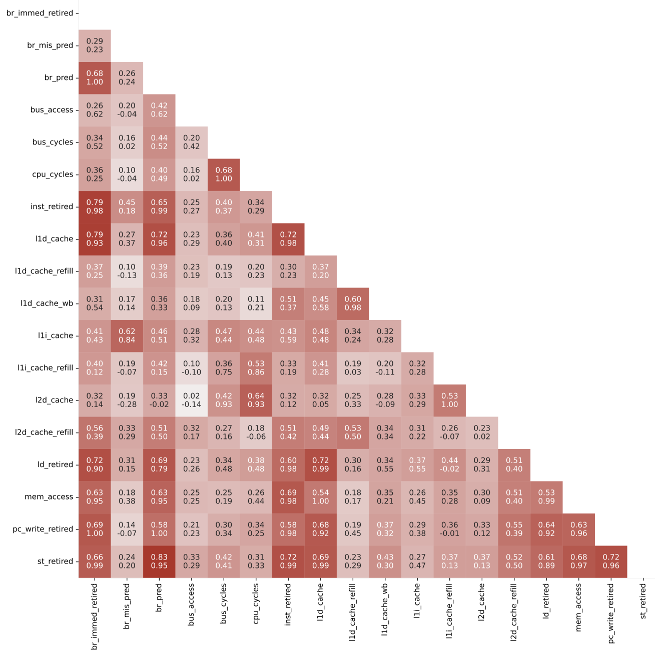


Fig. 2: Single-core experiment correlation matrix for pairs of HEMs. In each box, the upper value is the empirical correlation, the lower value the MVGD-correlation.

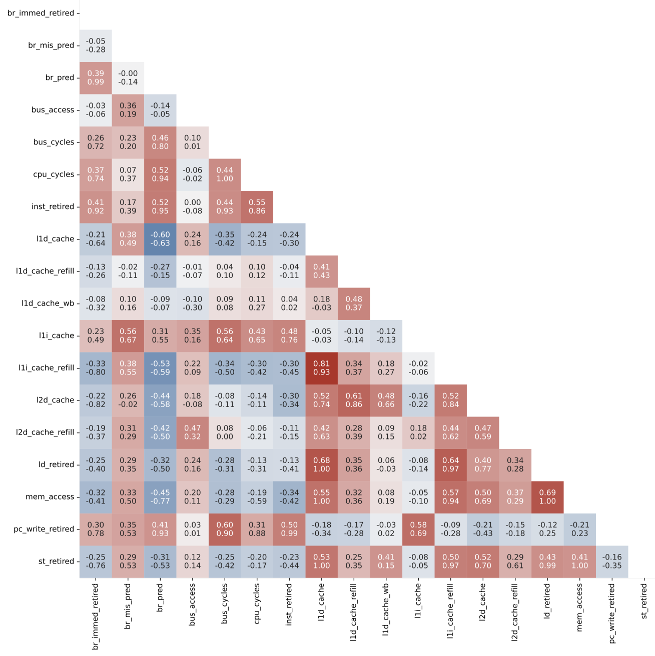


Fig. 3: Multi-core experiment correlation matrix for pairs of HEMs. In each box, the upper value is the empirical correlation, the lower value the MVGD-correlation.

lightweight API for code instrumentation. The data analysis frameworks implements the phases described in Sec. III-B.

In order to enable profiling support in the Linux kernel, the kernel configurations related to `BPF` and `IKHEADERS` must be enabled. The framework requires a version of the *perf* profiling tool that matches the compiled kernel version (*perf* can be found in the Linux kernel sources, under `./tools/perf`). Furthermore, the framework needs Python $>$ 3.9.2 (including pip).

The framework has been validated with Linux on a Raspberry Pi 3 Model B (Arm Cortex-A53 cores).[1] We used *Armbian Buster* as Linux distribution, running a 5.9.93 Linux kernel with enabled profiling-configuration.

A. HEM Correlation Matrices

We evaluated pairwise correlation of HEMs as well as the Arm MUCH approach in a single- and multi-core setup.

The single-core setup also serves as validation and uses a CPU-intensive benchmark that computes the *discrete logarithm* for random natural numbers. Each benchmark-run execute 90000 iterations. The multi-core setup uses the *Sysbench* [5] tool. Sysbench is a scriptable *multi-threaded benchmark tool* that creates complex time-based workloads. In our testing, we used $t = 10$ in order to generate multithreaded benchmarks of 10 seconds lengths. Fig. 1 presents the output of the tool with multiple runs of the benchmarks.

We performed testing runs using both setups and identified 17 statistically-relevant HEMs, *i.e.*, whose values were orders of magnitude larger than the remaining HEMs. Specifically, we focused on: br_mis_pred, br_pred, bus_access, bus_cycles, cpu_cycles, instr_retired, l1d_cache, l1d_cache_wb, l1d_cache_refill, l1i_cache, l1i_cache_refill, l2d_cache, l2d_cache_refill, ld_retired, mem_access, pc_write_retired, and st_retired. These HEMs have been allocated to 21 different sub-experiment, and each HEM is part of *5 sub-experiments*. Hence, there are 50×5 measurements for each of the hardware monitors.

The matrices in Fig. 2 and Fig. 3 underline the correlation between couples of HEMs $\{h_i\}$ and $\{h_j\}$. In each correlation *box*, the the upper value is the empirical correlation $\hat{\sigma_{ij}} = \hat{\sigma}_i \times \hat{\sigma}_j \times \hat{\rho}_{ij}$ and the lower one is the MVGD-based correlation after the MUCH data processing.

As expected, data from the predictable synthetic single process CPU-intensive benchmark presents very different correlation than the sysbench multiprocessing benchmark.

The single-core benchmark (Fig. 2) manifests a strong correlation between HEMs, mostly describing partial linear relationships between *HEMs* belonging to the *same context*, such as *cache misses*, or *branch predictions*. Instead, the multi-core benchmark (Fig. 3) presents less correlated data, including negative relationship between pairs of HEMs.

B. Accuracy Evaluation

We focus on the accuracy of the framework with respect to the empirical correlation of sub-experiment data and to potential errors in predicting HEM values in later benchmark executions.

We assessed the distribution of the delta between empirical pairwise correlation and MVGD sampled pairwise correlation

[1]The framework was also tested on Huawei Taishan Servers using a proprietary OS. We cannot disclose information on this setup.

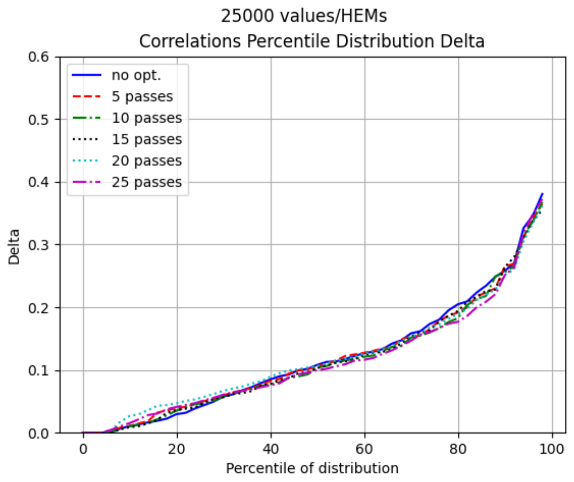


Fig. 4: Correlation delta as probability density function for single-core benchmark and 25000 values/HEM

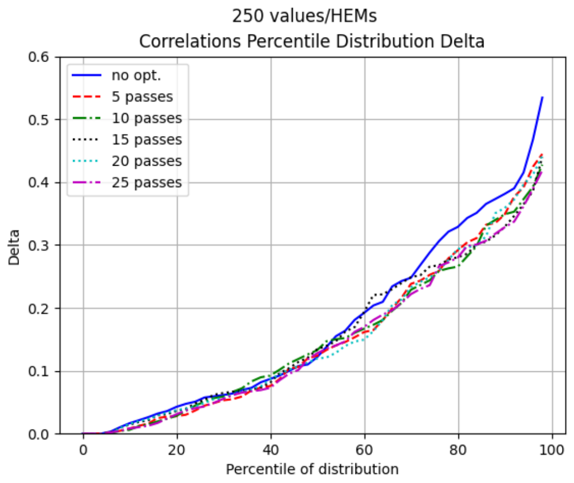


Fig. 5: Correlation delta as probability density function for single-core benchmark and 250 values/HEM

between couples of HEMs. This quantifies how much the reconstructed samples differ from the actual data after MVGD processing. Fig. 4 and Fig. 5 show the distribution of the correlation deltas for the single-core benchmark and 25000 and 250 samples/HEM respectively. Overall, we found the correlation delta to depend on the number of values sampled for each HEM and reaching 0.4 (0.6) for 25000 (250) samples of each HEM. As noted in [20], we also observed that the experimental and the reconstructed values depends on the number of iterations (*optimization steps*) performed during sorting to obtain the re-arranged full-wide HEMs vectors ((step (5) in Sec. III-B). In our experiments, the optimization step had a lower impact than the number of collected samples per HEM. This is visible in Fig. 5 that shows a higher variability of the correlation delta when only 250 samples/HEM are profiled.

TABLE I: MAPE: CPU-intensive single-process benchmark

MAPE: CPU-intensive single-process benchmark			
HEM name	MLR	MLP	RF
br_pred	0.000545	0.164585	0.000460
bus_cycles	0.002273	0.009775	0.006071
inst_retired	0.000349	0.012472	0.000330
l1d_cache	0.000454	0.011996	0.000347
l1d_cache_wb	0.036503	51.105616	0.014636
l1i_cache	0.007588	0.014341	0.003902
l1i_cache_refill	0.433256	4.051663	0.459776
l2d_cache	0.350933	2.164955	0.674850
l2d_cache_refill	0.037171	28.199508	0.027867
ld_retired	0.000535	0.065614	0.000350
mem_access	0.000633	0.011904	0.000549
pc_write_retired	0.000478	0.098950	0.000509
st_retired	0.000262	0.012920	0.000346
	0.870979	85.924299	1.189993

C. HEM Clustering

We used the framework to identify cluster of HEMs that can best characterize an application, despite the limited number of monitored PMCs in one run. We defined HEM clusters to contain the n HEMs that maximize the sum of *maximum absolute correlation values* with regards to every other HEMs in an experiment.

For the single core benchmark experiment, we discover that branch predictions, L1 cache counters, and CPU cycles counters (br_immed_retired, br_mis_pred, br_pred, cpu_cycles, l1d_cache_refill, l1i_cache_refill) are the most suited to capture the behavior of the benchmark with a maximum absolute correlation value of $\sim$ 7.866.

For the sysbench multi-core benchmark, the best cluster includes the bus access and the write-retired counters (br_immed_retired, bus_access, l1d_cache_refill, l1i_cache, l1i_cache_refill, pc_write_retired) with a maximum absolute correlation value of $\sim$ 6.874.

D. AI-based HEM Prediction

Arm MUCH provides a full statistical analysis of the correlation between HEMs. We used this *data set* to train three machine-learning models—Linear Regression (MLR), Multilayer Perceptron (MLP), Random Forest (RF)—and to evaluate the capability of the framework in *predicting* non-monitored HEM values.

After training on the set of re-arranged HEM vectors, *i.e.*, after applying MUCH, we fed the networks with subsets of HEM values coming from the HEM clustering experiments, and evaluated the accuracy of the networks in generating *non-monitored* HEMs. We used the full values of the HEM-clustering experiments as *empirical reference*. The accuracy of the *forecasted* values has been assessed using the *Mean Absolute Percentage Error (MAPE)*.

Tables I and II report the MAPE accuracy for MLR, MLP, and RF networks in predicting HEM values that were not measured during a benchmark run. In the tables, the best values are marked in blue color, while the ones exhibiting more than 10% MAPE error are marked in red. The values are the average of 10 experiments.

TABLE II: MAPE: Sysbench multithreaded benchmark

MAPE: Sysbench multiprocess benchmark			
HEM name	MLR	MLP	RF
br_pred	0.396886	0.003298	0.697910
bus_cycles	0.390227	0.161784	0.697271
cpu_cycles	0.299356	0.534736	0.697193
inst_retired	0.447805	0.234720	0.696650
l1d_cache	0.575274	0.330672	0.595609
l1d_cache_refill	47.127874	0.799111	0.350560
l1d_cache_wb	86.262104	10.248902	0.314104
l1i_cache	0.130094	0.004722	0.696297
l2d_cache_refill	40.071846	186.542361	0.291218
ld_retired	11.226796	1.327011	0.585759
mem_access	10.733564	0.425685	0.596006
pc_write_retired	0.341011	0.027340	0.696904
st_retired	4.933629	0.431589	0.607128
	202.936466	201.071931	7.522609

The Sysbench multithreaded benchmark (Table II) clearly shows that RF outperforms MLP and MLR. Notably, MLR predictions exceed five times the threshold (10) and MLP can produce very high errors (186.54 for l2d_cache_refill predictions). The CPU-intensive benchmark (Table I) confirms the trends. The results are preliminary, as more training and experiments will be needed to fully assess the capability of the framework. Nonetheless, the approach seems promising.

V. Conclusion

In this paper, we have presented Arm MUCH, a framework for the Armv8 architecture that adopts the MUCH [20] approach to overcome the limitations of modern PMUs that only allow a reduced number of HEMs to be monitored simultaneously.

We have validated the applicability of MUCH to Arm in contexts that include a complex operating system (Linux) and non trivial benchmark applications. Furthermore, we have investigated extensions of MUCH to i) derive minimal sets of HEMs that can best characterize the runtime behavior of an application, and ii) predict values of non-monitored HEMs using AI-networks trained on the full set of statistical data.

Our results confirm that Arm MUCH can capture the correlation between HEMs observed in different runs with adequate accuracy. Our experiments on clustering and prediction presented promising initial results that are worth investigating in future works, potentially in combination with tools to detect and analyze hardware noise,[2] to better understand whether the currently achieved accuracy can be improved. Additionally, the Random Forest approach to predict HEM values would benefit from a larger training set and more iterations.

In the future, we would also like to investigate the integration with custom eBPF programs and user-defined *uprobe* hooks to easily and precisely access performance counters.

References

[1] Python Numpy: documentation. https://numpy.org/doc/stable/reference/index.html#reference.

[2] Python Pandas: Dataframe. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html.

[3] Python Scipy: documentation. https://docs.scipy.org/doc/scipy/.

[4] Python Sklearn: documentation. https://scikit-learn.org/stable/modules/classes.html.

[5] Sysbench: repository. https://github.com/akopytov/sysbench.

[6] ARM. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/ Accessed: 2023-05-01.

[7] ARM. Arm Architecture Reference Manual Supplement. Memory System Resource Partitioning and Monitoring (MPAM) for Armv8-A. https://developer.arm.com/docs/ddi0598/latest Accessed: 2023-05-01.

[8] ARM. Arm CoreSight Architecture. https://developer.arm.com/Architectures/CoreSight Architecture Accessed: 2023-05-01.

[9] ARM. Arm Statistical Profiling Extension. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/statistical-profiling-extension-for-armv8-a Accessed: 2023-05-01.

[10] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38, 2019. doi:10.1109/SP.2019.00021.

[11] Trevor Hastie, Rahul Mazumder, Jason D. Lee, and Reza Zadeh. Matrix completion and low-rank svd via fast alternating least squares. *J. Mach. Learn. Res.*, 16(1):3367–3402, jan 2015.

[12] Intel. Resource Director Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html Accessed: 2023-05-01.

[13] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 1–14, 2019.

[14] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 45–54, 2013.

[15] Laurence H. Mutuel, Xavier Jean, Vincent Brindejonc, Anthony Roger, Thomas Megel, and E. Alepins. Assurance of Multicore Processors in Airborne Systems. Technical Report DOT/FAA/TC-16/51, FAA and Thales Avionics, 2017.

[16] NVIDIA. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems Accessed: 2023-05-01.

[17] Xing Pan and Frank Mueller. Controller-aware memory coloring for multicore real-time systems. SAC '18, page 584–592, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167132.3167196.

[18] Benjamin Recht. A simpler approach to matrix completion. *J. Mach. Learn. Res.*, 12(null):3413–3430, dec 2011.

[19] Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores . In *2022 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 133–145, 2022.

[20] Sergi Vilardell, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Much: Exploiting pairwise hardware event monitor correlations for improved timing analysis of complex mpsocs. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 511–520, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3412841.3441931.

[21] Sergi Vilardell, Isabel Serra, Roberto Santalla, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Hrm: Merging hardware event monitors for improved timing analysis of complex mpsocs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3662–3673, 2020. doi:10.1109/TCAD.2020.3013051.

[22] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 155–166, 2014.

[23] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

[2]https://www.kernel.org/doc/html/latest/trace/osnoise-tracer.html

Joint Time-and Event-Triggered Scheduling in the Linux Kernel

Gautam Gala, Isser Kadusale, and Gerhard Fohler
University of Kaiserslautern-Landau (RPTU), Germany
{gala, kadusale, fohler}@eit.uni-kl.de

***Abstract*—There is increasing interest in using Linux in the real-time domain due to the emergence of cloud and edge computing, the need to decrease costs, and the growing number of complex functional and non-functional requirements of real-time applications. Linux presents a valuable opportunity as it has rich hardware support, an open-source development model, a well-established programming environment, and avoids vendor lock-in. Although Linux was initially developed as a general-purpose operating system, some real-time capabilities have been added to the kernel over many years to increase its predictability and reduce its scheduling latency. Unfortunately, Linux currently has no support for time-triggered (TT) scheduling, which is widely used in the safety-critical domain for its determinism, low run-time scheduling latency, and strong isolation properties. We present an enhancement of the Linux scheduler as a new low-overhead TT scheduling class to support offline table-driven scheduling of tasks on multicore Linux nodes. Inspired by the Slot shifting algorithm, we complement the new scheduling class with a low overhead slot shifting manager running on a non-time-triggered core to provide guaranteed execution time to real-time aperiodic tasks by using the slack of the time-triggered tasks and avoiding high-overhead table regeneration for adding new periodic tasks. Furthermore, we evaluate our implementation on server-grade hardware with Intel Xeon Scalable Processor.**

***Index Terms*—Time-triggered, Event-triggered, Real-time, Scheduler, Linux**

I. Introduction

Real-time (RT) Safety-critical industries such as Thales (railway, [1]) and Airbus (aerospace, [2]) are exploring commercial-off-the-shelf (COTS) hardware platforms to benefit from reduced time to market, lower SWaP and higher computational power. Linux is interesting for COTS platforms due to extensive industrial and academic research, rich COTS hardware support, open-source versatility, and flexibility. It is free to run, study, modify to fit special requirements, and port to different hardware platforms. There are many useful open-source or freely available libraries, development tools, and applications created for Linux. Using Linux can help to keep costs down. Developers and system designers from all domains find the ability to adapt Linux systems to their requirements and hardware platform advantageous.Linux has a large open-source community that freely publishes bugs (possibly with fixes) and enhancements to Linux for all to benefit and a possibility for future maintenance by the community. Thus, using Linux often ensures we save efforts to re-invent what already exists.

Linux (in conjunction with KVM) is used to power most of the public cloud infrastructure and is heavily used in supercomputing [3]. With the advent of Industry 4.0, smart healthcare devices, software-defined vehicles, and other cloud-connected transportation (e.g., trains and trams), OEMs of safety-critical domains will find using a single Linux operating systems for cloud/edge servers, IoT/smart devices, and in-vehicle software beneficial. A single operating system will help reduce the costs and the burden of managing multiple suppliers, maintaining many development environments, and numerous application versions.

Using existing, often proprietary, RT operating systems (RTOSs) or hypervisors is expensive, lacks flexibility and rich hardware support of Linux, can lead to vendor lock-in, and needs tedious re-implementing of existing libraries, tools, and applications for these RTOSs.

Safety-critical industries have considered using RT-capable Linux, e.g., NASA for ground and space operations [4], Thales for an RT-capable cloud to support railway operations [1], and various automotive companies for Linux-based fully open software stack for the connected car [5]. The Linux community has made significant efforts to support RT applications, e.g., PREEMPT_RT [6] and Xenomai [7]. These efforts have focused on improving determinism and reducing latency in the Linux Kernel [8]. The current Linux kernel scheduler supports Constant Bandwidth Server (CBS) based resource reservations over an Earliest-Deadline First (EDF) scheduler to improve RT guarantees [9].

Unfortunately, the Linux kernel scheduler still does not support (table-driven) time-triggered (TT) scheduling. Safety-critical RT systems often require TT scheduling. Since the creation and validation of the scheduling table occur offline in the TT schedule, a system designer can factor in complex constraints such as latency and precedence constraints, which would otherwise incur large overheads to handle directly at run time. TT scheduling also enforces strong temporal isolation between tasks. It can easily ensure that a task overrunning its worst-case execution time (WCET) does not hamper other tasks' schedulability. A system using TT scheduling is highly predictable as events occur pre-planned at fixed points in time. Thus, testing and certifying the system is easier as there are only a few predictable scenarios to consider.

A TT scheduler mainly consists of a dispatcher that assigns resource(s) to task(s) based on an offline computed schedule (during the design phase). The scheduler receives this schedule as a *scheduling table* consisting of all the necessary scheduling decisions and the point in time the dispatcher should implement those decisions. Since a TT scheduler is quite simple, it has much lower overheads as compared to event-triggered (ET) schedulers (e.g., EDF-based), especially during peak load conditions [10]. Moreover, ET schedulers can produce widely different schedules for the same system when the sequence or timing of events changes, leading to lower predictability. A system with an ET scheduler requires exhaustive testing using simulated loads, considering even the rarest events. However, proving that the tests covered all possible scenarios that may occur at runtime is not easy.

In this paper, we build upon some basic ideas described in [11] and propose a new TT CPU scheduler for Linux. The main contributions are:

1) For ensuring static scheduling guarantees and minimizing scheduling latencies, we provide the ability to schedule periodic tasks (Linux threads, processes, VMs, or containers) in Linux using TT (table-driven) scheduling.
2) Inspired by Slot-shifting [12], we combine the flexibility to execute RT aperiodic (AP) and best effort (BE) tasks at run time in the slack of the periodic TT tasks, and thus, ensure efficient utilization of resources. We provide the option to add new periodic tasks to the offline table without the need for a high-overhead scheduling table regeneration.

We implement our approach as a new low-overhead Linux scheduling class (SCHED_TT) with a separate Slot-Shifting Manager (SSM) Kernel Module to help integrate RT AP tasks. We present an overview of the actual implementation and provide the scheduling overheads by performing experiments on a COTS server with Intel Xeon Processor (Cascade Lake).

The proposed TT CPU scheduler for Linux can be easily integrated with existing real-time resource management and orchestration frameworks to give static scheduling guarantees to tasks, VMs [1], [13], and containers [14].

II. Related Work

A. Real-time and Linux

PREEMPT_RT [6], an open-source patch for Linux Kernel, is generally used to create real-time Linux. The patch focuses on providing mechanisms to reduce latency and increase determinism in the Linux kernel and complements existing Linux scheduling. Many of the patches are now part of the mainstream Linux kernel. We use a Linux kernel with PREEMPT_RT as a base for further development.

Efforts from ACTORS EU project [15] followed by the work from Lelli et al. [9], and others led to the introduction of SCHED_DEADLINE in the Linux kernel scheduler to support CBS-based resource reservations over an EDF scheduler.

Litmus^{RT} provides a real-time extension for the Linux kernel to act as an experimental testbed for real-time research, especially on multiprocessor real-time scheduling and synchronization. It supports various clustered, partitioned, global, and semi-partitioned schedulers such as partitioned-EDF and partitioned fixed-priority scheduling.

Approaches such as RTAI [16], Xenomai [7], and RTLinux [17] use a hardware abstraction layer below the Linux kernel. The Linux kernel runs as the lowest priority (background) thread on top of this layer. Similarly, the Jailhouse partitioning hypervisor [18] runs bare metal and cooperates closely with Linux to run safety-critical applications. However, guests cannot share a CPU because Jailhouse has no scheduler. None of these approaches support running TT tasks natively by Linux.

B. Existing TT scheduling support in OSes or hypervisors

TTTech MotionWise Platform [19] emulates a TT scheduler in userspace via standard POSIX system calls. However, this solution has high scheduling latencies and jitter compared to a kernel-level implementation [20].

Specialized real-time hypervisors, such as XtratuM [21] and PikeOS [22], support TT schedule. However, they are often proprietary and have limited hardware and software support. Moreover, they are unsuitable for cloud environments as they support only a handful of guest operating systems and have additional limitations, such as limited CPU models for VMs.

Xen with the ARINC 653 scheduler [23] only considers cyclic scheduling on a single core. The Tableau [24] extension to the Xen hypervisor introduces support for TT scheduling of VMs to reduce scheduling overheads and enable high-density packaging of VMs. However, Tableau is not explicitly targeted to execute periodic safety-critical real-time VMs and uses a table regeneration process to add new tasks. Gala et al. [1] demonstrated that Xen has, in general, higher overheads than Linux (+ PREEMPT_RT patch) in conjunction with KVM. Therefore, KVM/Linux is better suited for low latency RT applications than Xen.

Very recently, Karachatzis et al. [20] presented an implementation and evaluation of a kernel-level time-triggered scheduling approach for Linux. The approach proposed improving the node's utilization by allowing other tasks to run in the slack of TT tasks. In addition to allowing non-RT tasks in the slack of TT tasks, our approach supports the guaranteed execution of new RT tasks by appending them to the scheduling table at run-time (without needing to regenerate the table). Our approach takes advantage of the slot-shifting algorithm to add the required flexibility. Our approach keeps run-time overhead low by separating the TT task dispatching via newly created SCHED_TT scheduling class (on TT cores) from the slot-shifting algorithm decision-making. The slot-shifting algorithm is executed on non-TT cores with a period equal to the slot length.

C. Joint scheduling of TT and ET RT tasks

Many previous scheduling algorithms have explored combining both time-and event-triggered approaches to take advantage of both the contrary scheduling approaches. Fohler [12] presented the Slot shifting algorithm for joint scheduling of TT and ET tasks. Schorr [25] presented a multiprocessor extension to the slot-shifting algorithm. However, previous works do not integrate these and other approaches (e.g., [26], [27]) to combine TT and ET tasks with the Linux kernel scheduler.

III. Background

A. Slot Shifting

Let us consider a global time whose progress is triggered by equidistant events. We consider a sparse time base where the time duration separating two of these events is called a *slot* A multicore slot on a cloud node can run N tasks at a time, where N is the total number of CPU cores scheduled under the TT schedule. For simplicity of explanation, we assume each task requires only one core. Thus, a slot consists of CPU allocation in time units (e.g., milliseconds) and a mapping of tasks to TT cores. We used an existing heuristic algorithm to create the mapping in the form of an offline scheduling table. The heuristic must know all tasks in the system before run time to create the scheduling table. We must store the scheduling table in the node. The node's scheduler uses it to execute tasks once the node boots. At run time, the TT scheduler cannot handle a task that is not present in the offline scheduling table.

Since the exact activation times of sporadic or aperiodic RT tasks are unknown before run time, we must consider them periodic when creating the scheduling table. Moreover, if the system designers need to add a new task to an existing scheduling table, they must recompute it. While creating the scheduling table, the task resource assignment occurs based on the worst-case resource demand. In the average case, most tasks utilize only a fraction of the allocated resources at run time.

The slot-Shifting [12], [25] algorithm provides a way to run or add AP RT tasks in the slack of TT tasks at run time without needing an expensive table regeneration process. Slot shifting defines *capacity interval* (or simply *interval*) as a set of consecutive slots that possess the exact mapping of tasks to cores. In other words, a new interval starts on a slot when the set of tasks assigned to this slot differs from the previous one. Intervals may contain some cores without any task assignment, resulting in idle slots within the interval. These slots form the *Spare Capacity (SC)* of the interval. The scheduler uses SC to execute AP RT tasks with guaranteed CPU execution time.

The scheduler executes any task assigned to the current interval (as per the scheduling table). In idle slots, it runs any ready tasks defined in the scheduling table that do not belong to the current interval. If no task is ready, the scheduler executes a best-effort or idle task. The scheduler reduces the run time SC at the end of every slot where a task assigned to the current interval does not execute. The run time SC increases when a task assigned to the current interval finishes before its WCET.

Slot shifting define *acceptance test* and a *guarantee routine* to handle AP RT tasks [25]. When a new AP RT task arrives, the acceptance test determines the sum of the remaining SC in the current interval, spare capacities in all the intervals before the task's deadline, and the usable SC in the interval where the deadline of the AP RT task lies. Then it checks if the determined sum is more significant than the WCET of the task (in terms of slots). If the test is successful, the guarantee routine adds the task temporarily to the scheduling table and updates the spare capacities of all affected intervals. The guarantee routine may need to split existing intervals. The result is that this routine guarantees the CPU allocation to the newly accepted AP RT task. Using a similar idea, we can permanently add a new periodic RT task to the scheduling table without requiring a high-overhead table regeneration process by considering offline SC values (instead of online values).

B. Linux scheduler

The Linux scheduler is part of the kernel that helps manage tasks and decide which to run next. It is modular and designed to be extensible [28]. It is a multi-queue scheduler that maintains a per-core run queue of ready tasks. It consists of a core scheduler and extensible modules called the scheduling classes, as shown in figure 1. Each class encapsulates a scheduling policy to decide which ready task (belonging to that class) to run next (from the run queue).

Linux currently supports four main scheduling classes:

1) *Deadline (DL)* for SCHED_DEADLINE policy to support CBS-based resource reservations over EDF scheduling.
2) *Real-Time (RT)* for POSIX fixed-priority scheduling with SCHED_RR (round-robin) and SCHED_FIFO policies.
3) *Completely Fair Scheduling (CFS)* to maintain balance in allocating processor time to tasks.
4) *Idle* for scheduling very low-priority jobs, usually the idle process.

These classes are hierarchically organized by priority:

$$DL > RT > CFS > IDLE$$

The core scheduler selects which task to run next from the run queue on each core by searching through the scheduling classes in decreasing order of priority. As a result, the tasks of lower priority classes run in the idle time of higher priority classes. We add a new highest priority TT scheduling class ($TT > DL$).

IV. Implementation

Let us assume a multicore processor with N CPU cores $(C_0, C_1, \dots, C_{N-1})$. We let M cores $(C_{N-M}, C_{N-M+1}, \dots, C_{N-1})$ out of these N cores $(M < N)$ to execute process belonging to the TT class (SCHED_TT policy). We will refer to them as TT cores.

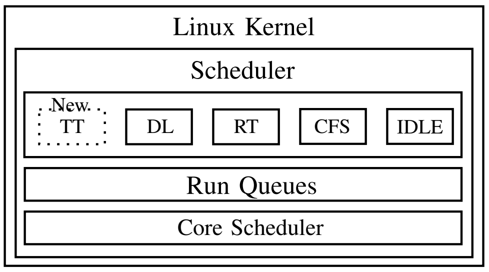


Fig. 1: Relevant parts of Linux CPU Scheduler

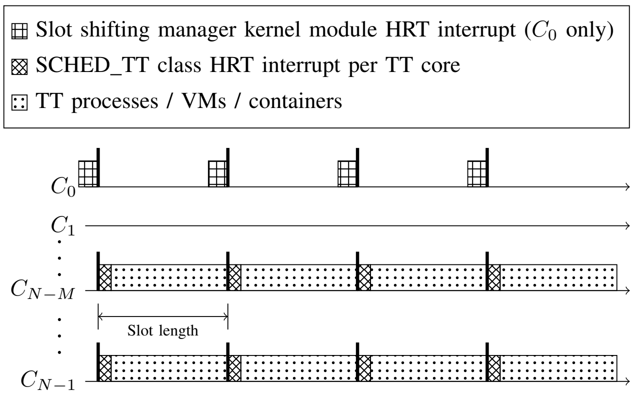
There will be at least one core (C_0) that will not run TT processes. We will refer to it/them as non-TT core(s). We also assume that each TT task only has a single thread.

To obtain real-time behavior from the Linux kernel (low latency and high determinism), we chose the fully preemptible kernel (RT) model via kernel configuration during build time. The fully preemptible kernel is available via the PREEMPT_RT patch. This model ensures that preemption is possible on almost all kernel code apart from some critical sections and implements mechanisms to reduce preemption. The Linux kernel performs significant asynchronous housekeeping work, such as timekeeping, timer callbacks, interrupt handlers, RCU callbacks, and kernel threads. Using standard Linux kernel configuration, every core is assigned housekeeping work. The "noise" from this housekeeping work can significantly impact applications running on the TT cores.

The *isolcpus* parameter helps us to isolate the TT cores from the general SMP balancing and scheduler algorithms. Similarly, we initialize the $nohz\_full$ to configure $full\ dynticks$ along with CPU Isolation, rcu_nocb_poll to offload RCU processing to non-TT cores, and the $irqaffinity$ parameter to affine the IRQs to non-TT cores (e.g., $nohz\_full = C_{N-M}, C_{N-M+1}, \ldots, C_{N-1}$). Once the system boots, Linux ensures that no processes execute on these TT cores unless instructed by the slot-sifting manager (SSM) kernel module and restricts all housekeeping work to non-TT cores ($C_0, \ldots, C_{N-M-1}$). Thus, we ensure almost housekeeping noise-free TT cores to run processes assigned to SCHED_TT policy as depicted in Figure 2. After the system boots, we insert SSM and ensure it runs on a non-TT core (e.g., C_0).

The SSM runs as a timer interrupt routine just before the start of each slot. This module runs a partitioned slot-shifting algorithm. It decides what should run in the upcoming slot on each TT core and informs the decision to TT scheduling class via a shared structure.

The TT scheduling class acts as the interface between the SSM and the core Linux scheduler. It runs as a periodic timer interrupt routine at the start of every slot. It checks the shared structure set by SSM to check which task to run in the current slot. If the task to run is not the same as the task in the previous slot, it marks the core for rescheduling by the Linux



⊞ Slot shifting manager kernel module HRT interrupt (C_0 only)
▩ SCHED_TT class HRT interrupt per TT core
⬚ TT processes / VMs / containers
C_0
C_1
C_{N-M}
Slot length
C_{N-1}

Fig. 2: SCHED_TT dispatcher and SSM kernel module

```
#define SCHED\_DEADLINE 6   // Existing
#define SCHED\_TT       7   // New
struct task_struct {
  ... // Existing task structure
  #ifdef CONFIG_SCHED_TT_POLICY
    unsigned int         TT_id;  // New
  #endif
}
```

Listing 1: TT class macro

```
struct rq { // Existing RQ structure
  ...
  struct dl_rq   dl;  // Existing
  #ifdef CONFIG_SCHED_TT_POLICY
    struct tt_rq      tt;          // New
  #endif
}
struct tt_rq {  // New
  ...
  bool  scheduler_running;
};
```

Listing 2: TT class run queue

core scheduler. The following sections explain the detailed implementation of the TT scheduling class and SSM.

A. *TT scheduling class*

We have developed this scheduling class from scratch to implement the SCHED_TT policy and enable TT scheduling in Linux. We have implemented it on top of Linux kernel version v5.19.9 with PREEMPT_RT patch.

We defined a macro *SCHED_TT* in "include/linux/sched.h" to declare the new highest priority scheduling class (see Listing 1). We also add a new variable to the task structure (process descriptor) to act as a logical identifier for a task belonging to the TT class.

The information about all ready processes is present in a per-core run queue (RQ) data structure (*struct rq*), which in turn has a TT RQ data structure (*struct tt_rq tt*) to store information about ready TT tasks on the core (see Listing 2). This structure also has a variable the slot-shifting kernel module uses to indicate to the TT scheduler if it should start running.

We made minor additions to the Linux core scheduler code to ensure the kernel knows the new highest priority class and does the additional logging for the TT class if requested. For example, we defined a macro *SCHED_TT* in "include/linux/sched.h" to declare the new highest priority scheduling class. We also add a new variable to the task structure (process descriptor) to act as a logical identifier for a task belonging to the TT class. The information about all ready processes is present in a per-core run queue (RQ) data structure (*struct rq*), which in turn requires a TT RQ data structure (*struct tt_rq tt*) to store information about ready TT tasks on the core.

The modular Linux scheduler requires each scheduling class to implement some functions specified in the Class structure.

```
DEFINE_SCHED_CLASS(tt) = {
  // enqueue new runnable TT task in Linux RQ
  .enqueue_task    = enqueue_task_tt,
  // dequeue stopped TT task from Linux RQ
  .dequeue_task     = dequeue_task_tt,
  // pick next appropriate task to run from
  // the TT scheduling class
  .pick_next_task = pick_next_task_tt,
  ...
```

Listing 3: TT class definition

```
void __init init_sched_tt_class(void){
  // set the tick equal to the slot length
  tt_set_tick_period_ms(TT_TICK_PERIOD);
  ...
  //Set a timer on each TT core
  for_each_possible_cpu(i) {
    tick_timer = per_cpu_ptr(&tt_tick_timer,i);
    hrtimer_init(tick_timer,CLOCK_MONOTONIC,
    HRTIMER_MODE_ABS_HARD);
    //function to call on each tick
    tick_timer->function = tt_cpu_tick;
  }
  ...
}
```

Listing 4: TT class initialization

```
static enum hrtimer_restart
tt_cpu_tick(struct hrtimer *timer){
  int cpu = smp_processor_id();
  ...
  //get the task id of current and next task
  new_sched_tid=this_cpu_read(tt_sched.next);
  prev_sched_tid=this_cpu_read(tt_sched.curr);
  if(new_sched_tid!=prev_sched_tid){
    // only if the previous slot task is not
    // same as the task for this slot
    if (new_sched_tid)
    sched_task = ss_tasks[new_sched_tid];
    else if (sched_task
             && task_cpu(sched_task)!= cpu){
      //If the task is on different core's RQ,
    } //move it to this one's RQ
    ...
    //mark for resched by Linux core sched
    resched_curr(rq);
  }
  //do not do anything if the task in previous
  //slot and the current one are the same
  ...
  // restart HRT for the next slot tick
  return HRTIMER_RESTART;
}
```

Listing 5: TT class CPU tick (per core)

Some crucial functions of the TT scheduling class to interface with the core Linux scheduler are show in Listing 3.

During kernel boot, the core Linux scheduler expects an interface (Listing 4) to initialize the new scheduling class. The initialization of the TT class includes setting up a per-core high-resolution timer (HRT) to have precise timer interrupts at slot boundaries.

The main job of the HRT timer callback (Listing 5) on each core is to check if the task id assigned by SSM to the current slot for this core differs from the previous one. If it is different, it marks the current core for rescheduling using resched_curr(). As a result, the Linux core scheduler runs and uses the pick_ next_task_tt() interface to get the pointer to the new task to run. It performs a context switch and then runs the new task in the current slot on that core. In case this interface of the TT class does not provide a task to run, the Linux core scheduler automatically looks for runnable tasks of lower-priority scheduling classes. The same is valid if the TT class tasks finish execution early.

Only non-DL scheduling classes should be used together with the TT class. It is possible to execute new RT aperiodic tasks in the slack of TT tasks via the SSM module.

B. *Slot Shifting Manager (SSM) kernel module*

We assume an external tool handles some things offline, such as precedence constraint resolution of tasks, calculations of the earliest start times and deadlines, and allocation of tasks to cores. The user must provide the information received from the external tool to the SSM kernel module via *sysfs* interface. sysfs is an in-memory file system that allows users to interface with kernel objects such as devices, modules, and other components.

The SSM sysfs interface allows users to input task count and properties of tasks and perform parts of offline phase specific to the slot shifting algorithm, checking current status, and initiating TT scheduling on TT cores. Before initiating TT scheduling, the user must use our task execution tool to execute the task binaries as Linux processes and map the Linux process identifiers (PIDs) to the slot-shifting task IDs. The tool sets the scheduling policy for the tasks to SCHED_TT.

Upon initiation of TT scheduling, SSM performs steps to prepare for slot-shifting and then sets up an HRT timer on core 0 to have precise interrupts before the start of the new slot. Let us assume $WCET_{ssm}$ is the worst-case observed execution time of the SSM module (based on experiments - Section V). The module sets the HRT timer to cause an interrupt $WCET_{ssm}$ time units before the slot boundary.

The HRT interrupt callback runs a loop to perform the following activity for managing each TT core. We have summarized the SSM HRT callback in Algorithm 1. Firstly, it performs some slot-shifting housekeeping activities such as incrementing slot numbers and updating intervals ($Update\_SlotShifting\_Intervals()$), removing finished tasks, and moving ready periodic tasks to the internal SSM ready queue ($Update\_Interal\_SlotShifting\_RQ()$). Then, it checks for any new (user-added) periodic or aperiodic tasks that became active. The module must try to add them to the existing scheduling table ($Check\_and\_Add\_New\_RT\_Tasks()$). It performs an acceptance test to check if enough spare capacity is present to add these tasks. If the test is successful, the guarantee routine

Algorithm 1 SSM HRT timer callback

```
function SSMOD_TIMER_CB(STRUCT HRTIMER *TIME)
    $core := N - M$
    while $core < N$ do
        $Update\_SlotShifting\_Intervals()$
        $Update\_Interal\_SlotShifting\_RQ()$
        $Check\_and\_Add\_New\_RT\_Tasks()$
        $Set\_Upcoming\_Slot\_Task()$
        $Update\_SlotShifting\_Spare\_Capacities()$
        $core := core + 1$
    end while
end function
```

adds the task temporarily/permanently to the scheduling table and updates the spare capacities of all affected intervals. Next, it selects the task to run in the next slot of the particular CPU core and updates the internal data structure accessed via the TT scheduling class at the start of each slot ($Set\_Upcoming\_Slot\_Task()$) to find out which task to run in that slot. Finally, it updates the spare capacity as required ($Update\_SlotShifting\_Spare\_Capacities()$). Since we are running a partitioned slot-shifting algorithm, the HRT callback repeats these steps for each TT core.

V. Experiments

We evaluated our implementation on a Dell COTS server with an Intel Xeon processor (Cascade Lake, 16 physical CPU cores, $2.3GHz$). We turned off certain configurations, such as power-saving mechanisms (e.g., frequency scaling, C-states) and hardware multi-threading, in BIOS and Linux Kernel to avoid unpredictability sources for the TT cores. Using power-saving mechanisms such as low power stats leads to higher scheduling overhead, while frequency scaling leads to an increase in the WCET of tasks. Kadusale et al. [29] present an energy-aware slot-shifting algorithm variation that considers lower power states and frequency scaling. However, we do not consider energy-aware slot-shifting in this paper. We ran the slot-shifting manager (SSM) on core 0 and isolated cores 1 to 15 to run offline scheduled TT tasks as explained in Section IV.

We specify parameters (Table I), such as the total target utilization, WCET range, and period range, to the UUnifast algorithm [30] to generate task sets for evaluation. Each task in every task set performs many arithmetic operations (in a loop) and uses the clock cycle performance counter event to determine the time needed to perform all the operations in each iteration of the loop. Offline scheduled tasks have total utilization of ca. 50%. New AP RT tasks that the slot-shifting manager must add during runtime have an additional total utilization of 50%.

Parameter	Offline tasks	AP RT tasks
WCET range	[1,15]	[10,15]
Period range	[15,50]	[10,15]
Total Taskset Utilization	50%	50%

TABLE I: Taskset Parameters

We created 50 task sets via the UUnifast algorithm for the evaluation. We parse and feed the UUnifast algorithm output to the slot-shifting manager via the sysfs interface (see Section IV-B). Each task set has an offline schedule duration between 480 to 520 slots, with a slot length of 3ms. We ran each test 5 times. Thus, we measure the results in the following sections for scheduling overheads by the TT class and the slot-shifting manager by considering more than 120×10^3 slots $\times 15$ cores $= 18 \times 10^5$ slots in total. The main aim of these task sets is not to test the working for the slot-shifting algorithm (see previous work [25]) but to measure the worst-case overheads for SCHED_TT and estimate the overheads for SSM. We do not expect a change in SCHED_TT overheads with different utilization values.

A. TT class overheads

This section shows observed periodic overheads of the individual components of the TT scheduling class and its interaction with the Linux core scheduler. We measured all the overheads by reading elapsed clock cycles (PMU events) at the start and end of the appropriate code sections. These components are depicted in Figure 3. Task Tick (T) represents the overhead for the TT class HRT callback (Listing 5). Tick Skew (TS) represents the maximum difference in occurrence time of HRT time callback on each core. Schedule Trigger (ST) is the time needed for the Linux core scheduler to activate once the TT class HRT callback marks a core for rescheduling. $\_\_schedule$ function (S) is the primary function of the Linux core scheduler. This function is triggered on TT cores only when the TT class marks a core for rescheduling. It puts the previously running task into the appropriate RQ, calls TT class' $pick\_next\_task$, and lastly, performs a context switch before passing the control to the newly selected task. $pick\_next\_task$ function (P) is called by $\_\_schedule$ to pick a new task of the highest priority scheduling class to run next. In our case, the highest priority scheduling class is SCHED_TT. $pick\_next\_task$ in turn calls $pick\_next\_task\_tt$ (Listing 3).

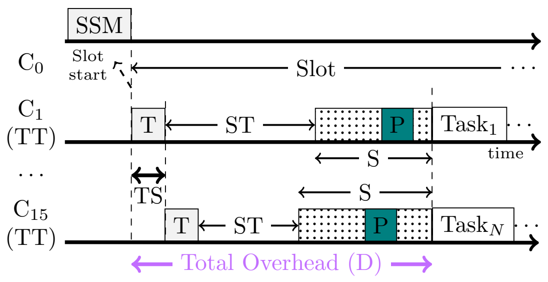


Fig. 3: TT class overhead components

We perform experiments with offline schedules that require and do not require migration of tasks between cores. We also compare our results to similar components from the existing RT scheduling class (SCHED_RR round robin policy), which is closest to TT class style scheduling. Results are shown in Figure 4. TT class performs slightly better for task tick without

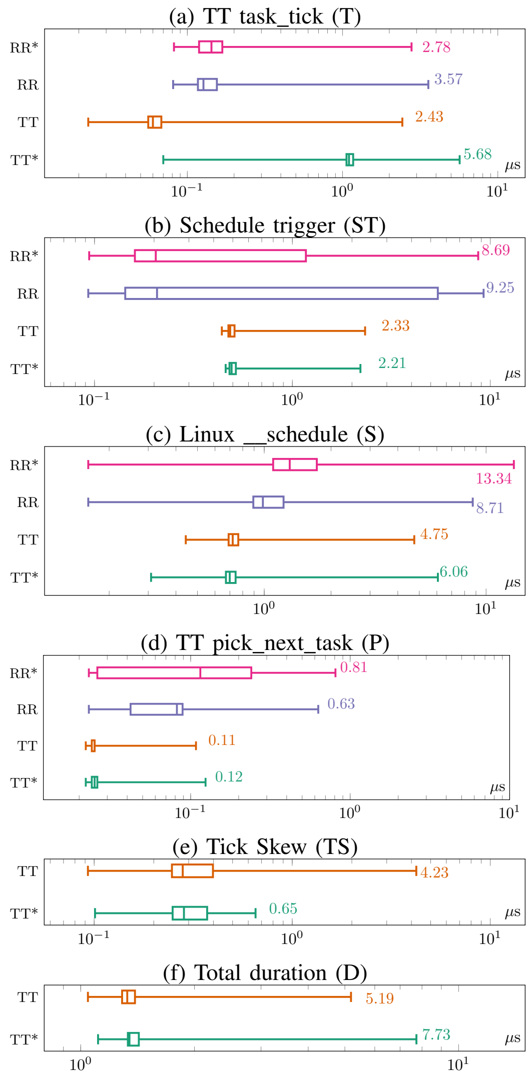


Fig. 4: Overhead measurements for TT class and RT class (RR) without and with(*) core migration

migration. However, the TT class performs slightly worse with migrations because we artificially generated an offline schedule that causes multiple migrations in the same slot. A similar schedule is challenging to simulate for SCHED_RR as it does task migration only for load-balancing purposes. We observed a maximum tick skew of $4.23\mu s$ for the TT class tick. We cannot make a meaningful comparison since RT class ticks are not precise. We observed lower overheads for the schedule trigger and *__schedule* function for the TT class. We expected little difference as most code (except some interfaces) is part of the Linux core scheduler. We observed a significant difference in *pick_next_task*, where the TT class has much lower overheads.

The maximum observed total overhead (D) for the TT class with and without migrations was $7.73\mu s$ and $5.19\mu s$ in the worst case, with more than 99% slots requiring $< 2.66\mu s$ and $< 2.47\mu s$, respectively. We targeted a slot duration of $3ms$. We read the clock cycles at the start of each slot and observed a

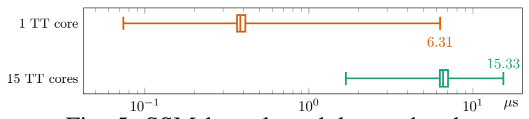


Fig. 5: SSM kernel module overhead

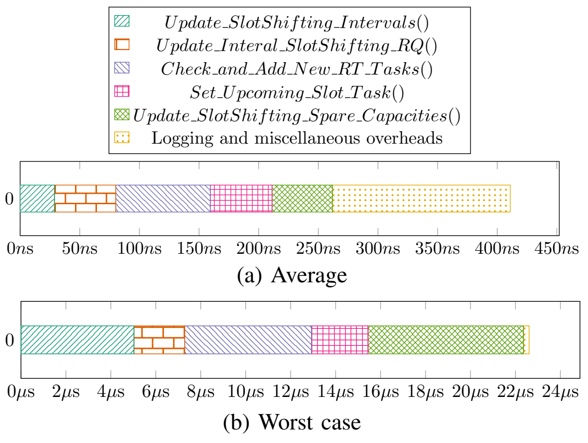


Fig. 6: Overheads of individual SSM functions

maximum slot duration of $3.003ms$, a minimum slot duration of $2.995ms$, and an exact $3ms$ duration for $> 99\%$ slots.

Comparison to existing work: Karachatzis et al. [20] experimentally demonstrate that their kernel-level TT scheduling implementation has lower scheduling latency ($max = 41.79\mu s$) as compared to the existing completely fair scheduling policies in Linux ($max = 12813.56\mu s$) and MotionWise [19] userspace TT implementation ($max = 85.67\mu s$). The experiments were conducted on a quad-core Intel Atom processor ($1.594GHz$) on Linux kernel v5.9.1 with PREEMPT_RT and optimizations for running RT tasks. We did not observe further details about how the scheduling latency was precisely determined to make a clear comparison. In our implementation, we observed a worst-case scheduling latency of $5.19\mu s$ (over 18×10^5 slots) with migration disabled (similar to them) on server-grade hardware (16 cores, $2.3Ghz$), Linux kernel v5.19.9, and similar patch and optimizations. We measured the latency by reading hardware clock cycles at appropriate time points within the Linux core scheduler.

B. SSM kernel module overhead

We measured the overheads for the SSM module by reading elapsed clock cycles (PMU events) at the start and end of the module. Figure 5 shows the observed periodic overheads of the SSM module running on core 0 and executing the slot-shifting algorithm for one single core and all 15 cores together. We also measured the overheads for the individual functions from Algorithm 1 by reading elapsed clock cycles (PMU event) at the start and end of the functions. Figure 6 shows the worst and average overheads of the individual functions. Similar overheads for slot-shifting algorithms were measured and explained by Schorr et al. [25]. Hence, we do not explain these results in detail in this paper.

Note that the worst case seems longer in Figure 6b as compared to Figure 5 since Figure 6b shows the cumulative worst case of individual functions. We use the worst-case observed value ($15.33\mu s$) to set up the HRT timer interrupt for SSM, as explained in Section IV-B.

We can observe that only a tiny portion of the overhead goes into interfacing with the TT class (to set tasks for upcoming slots). This indicates a possibility to integrate and test other joint TT and ET scheduling algorithms in the future instead of just allowing slot-shifting algorithms.

VI. Conclusion

We presented a new scheduling class for the Linux scheduler to support time-triggered (TT) scheduling of tasks on multi-core Linux nodes. We complemented the new scheduling class with a low overhead slot-shifting manager (SSM) running on a non-TT core to provide guaranteed execution time to real-time aperiodic tasks by using the slack of the time-triggered tasks and avoiding a high-overhead table regeneration for adding new periodic RT tasks. We have implemented the TT class for Linux kernel v5.19.9 with PREEMPT_RT patch and evaluated our implementation on server-grade hardware with Intel Xeon Scalable Processor. Our implementation has very low overheads and performs better than the existing RT class (SCHED_RR), especially for picking tasks to run next. We also observed that more than 99% of slots achieve the targeted slot length with a maximum error of $5ns$ in the rest. We observed extremely low worst-case TT class scheduling ($< 7.73\mu s$) and SSM overheads ($< 15.33\mu s$).

We are fine-tuning the implementation at present and working on making it open-source. In the future, we want to support multi-threaded / multicore tasks (process, VMs, and containers). In this paper, we focused on TT CPU scheduling alone. However, other sources of unpredictability exist, such as the memory hierarchy and network. Thus, we want to consider memory bandwidth and cache allocation to slots and explore integration with time-sensitive networking (TSN). We aim to make the SSM modular more generalized to allow integration of other joint time-and event-triggered algorithms. We leave the integration of TT Linux nodes in a distributed system or cloud to future work. It will require some form of clock synchronization with other nodes. Since embedded devices have a limited number of cores, we want to explore the possibility of avoiding non-TT cores by integrating SSM with SCHED_TT class.

Acknowledgment

We want to thank all the anonymous reviewers for their time and expertise in providing feedback on our paper. We highly appreciate their invaluable inputs and support.

References

[1] G. Gala, G. Fohler, P. Tummeltshammer, S. Resch, and R. Hametner, "RT-cloud: Virtualization technologies and cloud computing for railway use-case," in *2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC)*, 2021.

[2] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch, "Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*.

[3] G. Markus, B. Boisseau, and M. Sarrazin, "A CTO's guide to software-defined vehicles," in *Whitepaper*. Canonical.

[4] H. Leppinen, "Current use of linux in spacecraft flight software," *IEEE Aerospace and Electronic Systems Magazine*, 2017.

[5] P. Sivakumar, A. N. Lakshmi, A. Angamuthu, R. S. Devi, B. V. Kumar, and S. Studener, "Automotive grade linux: An open-source architecture for connected cars," in *Software Engineering for Automotive Systems*. CRC Press, 2022, pp. 91–110.

[6] *PREEMPT RT: Real-Time Linux Wiki*, available: https://rt.wiki.kernel.org/index.php/Main_Page, Last accessed: 03/21.

[7] P. Gerum, "Xenomai-implementing a rtos emulation framework on gnu/linux," *White Paper, Xenomai*, p. 81, 2004.

[8] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM Computing Surveys*, 2019.

[9] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, 2016.

[10] H. Kopetz, "Event-triggered versus time-triggered real-time systems," in *Operating Systems of the 90s and Beyond*. Springer, 1991, pp. 86–101.

[11] G. Gala, J. Castillo Rivera, and G. Fohler, "Work-in-progress: Cloud computing for time-triggered safety-critical systems," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 516–519.

[12] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *Proceedings 16th IEEE Real-Time Systems Symposium*, 1995, pp. 152–161.

[13] G. Durrieu, G. Fohler, G. Gala, S. Girbal, D. G. Pérez, E. Noulard, C. Pagetti, and S. Pérez, "DREAMS about reconfiguration and adaptation in avionics," in *ERTS 2016*, Toulouse, France, Jan. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01258701

[14] G. Monaco, G. Gala, and G. Fohler, "Shared resource orchestration extensions for kubernetes to support real-time cloud container," in *26th International Symposium On Real-Time Distributed Computing (ISORC'23)*, 2023.

[15] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An edf scheduling class for the linux kernel," in *Proceedings of the 11th Real-Time Linux Workshop*, 2009, pp. 1–8.

[16] J. Arm, Z. Bradac, and V. Kaczmarczyk, "Real-time capabilities of linux rtai," *IFAC-PapersOnLine*, vol. 49, no. 25, pp. 401–406, 2016.

[17] V. Yodaiken *et al.*, "The rtlinux manifesto," in *Linux Expo*, 1999.

[18] M. Baryshnikov, "Jailhouse hypervisor," B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2016.

[19] TTTech Computertechnik AG., *MotionWise: Safe Vehicle Software Platform*, available: https://www.tttech-auto.com/motionwise, Last accessed: June 2023.

[20] P. Karachatzis, J. Ruh, and S. S. Craciunas, "An evaluation of time-triggered scheduling in the linux kernel," in *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, ser. RTNS '23. ACM, 2023.

[21] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*. Citeseer, 2009, pp. 263–272.

[22] R. Kaiser and S. Wagner, "Evolution of the pikeos microkernel," in *International Workshop on Microkernels for Embedded Systems*, 2007.

[23] S. H. VanderLeest, "Arinc 653 hypervisor," in *29th Digital Avionics Systems Conference*, 2010, pp. 5.E.2–1–5.E.2–20.

[24] M. Vanga, A. Gujarati, and B. B. Brandenburg, "Tableau: A high-throughput and predictable vm scheduler for high-density workloads," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190557

[25] S. Schorr and G. Fohler, "Online admission of non-preemptive aperiodic tasks in offline schedules," *Proceedings Work-in-Progress Session*, 2010.

[26] J. Real, S. Sáez, and A. Crespo, "A hierarchical architecture for time- and event-triggered real-time systems," *Journal of Systems Architecture*, vol. 101, p. 101652, 2019.

[27] A. Syed, D. G. Pérez, and G. Fohler, "Job-shifting: An algorithm for online admission of non-preemptive aperiodic tasks in safety critical systems," *Journal of Systems Architecture*, 2018.

[28] I. Molnar, *Modular Scheduler Core and Completely Fair Scheduler*, available: https://lwn.net/Articles/230501/, Last accessed: May 2022.

[29] I. Kadusale, G. Gala, and G. Fohler, "Energy-aware time- and event-triggered kvm nodes," in *Real-time Cloud Workshop co-hosted with 35th Euromicro conference on Real-time systems*, 2023.

[30] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, 2005.

OSPERT and RT-Cloud 2023 Program

	Tuesday, July 11 2023
8:00 – 9:00	Registration
9:00 – 10:00	OSPERT Opening Remarks Session 1: OSPERT 2023 Keynote Co-Developing Hardware and Software *Ulrich Drepper, Red Hat Research*
10:00 – 10:30	Coffee Break
10:30 – 12:00	Session 2: RTOS Reactiveness and Awareness [OSPERT] Assessment of Efficient Dispatching in FreeRTOS *F. Hagens, K.-H. Chen* [OSPERT] ResourceGauge: Enabling Resource-Aware Software Components *A. Schmidt, L. Gerhorst, K. Vogelgesang, T. Hönig* [OSPERT] Arm MUCH: Full-spectrum hardware-event-based Armv8 application profiler *A. Misuraca, A. Bastoni*
12:00 – 13:30	Lunch
13:30 – 14:15	RT-Cloud Opening Remarks Session 3: RT-Cloud 2023 Keynote TBA
14:15 – 15:00	Session 4: From Real-Time OS to Real-Time Cloud Systems [OSPERT] Joint Time-and Event-Triggered Scheduling in the Linux Kernel *G. Gala, I. Kadusale, G. Fohler* [RT-Cloud] Policy Synthesis for Resource Allocation in Clouds *S. Gopalakrishnan*
15:00 – 15:30	Coffee Break
15:30 – 17:00	Session 5: RT-Cloud [RT-Cloud] AORTA: Advanced Offloading for Real-time Applications *A. Balador, J. Eker, R. U. Islam, R. Mini, K. Nilsson, M. Ashjaei, S. Mubeen, H. Hansson, K. E. Arzen* [RT-Cloud] Energy-aware Time- and Event-triggered KVM Nodes *I. Kadusale, G. Gala, and G. Fohler* [RT-Cloud] An RT-cloud solution towards security in Vehicular platooning systems *R. Rafael, H. Kurunathan, E. Tovar*
17:00 – 17:55	Session 6: Panel OSPERT + RT-Cloud Panel
17:55 – 18:00	Closing Remarks
	Wednesday, July 12th – Friday, July 14th 2023
	ECRTS main conference.