PRELIMINARY PROCEEDINGS OF

OSPERT 2025

The 19th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 8th, 2025 in Brussels, Belgium

in conjunction with

The 37th Euromicro Conference on Real-Time Systems
July 8–11, 2025, Brussels, Belgium

Editors:
Kuan-Hsun Chen
Marion Sudvarg

Contents

Message from the Chairs

Welcome to OSPERT'25, the 19th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. This year, OSPERT has experienced a resurgence. 10 technical papers, demos, tutorials, and calls for collaboration have been accepted for presentation during the full-day event. 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.

OSPERT will open with a keynote by Dr. Rich West. He will discuss the challenges the interdependence of safety and security pose in a product's lifecycle, their impact on Bosch and its products, potential solutions, and open research questions After the technical presentations, we will have a second keynote, shared between RTAutoSec and OSPERT, from Dr.-Ing. Zain Hammadeh, discussing safety and security in software development at the German Aerospace Center (DLR). In the afternoon, we will have technical sessions from the RT-Cloud workshop and a keynote given by Dr. George Violettas from SYSGO GmbH, Germany, discussing safety-critical cloud applications. At the end, we conclude with an overarching panel.

OSPERT'25 received 16 high-quality submissions. After a competitive review process, 10 were selected by the program commitee to be presented at the workshop. Each paper received at least three individual reviews. Our special thanks go to the program committee, a team of 10 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.

This year, OSPERT will also recognize an outstanding submission with a Best Paper Award, to be announced during the workshop's closing remarks. Stay tuned; these proceedings will be updated with the announcement!

OSPERT'25 would not have been possible without the support of many people. The first thanks are due to Renato Mancuso, Antonio Paolillo, Joël Goossens, Sebastian Altmeyer, 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,

Kuan-Hsun Chen
Universiteit Twente
The Netherlands

Marion Sudvarg
Washington University in St. Louis
The United States

Program Committee

Angeliki Kritikakou *University of Rennes, IRISA/INRIA*
Arpan Gujarati *University of British Columbia*
Christian Dietrich *Technische Universität Hamburg*
Dakshina Dasari *Robert Bosch GmbH*
Daniel Casini *Scuola Superiore Sant'Anna*
Gedare Bloom *University of Colorado at Colorado Springs*
Junjie Shi *Technische Universität Dortmund*
Marine Sauze-Kadar *CEA-Leti*
Ning Zhang *Washington University in St. Louis*
Peter Wägemann *FAU Erlangen-Nürnberg*

Keynote Talk

Challenges and Experiences Building a Software-Defined Vehicle Management System

Rich West
Professor, Computer Science, Boston University

No longer seen primarily as electromechanical machines, modern automobiles are becoming ”computers on wheels”. At the same time, automotive systems are increasingly embracing software technologies to manage vehicle functionality. As we move away from a multiplicity of electronic control units to manage chassis, body, powertrain, infotainment, connected and autonomous vehicles, we need to develop new management systems.

This talk builds upon earlier work on DriveOS, a management system developed with Drako Motors, for use in their electric vehicles. I describe some of the challenges and experiences building DriveOS, including several key areas of research for future development of such systems.

Rich West is a Professor in the Computer Science Department at Boston University, where he works with his research team on real-time and embedded operating systems. His work addresses issues concerning safety, security, predictability and resource management. He has an MS and PhD in Computer Science from the Georgia Institute of Technology, USA, and an MEng in Microelectronics and Software Engineering from the University of Newcastle-upon-Tyne, UK. He also works with Drako Motors on the development of DriveOS for next generation vehicles. DriveOS consolidates vehicle functions on a centralized computing platform, by integrating real-time and safety critical services with non-critical software using partitioning hypervisor technology.

UltraScale+ SpinalHDL Wrapper: Streamlining Ideas to Bitstream on UltraScale+ platforms.

Denis Hoornaert
Technical University of Munich
Munich, Germany
denis.hoornaert@tum.de

Giulio Corradi
Advanced Micro Devices
Munich, Germany
giulio.corradi@amd.com

Renato Mancuso
Boston University
Boston, Massachusetts, USA
rmancuso@bu.edu

Marco Caccamo
Technical University of Munich
Munich, Germany
marco.caccamo@tum.de

***Abstract*—In an embedded computing landscape that inexorably leans into heterogeneity, System-on-Chips (SoCs) featuring tightly integrated Field Programmable Gate Arrays (FPGA) are bound to proliferate. In particular, such architectures' high degree of flexibility and control caters well to the real-time community. Despite the appeal, real-time research exploiting HW/SW co-design on such architectures has remained tepid. While the usual suspects, such as the complexity of Hardware Description Languages, can be blamed, recent advancements in tooling (*e.g.*, languages, frameworks) have proven efficient in easing the design of FPGA-located accelerators. However, in the context of SoC with FPGA platforms, these solutions fall short of addressing the next hurdle: integrating the custom accelerators with the rest of the SoC, which requires the tedious implementation of various supporting software resources.**

This article presents the first iteration of the UltraScale+ SpinalHDL Wrapper; a SpinalHDL library dedicated to supporting HW/SW co-design on SoC with FPGA platforms. The support ranges from assisting during the design of accelerators to automatically inferring and generating ready-to-use software support, such as Linux Kernel modules and Vivado deployment scripts.

***Index Terms*—FPGA, UltraScale+, Hardware/Software co-design, Hardware Construct Languages**

I. Introduction

Modern System-on-Chip (SoC) for embedded systems are becoming more performant as the functional requirements have prompted a surge in computational demand. Combined with use case-specific Size, Weight, and Power (SWaP) constraints, this has pushed SoC architectures to become increasingly heterogeneous by integrating highly specialized accelerators. Nowadays, platforms featuring on-chip specialized units (*e.g.*, GPUs, TPUs) are ubiquitous. Particularly, SoC architectures featuring tightly integrated Field Programmable Gate Arrays (FPGAs) have attracted attention due to their inherent high degree of programmability and on-the-fly reprogrammability. The overall appeal of this SoC architecture is confirmed by the emergence of many platforms [1]–[4] among which the AMD-Xilinx UltraScale+ model [5] has garnered the most attention. This class of platforms is also referred to as *PS-PL* to denote the combination of a Processing System (PS) (*i.e.*, CPU cores and memory) and a Programmable Logic (PL) (*i.e.*, FPGA). This terminology is used in this article.

Since 2016, the opportunities offered by HW/SW co-design on UltraScale+ platforms have caught the attention of the embedded real-time community. Much research has focused on implementing PL-located accelerators, with application domains ranging from image processing [6] to artificial intelligence [7]. Several research groups have also investigated the impact of inter-accelerator bus activity on the latter's performance [8], [9] and proposed mitigation policies [10], [11]. Frameworks [12], [13] tying together the FPGA's support for dynamic and partial reprogramming with scheduling models have been proposed. Finally, a series of papers explored the idea of using the PL as part of the SoC's memory system, enabling memory requests manipulation [14], [15] and traffic regulation [16], [17]. However, despite the many potentials, these research threads have remained tepid within the embedded real-time community, as illustrated in Fig. 1.

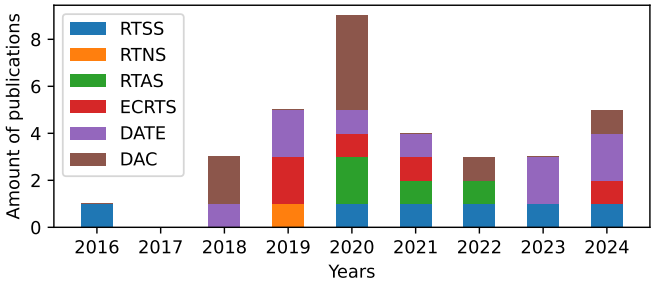


Fig. 1: Published paper since 2016 employing HW/SW co-design on UltraScale+ in real-time oriented conferences.

This trend can be in part associated with the notorious difficulty of designing accelerators for FPGAs compared to programming for other Processing Elements (PEs, *e.g.*, CPUs, GPUs). The commonly agreed-upon culprits are the *traditional* Hardware Description Languages (HDLs, *e.g.*, VHDL, Verilog). Aspects such as their verbosity and confusing specification model are often considered key factors slowing down productivity and learning rate. In that regard, considerable efforts from the research community and the industry have led to the development of advanced approaches and tooling to speed up hardware design time. Notably, High-level Synthesis (HLS) and Hardware Construct Languages (HCL) have addressed these issues by using higher abstraction languages to design and generate digital circuits.

Unfortunately, however, in the context of PS-PL platforms,

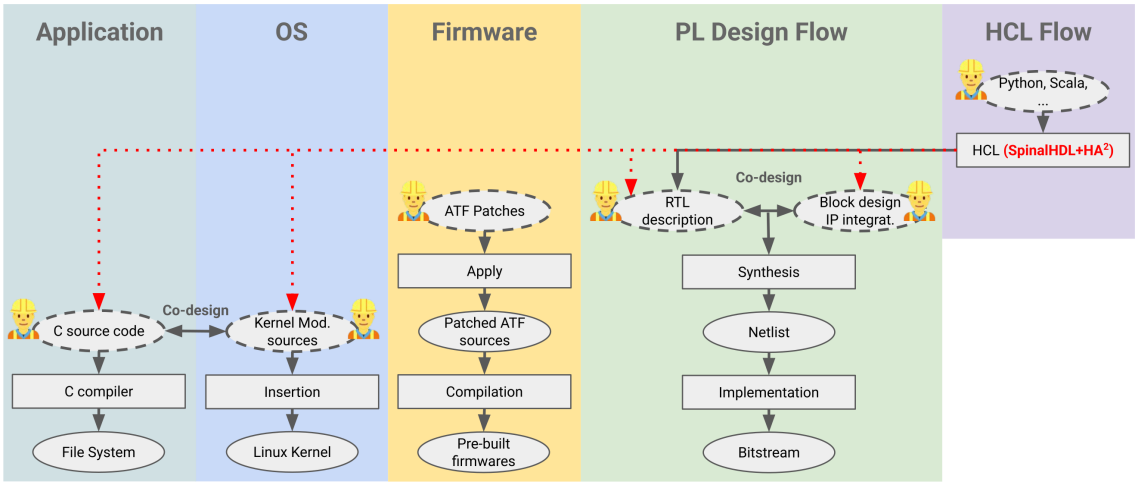


Fig. 2: Overview of the co-design flow for UltraScale+ platforms. Each layer of the system involves manual work requiring specialized expertise (dashed ellipse). The *"PL Design Flow"* can be extended by a HLS or HCL. Our SpinalHDL+USW (dotted red arrows) only requires human intervention for specifics about the hardware, firmware, and final user application.

the aforementioned tools do not address the next productivity hurdle: *hardware/software interaction*. Existing tools [18], [19] focus solely on the accelerator's logic while integration tools [20] consider the PL side in isolation, leaving designers with the tedious—yet delicate—task of integrating the FPGA accelerator with the rest of the system by implementing the necessary software drivers. Nonetheless, proper integration on such a platform is challenging because it may require firmware and Operating System (OS) expertise.

As the authors found out, after several years of experience with the UltraScale+ platform, most of the productivity hurdles often correspond to repetitive implementation tasks. As such, instead of curating their experience as a set of guidelines and documents, the authors propose the UltraScale+ SpinalHDL Wrapper (USW), an open-source [21] library extending a HCL (*i.e.*, SpinalHDL) that aims to generate ready-to-use support to ease HW/SW co-design on AMD-Xilinx UltraScale+ platforms. Such support includes (1) pre-defined specialized hardware constructs for several UltraScale+ boards (*e.g.*, Kria KV260 and Kria KR260), (2) the generation of Linux kernel modules, and (3) the generation of AMD-Xilinx Vivado TCL scripts to enable "one command line" deployment. Through this, USW aims to lower the entry barrier of HW/SW co-designing on UltraScale+ platforms.

II. Ancillary Concepts

A. FPGA and Hardware Description Language

A Field Programmable Gate Array (FPGA) is a type of re-programmable PE capable of emulating *virtually* any specialized digital circuits. FPGAs sit at the cross-road between specialized hardware accelerators and general purpose (*e.g.*, ISA-based) PEs. Like the latter, FPGAs can implement workload-tailored data manipulation and display aggressive parallelism (*e.g.*, via pipelining) while, like general-purpose PEs, they can be dynamically re-programmed on demand.

The design of digital circuits in FPGA is done via HDLs (*e.g.*, Verilog, VHDL), which uses the Register Transfer Level (RTL) abstraction. As the names suggest, with these languages, one *describes* the data flow from registers to registers, wires to wires, and vice versa. An Electronic Design Automation tool (*e.g.*, Vivado) is employed to synthesize the RTL description into a logically equivalent and target-specific *bitstream* that can be flashed onto the FPGA. This process, illustrated in Fig. 2 (green-shaded box), is performed through a sequence of steps referred to as the *PL design flow*. Essentially, once the HDL description is created, it must be connected and co-designed with—potentially vendor-locked—third-party IPs. The output design is *synthesized* into an intermediate representation called a *netlist* on which a series of optimizations are applied. The *implementation* step yields the bitstream.

Due to the tediousness of designing hardware accelerators with traditional HDLs, many projects and industrial products have sought to lift the level of abstraction. Existing solutions rely on CPU programming languages to generate HDLs, sitting atop the usual design flow as shown in Fig. 2. Two notable approaches exist. (1) ***HLS*** [22], [23] aim at transpiling C code into HDL such as Verilog to take advantage of the existing code base and enable a fast *time-to-bitstream* for non-HDL experts. However, the procedural-to-RTL transformation is not straightforward and requires designers to expertly guide the tools via `C` pragmas. (2) ***HCL*** such as Chisel [19] and SpinalHDL [18] take a different route. They posit that the limited code re-utilization and associated language features of HDLs are the main productivity hurdle, not the RTL abstraction. Hence, these HCLs still use the RTL abstraction but embed it within high-level programming languages such as Scala. The latter acts as both a framework and a pre-processing language, offering object-oriented and functional programming features, as well as expressiveness to elaborate and test the designs. Because the final hardware description

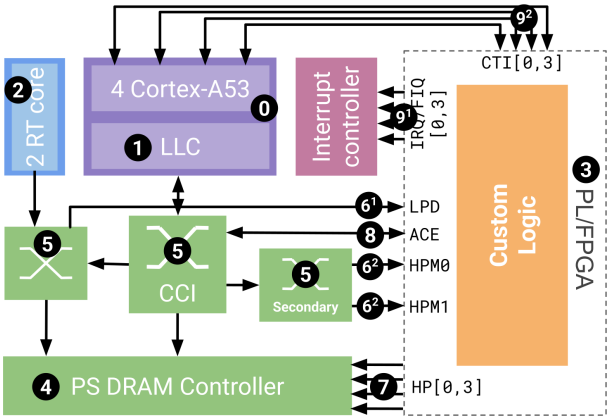


Fig. 3: Overview of a UltraScale+ SoC where only relevant components are depicted. Components are marked by ❶.

uses the RTL abstraction, unlike HLSs, the final logic corresponds 1-to-1 to the desired semantics, avoiding any *"black-magic"* [18] transformation.

B. PS-PL platforms

As depicted in Fig. 3, the UltraScale+ multi-processor SoC manufactured by AMD-Xilinx features four ARM Cortex-A53 CPU cores ❶ equipped with a 1 MB last-level cache ❶, a pair of real-time ARM CPU cores ❷, and an FPGA ❸ (*i.e.*, the PL side). These elements and the DRAM controller ❹ are linked via a system of interconnects ❺ and dedicated signals.

A key aspect of co-designing PS software with PL accelerators is their interactions. On UltraScale+, PS-PL interaction is facilitated by a diverse set of interfaces, including: (1) seven unidirectional AXI4 interfaces: (a) three of whom are memory mapped PS-to-PL interfaces used by the PS to fetch data from the PL ❻ and (b) four PL-to-PS interfaces that allow access to any on-chip memories and the main memory ❼. (2) One two-way cache coherent interface (ACE) that allows the PL to become a member of the SoC cache coherence domain ❽ (*i.e.*, snoop and be snooped by the CPU cores). (3) Several direct PS-to-PL and PL-to-PS interrupt ❾¹ and cross-trigger ❾² lines enable fast communications to/from the PS-side interrupt controller and CoreSight debugging infrastructure.

With an increased heterogeneity comes an increase in design complexity and scope. In fact, not only should designers take into account the PL, they must also prepare the PS and its various software requirements. In this widened co-design flow (shown in Fig. 2), each traditional layer of a system (*i.e.*, firmware, OS, and user/application) must be set up and tailored for communication with the PL-located accelerator(s). On UltraScale+ platforms, it comes as firmware patches (*e.g.*, to enable the PL-side ACE port) and OS kernel modules (*e.g.*, to map the various ports and interrupt lines).

III. Objectives and Overview

Unlike existing FPGA integration tools (*e.g.*, LiteX [20]), the proposed approach does not consider the PL side as an isolated block but instead focuses on its integration with the PS side. Based on the authors' experience, much of the software support required to smoothly interface software with PL-located accelerators can be automatically generated and, especially, inferred from the hardware module description. The proposed approach leans on this observation to ease HW/SW co-design on UltraScale+ platforms.

This article proposes to extend SpinalHDL [18] (an HCL) via a dedicated library called the UltraScale+ SpinalHDL Wrapper (USW). The library assists the hardware designer by providing pre-defined hardware constructs common to all UltraScale+ boards and implicitly generating ready-to-use software resources associated with them. As illustrated in Fig. 2 by the dashed red arrows, the software resources generation ranges from Vivado TCL script to kernel modules. Note that using USW does not lock the designer in, as the generated support can be used as is or as the starting point toward further (manual) customization.

This section presents some of the constructs and tools offered by USW. The code snippet displayed in Listing 1 exemplifies the use of USW; however, details regarding SpinalHDL syntax will not be presented due to space constraints. For further details, readers are invited to look at [18].

A. Top Module, Interfaces, and I/Os

The top module is designed such that it contains all PL's elements. As such, it mirrors all PS-PL interfaces and I/Os. The idea is that the SpinalHDL description is self-contained, and no subsequent manipulation of the outcome is needed.

This implies that most information must come or be derived from the SpinalHDL description. In the proposed library, information such as (1) the target board, (2) the target frequency, and (3) the desired interfaces and I/Os are specified directly in the top-module definition. The top module can be created via a Scala class that inherits a curated class describing the target board (*e.g.*, Kria KV260, Kria KR260, or ZCU102). For instance, the code snippet in Listing 1 implements a top module called `ConfigTestPort` on the Kria KV260 (see line 9). This parent class takes two construction parameters: a frequency and an I/O configuration.

The frequency parameter indicates the desired synthesis (line 10) but does not imply that the Verilog produced can operate at that frequency. The library provides an early indication of whether the target board can support the frequency, sets the closest if not, and considers it when generating the Vivado TCL script (see Sec. III-D). The I/O configuration indicates which interfaces and I/Os should be implemented (lines 11-13). The associated apertures become available in the description block by enabling specific options. In our example, asserting `withLPD_HPM0` tethers the design to the `LPD HPM0` PS-PL interface and allows access to the individual fields in the description via `io.lpd.hpm0` (❻¹ in Fig. 3).

B. Configuration Port

A configuration port is a key component of any accelerator on UltraScale+ as it allows the PS-located software layers to instrument the PL-located accelerators. Implementing an

```
1  package example
2
3  import spinal.core._
4  import spinal.lib._
5  import kv260._
6  import ultrascaleplus.scripts._
7  import ultrascaleplus.configport._
8
9  case class ConfigPortTest() extends KV260(
10   frequency = 100 MHz,
11   config    = new KV260Config(
12     withLPD_HPM0 = true, withIO_PMOD0 = true
13   )
14 ) {
15   val configPort = ConfigPort(io.lpd.hpm0,
     ↪  io.lpd.hpm0.getPartialName())
16
17   val clockCount = Reg(UInt(64 bits)) init(0)
18   configPort.read(clockCount,
     ↪  io.lpd.hpm0.apertures(0).base)
19
20   clockCount := clockCount+1
21
22   for (i <- 0 until io.pmod0.length)
23       io.pmod(i) := clockCount(63-i)
24
25   KernelModule.addIO(io.lpd.hpm0)
26   KernelModule.generate()
27   this.generate()
28 }
```

Listing 1: SpinalHDL code snippet implementing an AXI4 config. port using USW for the Kria KV260 (line 9). Configuration port is tethered to the `LPD HPM0` (line 15) PS-PL interfaces and enables access to a 64-bit wide counter (lines 17-18).

AXI4 compliant configuration port is tedious, but, luckily, SpinalHDL provides an easy-to-use generator [24]. However, the generator lacks a few features to ease the integration with the PS side.

To this end, our USW library provides a specialized version: the `ConfigPort` generator. The latter extends via inheritance the SpinalHDL's generator to provide ready-to-use `C` code generation support for the PS side. More precisely, after being instantiated (line 17), registers can be added to the configuration in read-only, write-only, and read-write modes. For example, in Listing 1 line 18, the `clockCount` register is added for read-only access at address `io.lpd.hpm0.apretures(0).base`. At SpinalHDL elaboration-time, the USW library produces boilerplate `C` code that defines a `C` struct with a field for each register added to the configuration port. Their placement reflects the address specified in the hardware description (Listing 1, line 18). When required, padding is automatically added to the struct. The generated `C` struct provides the typical code to memory-map (`mmap`) the associated PS-PL AXI4 interface and cast the returned pointer into the generated struct.

C. Kernel Module

Part of the challenge when co-designing an accelerator with Linux is to allow access to the PS-PL interfaces. In particular, allocating address ranges falling within the `FPD HPM[0,1]` apertures as cacheable regions directly entails the involved alteration of the Linux kernel or the use of a hypervisor as done in [15], [16]. An alternative is to create an *"insertable"* kernel module that creates `/dev` file system targets. When these `/dev` targets are mapped (via `mmap`), they return a pointer to the base address of the desired aperture.

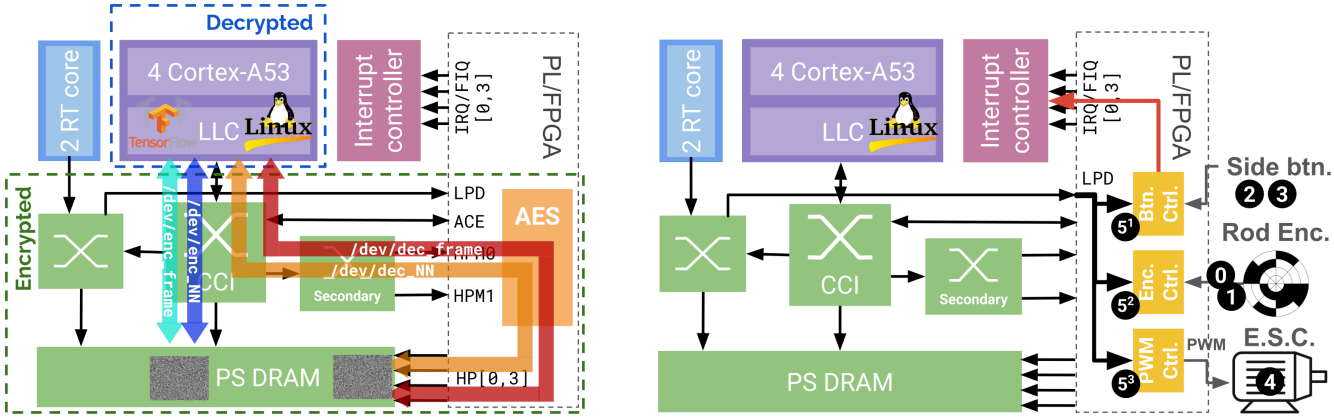
The library's kernel module generator leverages the latter option. As shown in Listing 1 line 28, it is as simple as invoking a singleton named `KernelModule` and calling its `addIO` method on the desired AXI4 port. In this case, the generated kernel module will create, after insertion, a `/dev/lpd_hpm0` file that, when mapped, allows I/O (*i.e.*, uncached) access to the region. Cacheable targets can be generated using the `add` method instead.

D. Vivado TCL Script

The generation of TCL scripts is supported and provided natively by Vivado and is the *de facto* preferred way to share and maintain versioning of Vivado projects. However, unless designers can expertly program in TCL from scratch, the design and all PS-PL I/O connections must be established manually over the GUI. This introduces a human-in-the-loop step that still requires some expertise with the tool, effectively keeping the entry barrier high and preventing fast prototyping. Moreover, any future changes to the design's PS-PL interfacing or in-use I/O entail manipulating and regenerating the TCL script using Vivado. Finally, Vivado projects' portability and deployment are challenging because the TCL scripts are version dependent, and licenses are available only for some versions.

Following the aforementioned philosophy, our USW library can generate a Vivado TCL script to easily share and deploy UltraScale+ designs at the elaboration of a SpinalHDL description (*i.e.*, from SpinalHDL to Verilog). The automated TCL script generation is made possible by the information provided in the description's top module. The produced and ready-to-use TCL scripts are placed in the `vivado/` folder and named after the design they implement—*e.g.*, `ConfigPortTest.tcl` for the design in Listing 1. A clear advantage of inferring the TCL script from the design is the augmented cross-version portability. Concretely, with USW, sharing the design sources is sufficient to allow any collaborator to seamlessly deploy and implement the design regardless of their installed Vivado version. We have tested that USW-generated TCL scripts work correctly for the 2024.2, 2024.1, 2022.2, and 2019.2 versions of Vivado and expect broad compatibility with other versions.

Like other USW-generated support, TCL scripts do not need to be regenerated every time. For instance, if no I/O nor PS-PL interface is altered, removed, or added, the TCL script can serve as an initial springboard toward further manual customization by the designer. Alternatively, the TCL script can be disregarded at the designer's discretion. Finally, note



(a) The AES accelerator receives cacheable transactions on the `FPD HPM0` port. The a-la PLIM module deciphers and ciphers data on reads and writes. The NN and frames remain encrypted in memory.

(b) The PL-located PWM controllers are configured by the CPUs via the `LPD HPM0` port to (1) emit one PWM signal aimed at controlling the pole's car and (2) maintain an estimation of the system's state.

Fig. 4: Abstract overview of the two example use-cases described in Sec. IV implemented on the AMD-Xilinx UltraScale+ platform using SpinalHDL and the USW library.

that any (*i.e.*, small or not) changes applied to a hardware design (*e.g.*, change of condition operator from `&` or `|`) obligatorily implies going down the PL design flow again. This is a lengthy process even for smaller projects, making the redeployment phase of the Vivado project via the USW-generated TCL script a hardly noticeable overhead.

E. SpinalHDL+USW to Bitstream

Once a SpinalHDL description exists, generating the bitstream is a matter of two command lines, as shown below. The resulting bitstream file is placed in the project's folder, and the Vivado project is created in the `vivado/` folder.

```
sbt "runMain example.ConfigPortTestVerilog"
vivado -source vivado/ConfigPortTest.tcl
```

IV. Demonstrators

As mentioned in Section I, HW/SW co-design on PS-PL platforms (and the UltraScale+ platform in particular) have already been successfully deployed in several real-time contexts. Naturally, USW can be used (or has already been used) to replicate the prototypes presented in [10], [15]–[17]. For instance, readers can find SpinalHDL support to generate PLIM-like IPs[1] and a simplified implementation of the PLIM-style *bleacher*[2] from [15] in the example designs of USW.

The authors selected two divergent HW/SW co-designed projects to illustrate the PS-PL interfaces and I/O generation abilities. These examples are individually presented in this section and illustrated in Fig. 4.

[1]https://github.com/denishoornaert/ultrascale-spinal-wrapper/blob/master/hw/spinal/example/Plim.scala

[2]https://github.com/denishoornaert/ultrascale-spinal-wrapper/blob/master/hw/spinal/example/Bleacher.scala

A. Security: On-the-fly AES Encryption

This use case, whose rationale is exhaustively presented in [25], aims to enable on-demand access to confidential data while always keeping the latter encrypted in memory. Typically, the PL-located accelerators encrypt/decrypt confidential data block-by-block upon explicit instrumentation by the PS CPU cores. Instead, as illustrated in Fig. 4a, the design follows the PLIM [15] approach to allow greater flexibility and smoother integration with the PS-located software. This approach creates an alternative PL-traversing route to memory featuring an encryption engine that encrypts and decrypts individual transactions.

Concretely, the encryption engine is implemented as a *counter-mode* AES-128 accelerator [26] from scratch using SpinalHDL+USW for the Kria KV260. The tethering of the three PS-PL interfaces (`LPD HPM0`, `FPD HPM0`, and `FPD HP0`) with the top module is realized via the USW library and the generation of the kernel modules mapping the two primary ports is delegated to the USW library (see Sec. III-C). The `LPD HPM0` port is attached to a configuration port (see Sec. III-B) that stores crucial information (*e.g.*, encryption key) and USW generates a ready-to-use `C` struct for HW/SW interfacing.

On the software side, the confidential (*i.e.*, encrypted) workloads consist of (1) the weights of neural network detecting the presence of humans (implemented and run via TensorFlow Lite) and (2) encrypted video frames coming from a remote device (*e.g.*, over the internet). This data is accessible via four `/dev` targets as illustrated in Fig. 4a (see arrows); two of which are generated by the USW library. More accurately, the orange and red arrows represent the data path created by the `/dev` targets generated by USW. They allow the software direct communication with the PL and access to the decrypted version of neural network (NN) weights (*i.e.*, `/dev/dec_NN`) and the video frame (*i.e.*, `/dev/dec_frame`) respectively.

From a subjective point of view, contributors to the project

concurred that the establishment of the `/dev` targets and `C` struct as a high-level and well-specified interface rendered the HW/SW co-designing more approachable. In particular, it enabled better cross-team communication. This observation underscores the usefulness of streamlining software support.

B. Robotic: Inverted Pendulum

This demonstrator showcases the ability of the library to support HW/SW co-designed robotic systems. In such systems, software-implemented controllers running on the PS interact with custom PL-located peripheral (*i.e.*, I/O) controllers to sense and act on their environment. These controllers can also feature high-frequency stream processing (*i.e.*, accelerated) logic.

We selected a custom inverted pendulum mechanical system as a use case. To achieve the goal of balancing and maintaining the free-flowing rod in an upward position, the cart, actuated by a motor, is controlled by an LQR+Simplex controller. The latter interacts with the PL to set its action (*i.e.*, motor RPM) and access an estimation of the system's state.

For this use case, the Kria KR260 board is used. As illustrated in Fig. 4b, the PL hosts an *inverted pendulum* [27] design consisting of three controllers: two encoders with state estimation logic (❶ and ❶), two buttons indicating each end of the rail (❷ and ❸), and one PWM controller (❹). The encoder controllers are tasked to monitor *all* switching activity on the `PMOD0` associated I/O lines and process them to update state estimation. In our use case, one provides an estimate of the rod position as an angle w.r.t. to the starting point (*i.e.*, pointing downwards), and the second one provides an estimate of the cart's position as a distance w.r.t. the middle of the rails. All controllers can be instrumented and their state accessed by the PS software via a configuration port (see Sec.III-B; ❺) generated and connected to the `LPD HPM0` port by USW. The PS-located controller utilizes the associated `C` struct and a kernel module (see Sec. III-C) generated by our USW framework.

V. Discussion

A. Scalability and Portability

The choice of the UltraScale+ platforms as a focus point is logical considering its widespread availability and omnipresence in the few real-time research using PS-PL platform for HW/SW co-design. However, the USW can be extended to support *virtually any* platform. This is greatly facilitated by the UltraScale+ SoC implementation of standard open-specification protocols (*e.g.*, AXI) as PS-PL interfaces. The combined flexibility and agile approach of USW and SpinalHDL ease the introduction of new interfaces and protocols.

Naturally, expanding USW support to include closely aligned SoCs such as AMD-Xilinx' Zynq and Versal are expected to be easier as they share a common interface naming convention and synthesis backend (*i.e.*, Vivado) with the UltraScale+ platform. The amount of effort required to support other platforms will vary depending on a few factors. For instance, the Enzian [3] platform would benefit from the same backend support as the PL is AMD-Xilinx based. However, some work will be required to add the bus protocol specific to their platform. On the other hand, platforms like [1], [2] rely on different backends, meaning that the TCL script generation may have to be revamped. This could be addressed by raising the abstraction level to provide a common interface and automatic backend selection. Interface-wise, expanding support is easier as the interface protocols used (*e.g.*, AXI, TileLink) are already supported by SpinalHDL.

B. PS-PL Interfaces, Interactions, and Timings

Many related research works cited in this paper already provide timing measurements for accessing PL and PS elements that traverse their shared interfaces. In particular, [28] provides an exhaustive analysis and result set for PL-to-PS communications. It shows that the PS and PL can access the main memory with up to $4.8\ GBps$ of throughput. In [15], the authors report that accessing PL-located memory blocks can be done at $\pm 800\ MBps$. These results provide insight into the level of performance one can expect when employing *"simple"* direct access means like the memory region mapping supported by USW (see Sec. III-C).

However, in the future, USW aims to enable the generation of more sophisticated means of interaction between the PL-located accelerators and the PS-located OS, such as `io_uring` and virtIO. In these cases, the previously reported measurements only represent a performance upper limit as the protocols' control logic must also be considered. The exact final timings are difficult to predict as they are influenced by orthogonal factors such as the communication protocol's control logic, the PL frequency, bus concurrency, etc. Implementing and evaluating such OS-to-accelerator communication is part of the authors' future research.

VI. Conclusion and Extensions

This article presents USW; an open-source SpinalHDL library to simplify HW/SW co-design on AMD-Xilinx UltraScale+ platforms. It does so by conveniently generating ready-to-use software infrastructure, effectively supporting collaborative development on the UltraScale+ platforms.

At the time of writing, the library is in its inception, and further development and refinement will take place. The authors foresee three directions. (1) Expanding the supported interfaces and I/O types (*e.g.*, CoreSight trace port interface, Raspberry Pi camera), boards, and Vivado versions. An important milestone is to provide hardware and software support for designing cache-coherent PL accelerator in the form of (a) re-usable hardware templates and (b) generating ready-to-use Arm Trusted Firmware patches. (2) Addition of support for established AMD-Xilinx technologies such as partitioning of the PL and partial reprogramming. (3) Simplification of PS-PL communication via the generation of software support and hardware modules following established protocols such as `io_uring`, `remote_proc`, and ROS.

VII. Acknowledgments

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. This research was supported by the National Science Foundation (NSF) under grant number CSR-2238476, and by Red Hat Research (#2025-01-RH01). The presented demonstrators are the results of collaborations with many colleagues and students. Heartfelt thanks to Bassel El Mabsout and Patrick Carpanedo for making the *"On-the-fly AES encryption and Decryption"* a reality and to Andres Rodrigo Zapata Rodriguez, Lukas Neef, and Joey Dihardjo for the realization of the *"Inverted pendulum"*. Finally, many thanks to Francesco Ciarolo for the support, brainstorming sessions, and early technical contributions.

References

[1] C. Intel, "Intel's stratix 10 fpga: Supporting the smart and connected revolution," https://newsroom.intel.com/editorials/intels-stratix-10-fpga-supporting-smart-connected-revolution/, October 2016, accessed on 19.01.2022.

[2] M. M. T. Inc., "Polarfire soc - lowest power, multi-core risc-v soc fpga," https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga, July 2020, accessed on 09.01.2020.

[3] G. Alonso, T. Roscoe, D. Cock, M. Ewaida, K. Kara, D. Korolija, D. Sidler, and Z. ke Wang, "Tackling hardware/software co-design from a database perspective," in *Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, Netherlands, Jan. 2020.

[4] T.-J. Chang, A. Li, F. Gao, T. Ta, G. Tziantzioulis, Y. Ou, M. Wang, J. Tu, K. Xu, P. J. Jackson, A. Ning, G. Chirkov, M. Orenes-Vera, S. Agwa, X. Yan, E. Tang, J. Balkind, C. Batten, and D. Wentzlaff, "Cifer: A 12nm, 16mm2, 22-core soc with a 1541 lut6/mm2 1.92 mops/lut, fully synthesizable, cachecoherent, embedded fpga," in *2023 IEEE Custom Integrated Circuits Conference (CICC)*, 2023, pp. 1–2.

[5] Xilinx, "Zynq ultrascale device technical reference manual," https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual, September 2022, accessed: 08.11.2022.

[6] Y. Tan, Y. Zhu, Z. Huang, H. Tan, and W. Chang, "Brief industry paper: Real-time image dehazing for automated vehicles," in *2023 IEEE Real-Time Systems Symposium (RTSS)*, 2023, pp. 478–483.

[7] F. Restuccia and A. Biondi, "Time-predictable acceleration of deep neural networks on fpga soc platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 441–454.

[8] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, vol. 165, Dagstuhl, Germany, 2020, pp. 12:1–12:23.

[9] M. Mattheeuws, B. Forsberg, A. Kurth, and L. Benini, "Analyzing memory interference of fpga accelerators on multicore hosts in heterogeneous reconfigurable socs," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1152–1155.

[10] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "Axi hyperconnect: a predictable, hypervisor-level interconnect for hardware accelerators in fpga soc," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020.

[11] M. Pagani, E. Rossi, A. Biondi, M. Marinoni, G. Lipari, and G. Buttazzo, "A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, vol. 133, Dagstuhl, Germany, 2019, pp. 24:1–24:24.

[12] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable fpgas," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 1–12.

[13] J. Goossens, X. Poczekajlo, A. Paolillo, and P. Rodriguez, "Acceptor: a model and a protocol for real-time multi-mode applications on reconfigurable heterogeneous platforms," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, ser. RTNS '19. Association for Computing Machinery, 2019, p. 209–219.

[14] S. Roozkhosh, D. Hoornaert, and R. Mancuso, "Caesar: Coherence-aided elective and seamless alternative routing via on-chip fpga," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 356–369.

[15] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: Cache bleaching," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 296–309.

[16] D. Hoornaert, S. Roozkhosh, and R. Mancuso, "A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, vol. 196, Dagstuhl, Germany, 2021, pp. 2:1–2:22.

[17] I. Izhbirdeev, D. Hoornaert, W. Chen, A. Zuepke, Y. Hammad, M. Caccamo, and R. Mancuso, "Coherence-aided memory bandwidth regulation," in *IEEE Real-Time Systems Symposium (RTSS)*, 2024.

[18] C. Papon and Y. Xiao, "Spinalhdl," https://github.com/SpinalHDL/SpinalHDL, accessed: 16.04.2025.

[19] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. Association for Computing Machinery, 2012, p. 1216–1225.

[20] F. Kermarrec, S. Bourdeauducq, J.-C. L. Lann, and H. Badier, "Litex: an open-source soc builder and library based on migen python dsl," 2020.

[21] D. Hoornaert. Ultrascale spinalhdl wrapper. [Online]. Available: https://github.com/denishoornaert/ultrascale-spinal-wrapper

[22] AMD, "Ug1399: Vitis high-level synthesis user guide," https://docs.amd.com/viewer/book-attachment/o4VR_92ERnsC86VNmOmjrw/dUOl6pD9nb5aE0UIQbrzNA-o4VR_92ERnsC86VNmOmjrw, July 2023, accessed: 16.04.2025.

[23] L. Josipovic, A. Guerrieri, and P. Ienne, "Synthesizing general-purpose code into dynamically scheduled circuits," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 97–118, 2021.

[24] C. Papon. Axi4slavefactory. [Online]. Available: https://github.com/SpinalHDL/SpinalHDL/blob/dev/lib/src/main/scala/spinal/lib/bus/amba4/axi/Axi4SlaveFactory.scala

[25] S. Roozkhosh, B. El Mabsout, C. Rodrigues, P. Carpanedo, D. Hoornaert, S. Min Tan, B. Lubin, M. Caccamo, S. Pinto, and R. Mancuso, "Burning fetch execution: A framework for zero-trust multi-party confidential computing," in *GenZero 2024*, to appear.

[26] D. Hoornaert. Aes in spinalhdl. [Online]. Available: https://github.com/denishoornaert/AES

[27] L. Neef and D. Hoornaert. Inverted pendulum. [Online]. Available: https://github.com/denishoornaert/InvertedPendulum

[28] S. W. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, and W.-m. Hwu, "Analysis and optimization of i/o cache coherency strategies for soc-fpga device," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 301–306.

Tintin: PMU Scheduling to Minimize Uncertainty

Marion Sudvarg*†, Ao Li*, Zihan Li*, Sanjoy Baruah*, Chris Gill*, Ning Zhang*
*Department of Computer Science and Engineering, †Department of Physics
Washington University in St. Louis
(msudvarg, ao, tomson.li, baruah, cdgill, zhang.ning)@wustl.edu

***Abstract*—Hardware performance counters (HPCs) on CPUs are essential for tracking program and system behavior by recording microarchitectural events. A major challenge is that, even on modern platforms, the number of available HPCs is limited compared to the events of interest. Existing software tools mitigate this by multiplexing events in a round-robin fashion; however, this approach may introduce significant errors. In a recently-published paper at OSDI, we introduced Tintin, a new HPC profiling infrastructure that mitigates multiplexing errors by characterizing event count uncertainty at runtime and scheduling events on counters to minimize it. This paper provides details on Tintin's elastic model for HPC scheduling that were omitted from the original paper, including a new method for interpolating multiplexed event counts, evaluation of Welford's method for online variance updates, an elastic scheduling policy to minimize predictive uncertainty, and handling of overlapping profiling scopes (e.g., core-level and process-level event monitoring). We also provide a roadmap for using Tintin with the RT-Bench framework for real-time systems benchmarking.**

***Index Terms*—hardware performance counters, measurement uncertainty, elastic scheduling**

I. Introduction

Hardware performance counters (HPCs), which are widely available on modern server, desktop, and embedded CPUs, provide essential tracking of program and system behavior. HPC data are valuable across a wide range of application domains, including debugging [1]–[4], workload optimization [5]–[7], power analysis [8]–[10], diagnostics [7], online resource provisioning [11]–[15], and intrusion detection [16], [17]. Often, HPC data are used to make predictions about program performance to inform decisions. For each use case, it is important to model how target metrics or behaviors of interest depend on hardware event counts.

Brokering HPC Access. HPCs are typically made available to system software via a per-core performance monitoring unit (PMU). The PMU provides an interface to program each HPC to monitor a particular microarchitectural event (e.g., cache loads or misses). Effective PMU usage presents a major challenge even on modern platforms: the number of available HPCs is limited (often 2–6 per PMU) versus how many types of events they can monitor (dozens to several hundred). Towards managing this tension, event profiling tools such as Linux Perf [18] broker access to HPCs. In much the same way that threads and address spaces virtualize the limited physical CPU and memory resources on a system, these tools provide an abstraction layer over the PMU and its limited HPCs.

However, problems arise when using existing event profiling tools to broker HPC access. First, those that mitigate the challenge of limited HPC availability do so via *event multiplexing*, monitoring events in a time-shared round-robin manner, where each event typically receives an equal portion of time. This implies that events will remain unmonitored for some time; observed counts are typically interpolated over these intervals, which may introduce significant errors. Second, event profiling may be requested from multiple *overlapping scopes*. For example, a running process could profile its own hardware events while event counts are simultaneously collected for the processor core it's on. Profiling tools typically treat these scopes independently, which may exacerbate the effects of multiplexing, especially when they share events in common. Moreover, in some cases, scope conflicts can give rise to starvation scenarios where an event is never monitored.

Tintin. In a recent OSDI paper [19], we presented Tintin, a new hardware event profiling infrastructure that addresses the aforementioned challenges while providing flexible specification of event profiling scopes with fine granularity. Of primary relevance to this paper, Tintin characterizes multiplexing errors as uncertainty, which it dynamically tracks during runtime. We demonstrated that the problem of scheduling events on HPCs to minimize overall uncertainty reduces to elastic scheduling, then implemented our algorithm from [20] to allocate HPC time in the Linux kernel. Moreover, since multiplexing-based errors cannot be fully eliminated, Tintin also reports uncertainty via user-space interfaces; applications may use these to enhance their predictive models or rule-based decision logic.

Contributions. A few details were omitted from [19] due to space constraints. First, Tintin implements a custom method for interpolation over multiplexed event counts to address a problem with the original Trapzeoid Area Method proposed in [21]. Tintin's approach is derived and shown to be correct in §III-A. Second, Tintin uses Welford's method [22] to track observed variance in event rates without storing past observations, which improves efficiency in both time and memory. In §III-B, we provide additional implementation details of Tintin's weighted version of Welford's method; this is evaluated in the Linux kernel against a traditional variance computation in §VI-B. Third, although we've already presented Tintin's elastic hardware event scheduler in [19], supplementary details relating input uncertainty to predictive uncertainty are added in §IV-A. Finally, Tintin handles joint scheduling of events from multiple profiling scopes; a previously omitted derivation of its policy is found in §IV-B.

Of particular relevance to the real-time operating systems

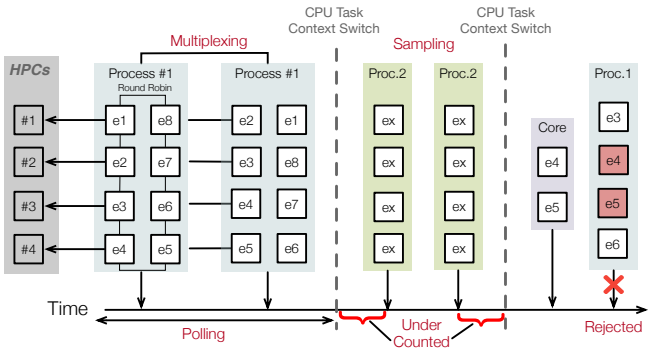


Fig. 1. The Linux *perf_event* subsystem multiplexes events on limited HPCs. It is not aware of scope overlap; joint scheduling and common attribution remain unsupported. If a per-core event is pinned to an HPC, per-task events may be rejected, leading to measurement starvation.

community in general, and this year's OSPERT workshop in particular, we provide a roadmap for using Tintin with the RT-Bench [23] open-source framework which integrates existing benchmarks into a recurrent real-time task model. A presentation of its latest updates also appears at this year's OSPERT [24]. RT-Bench uses Linux *perf_event* to sample hardware events during benchmark execution. in §V, we illustrate the minor code changes necessary to allow RT-Bench to use Tintin's elastic scheduling and uncertainty reporting to enable monitoring of more events than available HPCs with minimal losses in accuracy.

For completeness, §VI-A summarizes experimental results from [19] comparing Tintin to the Linux *perf_event* subsystem.

II. Background

A. Hardware Performance Monitoring

Hardware Performance Counters. Modern processors make hardware event profiling available to system software via a per-core performance monitoring unit (PMU), which provides a set of programmable hardware performance counters (HPCs) that can be configured individually to measure a specific type of microarchitectural event, e.g., cache or bus accesses, cache writebacks or refills, branch misprediction, etc. When enabled, an HPC increments whenever its programmed event occurs. The number of measurable events is substantial, typically exceeding several dozen on ARM processors (e.g., 58 on the Cortex-A53 [25] and 151 on the Cortex-A78 [26]) and over one thousand on Intel processors (e.g., 1,623 on HaswellX [27]). However, the number of available HPCs is very limited; most processors provide only 2-6 per physical core [27], [28], and this number is effectively halved per logical core when enabling Simultaneous Multithreading (SMT).

The Linux *perf_event* Subsystem. This is Linux's kernel-level abstraction layer for interacting with HPCs. It serves as the de facto infrastructure widely utilized by many tools such as PAPI [29], Intel EMON [30], VTune [31], pmu-tools [32], and the Linux Perf utility [28]. It provides access to hardware-level HPC data and software-level data (e.g., memory footprint and tracepoints). The former is the exclusive focus of Tintin.

Scheduling Events on Limited HPCs. Existing work has sought to select relevant events carefully for a given application. However, the number of events of interest often remains larger than the available HPCs. For example, in [33], the 120 events available on an ARMv7-A CPU were narrowed to only 34 of importance for predictive DVFS. CounterMiner [34], an offline analysis tool for predictive modeling with event counts, achieves the most accurate IPC predictions for HiBench [35] workloads using $\sim$150 events. Furthermore, some applications profile derived metrics, such as *Memory_Bound* [5]. These combine as many as 16 individual events [36]. Pond [14], which we evaluated as a case study for Tintin in [19], provides a memory pooling model for cloud infrastructure. Its latency-prediction model takes 7 derived metrics as inputs, spanning 20 events on Intel Skylake.

The current approach to managing this limitation is through event *multiplexing*. When an HPC is configured to count a particular event, we say that the event is *scheduled* on the counter. If the number of events to be monitored exceeds the number of available HPCs, they must time-share the counters; *perf_event* schedules events for monitoring in round-robin fashion [21], [27], [37], [38]. Fig. 1 presents an illustrative example in which there are 4 available HPCs, while Process #1 requires monitoring for 8 events. In this scenario, the events {e1, e2, e3, e4} are scheduled in the initial time slice, followed by events {e2, e3, e4, e5}. HPC measurements for each event can then be interpolated to estimate the total count, but this unavoidably introduces errors. For example, we found that the standard deviation for hardware event counts generated by the *541.leela_r* Go engine in the SPEC CPU®2017 benchmark suite [39] as reported by Linux Perf increased by over 6$\times$ when profiling 8 events compared to 4 [19].

Profiling Scope. The *perf_event* subsystem enables specification of hardware event monitoring for either individual tasks (processes/threads) or CPU cores. Upon CPU task scheduling, it first schedules events bound to the current core before adding those associated with the active process. Since events are bound to per-core and per-task data structures, they are managed independently, even if they share common events of interest. The right side of Fig. 1 illustrates this scenario: a user assigns two events, {e4, e5}, to the current core; these are placed on HPCs #2 and #3. Consequently, events {e4, e5} monitored for Process #1 are not scheduled, even though they represent the same event types, resulting in starvation.

B. Tintin

Tintin, a hardware event monitoring infrastructure introduced in our recently-published paper at OSDI [19], aims to solve these limitations via the three components in Fig. 2.

Quantification of Multiplexing Errors. Multiplexing errors can be quantified at runtime based on *variance* in observed event rates. The Tintin-Monitor component tracks variance, then reports *expected error* back to user-space applications alongside the measured counts, helping to better inform profiling-based decision-making. Details on Tintin-Monitor's

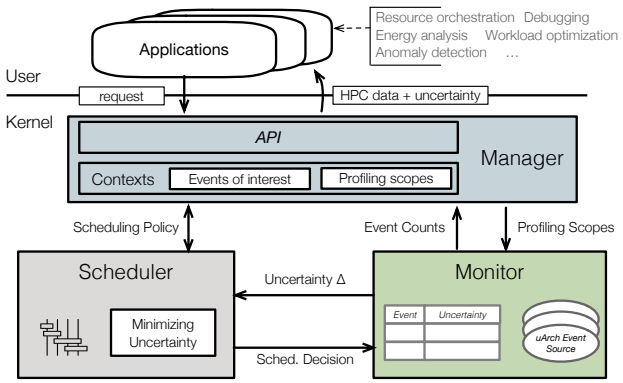


Fig. 2. Tintin design overview.

rate-based count interpolation and variance tracking which were omitted from [19] are provided in §III.

Scheduling to Minimize Uncertainty. By allocating more HPC time to events with larger variance, overall error is minimized. The Tintin-Scheduler component uses the errors reported by Tintin-Monitor to schedule events on HPCs, assigning each event a unique share of time with the objective of minimizing overall error. This problem is shown to be semantically equivalent to elastic scheduling. §IV-A provides deeper insights into this connection than were given in [19].

Indirection to Handle Profiling Scopes. An additional level of indirection can provide a uniform mechanism to handle the heterogeneity of profiling requirements. Tintin provides abstractions to handle issues related to profiling scope, including a uniform API that enables greater flexibility and granularity in specification via the Tintin-Manager component. It elevates a profiling scope to a first-class object, an Event Profiling Context (*ePX*). Tintin-Manager manages *ePX*s collectively for all applications, enabling overlapping events from concurrently-active scopes to be scheduled jointly. Details on how joint scheduling is handled within the elastic model, which were only mentioned cursorily in [19], are provided in §IV-B.

III. Count and Uncertainty Estimation

Formally, an event $e_i \sim (x_i, \sigma_i)$ has an estimated count x_i and uncertainty σ_i due to interpolation over multiplexed observations. In this section, we describe the design rationale behind Tintin's interpolation and uncertainty tracking mechanisms.

A. Interpolation with the Trapezoid Area Method

To derive the estimated total count x_i from a measured count x_i', many tools (e.g., Linux Perf) use count-based interpolation: $x_i = x_i'/U_i$, where U_i is the fraction of time that event type e_i was scheduled on a counter. Several alternative methodologies are explored in [21], [40], [41]. Besides those that avoid multiplexing via multiple program runs and offline analysis, the trapezoid area method (TAM) —which uses rate-based interpolation— is suggested to be the most accurate [21].

Formally, an event e_i is measured with count x_i^j over some continuous time interval $I_i^j = [a_i^j, b_i^j]$ of duration δ_i^j. Then its average rate r_i^j over this interval is x_i^j/δ_i^j. Over consecutive intervals, TAM's original implementation [40] assumes that

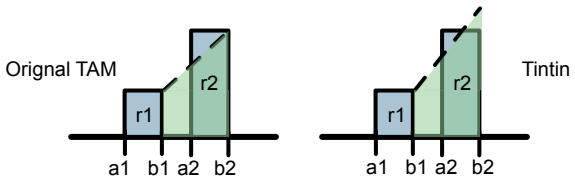


Fig. 3. Tintin's TAM implementation.

the arrival rate follows the straight line connecting the points (b_i^j, r_i^j),(b_i^{j+1}, r_i^{j+1}), and interpolates the count using the area of the resulting trapezoid, as illustrated in Fig. 3. However, this violates the following proposed invariant:

If an event e_i is always monitored, the produced event count estimate x_i' should equal the observed count x_i.

Tintin-Monitor instead constructs the trapezoid so that its top passes through the midpoint $((b_i^{j+1} + a_i^{j+1})/2, r_i^{j+1})$ of the second measured interval. This is derived as follows.

For event e_i, we assume a constant change in the rate of event arrival from interval I_1 to I_2. This is the line $r(t) = mt + c$ connecting the center of each interval at points:

$$\left(\frac{a_1 + b_1}{2}, r_1\right) \text{ and } \left(\frac{a_2 + b_2}{2}, r_2\right)$$

Solving for the slope:

$$m = \frac{r_2 - r_1}{\frac{a_2+b_2}{2} - \frac{a_1+b_1}{2}} = \frac{2(r_2 - r_1)}{a_2 + b_2 - a_1 - b_1}$$

Then solving for the intercept c:

$$r_1 = \frac{(r_2 - r_1)(a_1 + b_1)}{a_2 + b_2 - a_1 - b_1} + c$$

The formula can then be expressed as:

$$r(t) = \frac{2(r_2 - r_1)}{a_2 + b_2 - a_1 - b_1}t + r_1 - \frac{(r_2 - r_1)(a_1 + b_1)}{a_2 + b_2 - a_1 - b_1}$$

After measuring event e_i during interval I_2, we interpolate over the unmonitored time to estimate the total event arrival count Δx_i since the end of I_1 by integrating from b_1 to b_2:

$$\Delta x_i = 0.5 \cdot (r(b_1) + r(b_2)) \cdot (b_2 - b_1)$$

This equals:

$$\begin{aligned}\Delta x_i &= \left(\frac{(r_2 - r_1)(b_1 + b_2)}{a_2 + b_2 - a_1 - b_1} + r_1 - \frac{(r_2 - r_1)(a_1 + b_1)}{a_2 + b_2 - a_1 - b_1}\right) \cdot (b_2 - b_1) \\ &= \frac{(b_1 - b_2)(r_1(a_2 - b_1) + r_2(b_2 - a_1))}{a_1 - a_2 + b_1 - b_2}\end{aligned}$$

Invariance can be proven because, if $b_1 = a_2$ then we have:

$$\Delta x_i = \frac{(a_2 - b_2)(r_1(a_2 - a_2) + r_2(b_2 - a_1))}{a_1 - a_2 + a_2 - b_2}$$

$$\Delta x_i = \frac{(a_2 - b_2)(r_2)(b_2 - a_1)}{a_1 - b_2} = r_2(b_2 - a_2)$$

Which is precisely the count in interval I_2.

B. Tracking Variance with Welford's Method

As described in [19], for an event e_i with interpolated count x_i, we define uncertainty as the expected error σ_i in the count. The instantaneous rate of event arrival is a random variable $\mathbf{r}_i$, with a mean rate r_i^j obtained during each measurement interval I_i^j. Since it is infeasible to measure instantaneous rates, Tintin-Monitor instead characterizes expected variance $E(V(\mathbf{r}_i))$ by taking the variance of the sample means, weighted by duration. In many stochastic processes, variance scales linearly with time [42]; since we are measuring variance in *rate*, it therefore follows that the expected variance $V(x_i, t)$ in event *count* x_i over the time t that the event is not monitored can be expressed as $V(x_i, t) = V(\mathbf{r}_i) \cdot t^2$. Then the expected error σ_i in the estimated event count is the standard deviation $\sqrt{V(x_i, t)}$.

As noted in [19], to update the observed variance in sample means efficiently during online profiling, Tintin-Monitor uses a weighted version of Welford's method [22]. Unlike standard variance calculations, this only requires one pass through the data, obviating the need for Tintin to store all sampled data. At the end of monitoring intervals I_i^j, variance is updated as

$$V_i = V_{i-1} \times (r - \mu_i) \times (r - \mu_{i-1}) \tag{1}$$

where μ is the time-weighted mean over collected counts. To avoid involving floating-point operations in the involved division steps, we apply a scaling factor, where necessary, to the numerator then perform integer division to achieve a fixed-precision result. To prevent overflow, we use 64-bit integers and hand-tune the order of operations to avoid large values.

Because Welford's method is a one-pass technique, as new measurements are obtained, variance is updated in constant time. §VI-B shows an empirical comparison of the kernel overhead imposed by Welford's method, versus a traditional two-pass variance calculation.

IV. Elastic Scheduling to Minimize Uncertainty

A. Hardware Event Scheduling

Tintin-Scheduler aims to adjust allocations of time for events on limited HPCs to minimize overall uncertainty.

Scheduling Model. Formally, for a profiling scope, we define $\mathbf{E} = \{e_i\}$, the set of events to be monitored, where $|\mathbf{E}| = n$. Similarly, $\mathbf{C} = \{c_j\}$ denotes the set of HPCs available to monitor them, where $|\mathbf{C}| = m$. The problem is to determine a schedule $\mathcal{S}(t) : \mathbf{C} \to \{\mathbf{E}, \phi\}$ that defines, at each time t, the event assigned to each counter. We define an event's *utilization* U_i as the fraction of time it occupies a counter.

Scheduling Policy. Many applications use HPC data as inputs to statistical or learning-based models that predict program performance to inform decisions. For example, Pond [14] is a Microsoft Azure resource orchestration system. In cloud environments, memory pooling may improve DRAM utilization and reduce cost, but using pooled instead of local memory substantially increases the latency of some virtual machine workloads. To allocate limited local memory appropriately, Pond uses hardware event data to predict whether a VM workload's latency will be sensitive to pooled memory use.

Uncertainty in input values induces uncertainty in output predictions. Tintin-Scheduler is designed primarily to schedule events on HPCs so as to minimize output uncertainty in prediction, although it works in general to minimize overall event count uncertainty when no single predicted metric is targeted. Tintin-Miner, an offline analysis tool for which development is currently a work in progress, uses model-agnostic variance-based sensitivity analysis to gauge the impact of each input event on output fidelity. Specifically, Tintin-Miner employs Sobol's indices [43], which measure the contribution of each input variable to the output variance. By changing the counts of an individual event e_i and then measuring the variance of output, it calculates the event's first-order sensitivity index S_i^1, which represents the percentage of model output variance contributed individually by that event as an input to the model (i.e., the effect of changing the count x_i only).

To minimize predictive uncertainty, we therefore want to minimize the total variance, weighted by sensitivity, of the inputs. From §III-B, the expected error σ_i in the estimated count of event e_i is $\sqrt{V(\mathbf{r}_i)} \cdot t$, where t is the amount of unmonitored time. Future unmonitored time for event e_i is thus proportional to $1 - U_i$, where U_i is its utilization; thus, we express the expected error as $\sigma_i = \left(\sqrt{V(\mathbf{r}_k)}\right) \cdot (1 - U_i)$. Variance is thus $\sigma_i^2 = V(\mathbf{r}_i) \cdot (1 - U_i)^2$. The scheduling problem is therefore stated as the following constrained optimization:

$$\min_{U_i} \quad \sum_{i=1}^{n} w_i V(\mathbf{r}_i) \cdot (1 - U_i)^2 \tag{2a}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} U_i \leq m \quad \text{and} \quad \forall_i, \quad U^{\min} \leq U_i \leq 1 \tag{2b}$$

Where $U^{\min}$ is a lower-bound on the scheduling quantum (minimum schedulable utilization time slice) to avoid event starvation or floods of timer interrupts. For generality, we use w_i to denote a weight assigned to event e_i; from our sensitivity model described above, this may be set equal to Sobol's first-order sensitivity index S_i^1. In general, if lacking sensitivity analysis, Tintin assigns $w_i = \frac{w_i^*}{x_i^2}$, where w_i^* is a user-specified weight (set via Tintin's syscall API) and x_i normalizes the standard deviation by the estimated event count.

Eqn. 2 is equivalent to the formulation in [44] of elastic scheduling as a constrained optimization problem. The elastic real-time model, proposed by Buttazzo et al. [45], [46], adapts task utilizations to avoid overload on limited processor resources. Here, we solve the similar problem of adapting the scheduling utilizations of hardware events on limited HPCs. Tintin-Scheduler uses our quasilinear-time elastic scheduling algorithm from [20], [47] to assign utilizations. In §VI-A, we show that Tintin's dynamic variance-weighted elastic scheduling policy improves the accuracy of its event count estimates, even in the absence of sensitivity data.

B. Scheduling Multiple Scopes

As mentioned in §II-B, Tintin collectively schedules all events from any active *EPX*s. The details were omitted from our accepted OSDI paper [19]; we outline the approach here.

Scheduling Model. The problem remains to determine a schedule $\mathcal{S}(t) : \mathbf{C} \rightarrow \{\mathbf{E}, \phi\}$ that defines, at each time t, the event assigned to each counter. The complication now is that multiple active *EPX*s may share events in common. For example, the system might be monitoring events $\{e1, e2, e3, e4\}$ on core 0, while a task running on that core monitors events $\{e1, e2, e5, e6\}$. It does not make sense to treat $e1$ and $e2$ separately for each profiling scope for purposes of attribution: if counts are read for either event, they should be attributed to both *EPX*s. Otherwise, unmonitored time becomes unnecessarily inflated, contributing additional uncertainty. However, each *EPX* independently tracks counts and variance for its events, which raises questions about how these should be used as inputs to the scheduling problem.

We denote event $e^*_{i,j} \sim (x_{i,j}, \sigma_{i,j}, w_{i,j}, e_{i,j})$ as the i^{th} event associated with the j^{th} *EPX*. As usual, x and σ are the estimated counts and error; w denotes its weight (see §IV-A) and e denotes the identifier of the hardware event, implying that $e_{a,j}$, $e_{b,j}$ should be unique within a single *EPX*, but $e_{a,j}$ and $e_{b,k}$ might be the same, indicating common events among *EPX*s. Each *EPX* is also assigned a weight z_j. This defaults to 1, but is user-settable via Tintin-Manager's syscall API.

Scheduling Policy. Our objective now is to minimize total weighted uncertainty among the predictive models underpinning each *EPX*. By extension of Eqn. 2, this can be stated as:

$$\min_{U_k} \quad \sum_{i=1}^{n}\sum_{j=1}^{m} z_{i,j} w_{i,j} \sigma^2_{i,j} \cdot (1 - U_k)^2 \tag{3a}$$

$$\text{s.t.} \quad \sum_{U_k} \leq m \quad \text{and} \quad \forall_k, \quad U^{\min} \leq U_k \leq 1 \tag{3b}$$

where we assume there are m active *EPX*s, and we denote the utilization of event $e_{i,j}$ as U_k to reflect that there may be events in common. We may instead denote the collection of monitored *events* $\{e_k\}$, where each event is assigned a set of values $\{z_{k,j}\}$, $\{w_{k,j}\}$, $\{\sigma_{k,j}\}$ according to the *EPX*s that track it. If an event is not tracked by some *EPX*, these values may be set to 0. We can then re-state the optimization problem:

$$\min_{U_k} \quad \sum_{e_k} \left(\sum_{j=1}^{m} z_{k,j} w_{k,j} \sigma^2_{k,j} \right) \cdot (1 - U_k)^2 \tag{4a}$$

$$\text{s.t.} \quad \sum_{U_k} \leq m \quad \text{and} \quad \forall_k, \quad U^{\min} \leq U_k \leq 1 \tag{4b}$$

Notice that this is exactly the quadratic optimization problem in Eqn. 2, but with coefficients $\left(\sum_{j=1}^{m} z_{k,j} w_{k,j} \sigma^2_{k,j}\right)$ for each event e_k. It therefore can be solved using the same algorithm with transformed inputs.

V. RT-Bench Integration

RT-Bench [23] is an open-source tool to address the benchmarking needs of real-time systems. Rather than providing its own computational workloads, RT-Bench serves as a framework in which existing benchmarks, such as Isolbench [48] or SD-VBS [49], can be integrated. RT-Bench provides a wrapper to run benchmark workloads as periodic tasks using a specified period and deadline under Linux's real-time schedulers. It supports processor pinning and constraints on memory allocation.

```
-   int fd = syscall(__NR_perf_event_open, &attr, …);
+   int fd = syscall(__NR_tintin_event_open, &attr, …);
```

```
-     read(fd, &measurement, sizeof(struct read_format));
+     read(fd, &measurement, sizeof(struct tintin_read_format));
      long unsigned value = measurement.value;
+     long unsigned uncertainty = measurement.uncertainty;
```

Fig. 4. Simplified code snippets for using Tintin instead of Linux *perf_event* in RT-Bench. Both changes are made to files in `generator/src`. **(Top)**: Set up monitoring in `performance_counters.c`. **(Bottom)**: Retrieve count and uncertainty measurements in `performance_sampler.c`.

RT-Bench reports performance statistics from the running benchmarks, including several hardware event counts. When initializing a benchmark workload, it establishes monitoring via Linux *perf_event*. It then launches a dedicated performance monitoring thread, PMThread, to perform high-frequency HPC sampling via returned file descriptors. In its original release, RT-Bench only monitored L2 cache refills [23], but it has since been updated to monitor L1 and L2 references and refills, retired instructions, and CPU cycles. These, and other updates, are also presented at this year's OSPERT [24].

In its current version, RT-Bench does not monitor more events than the number of HPCs, and it assigns all events to the same group, guaranteeing simultaneous monitoring. However, as stated in [23], RT-Bench may benefit from adding more performance counters. Indeed, access patterns to other shared hardware resources, such as the L3 cache, memory bus [50], and TLB [51], may cause significant contention and delays in real-time task execution [52]–[55].

With its ability to significantly reduce measurement errors due to HPC multiplexing, Tintin is a suitable candidate for use in RT-Bench instead of *perf_event*. Moreover, Tintin can report both event counts *and* variance-based uncertainty in those counts. If greater confidence is desired, an RT-Bench user can make an informed decision to rerun a benchmark, either assigning greater weight to selected events for elastic scheduling (the w_i values in Eqn. 2a), or running the benchmark multiple times, each time profiling different subsets of the events of interest. Integrating Tintin into RT-Bench is straightforward, requiring the minimal code changes shown in Fig. 4.

VI. Evaluation

A. Overview of Prior Results

Here, we summarize a few of the key results from our OSDI paper on Tintin's performance [19].

Accuracy and Overhead. We used the SPEC CPU®2017 [39] and PARSEC 3.0 [56] benchmark suites to assess the accuracy of the event counts collected online by Tintin, as well as its runtime overhead, in comparison to Linux *perf_event*. We select the 24 default predefined events in Linux *perf_event* to profile simultaneously [57]. To measure event count accuracy, we obtain ground truth by pinning one event to an HPC in each run. We repeat the experiments for the first 5 predefined

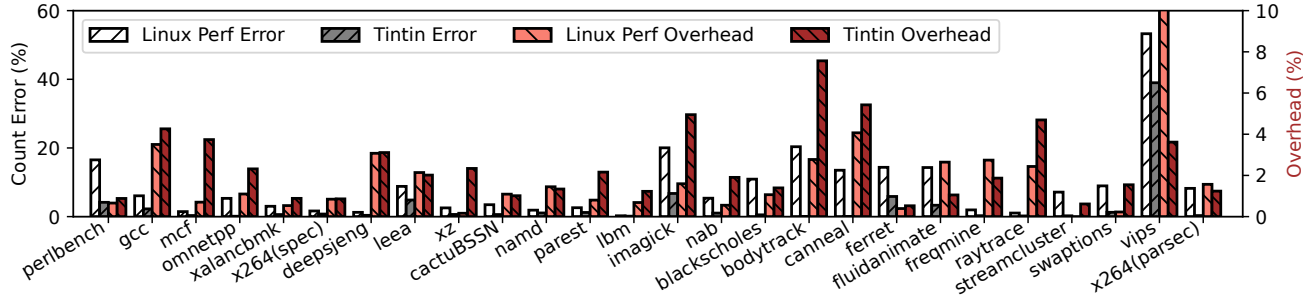


Fig. 5. Benchmark accuracy and overhead results from [19].

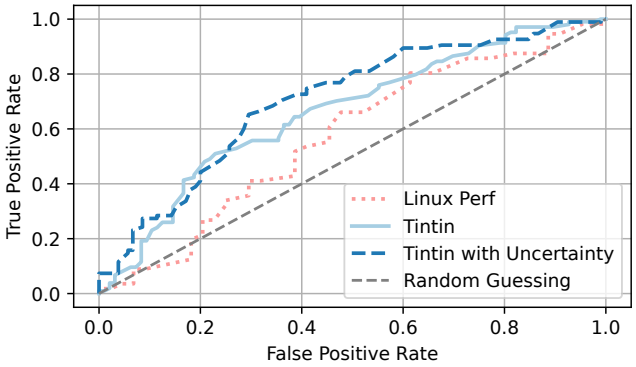


Fig. 6. HPC-based rootkit detection accuracy.

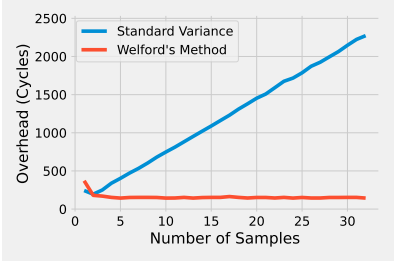


Fig. 7. Comparison of overheads induced by variance computation.

events in `PERF_TYPE_HARDWARE` in [57]. Runtime overheads are obtained by measuring mean benchmark execution times across 10 runs; these are normalized to the execution times without profiling. Results are shown in Fig. 5.

Event counts obtained with Linux *perf_event* were, on average, 9.01% off from the ground truth, with a maximum error of 53.27%. In comparison, Tintin's errors remained well under 5% in most cases, with an average of 2.91%. Tintin exhibits an average overhead of only 2.4%, only slightly higher than *perf_event*'s 1.9%. In the worst-case, we observed Tintin's overhead reached up to 7.6% while *perf_event*'s reached up to 12.7%. Tintin's achieves better execution time performance in scenarios where elastic scheduling allows fewer interrupts.

Impact of Quantifying Uncertainty. Many applications use HPC data to make predictions, e.g., to identify malicious behavior in intrusion detection systems. To evaluate Tintin's ability to enhance such systems, we adopted the experimental setup from [17] on detecting Linux rootkits. As malware, we use the open-source rootkit Diamorphine [58]. We train a random forest classifier (as in [17], [59]) on the events defined under `PERF_TYPE_HARDWARE`. We compare its accuracy when collecting event counts using *(i) perf_event*; *(ii)* Tintin with just elastic scheduling, weighting each event equally; and *(iii)* additionally using the measurement uncertainty reported by Tintin as additional inputs to the classifier. Fig. 6 presents the resulting ROC curves. The area under the curve (AUC) for *perf_event* is 0.57. Tintin improves the AUC to 0.66 with elastic scheduling, and to 0.70 with reported uncertainty.

B. Benefits of Welford's Method

To highlight the benefit of using Welford's method over a traditional two-pass variance calculation, we profile the in-kernel execution time of updating the running variance for an event. For direct comparison, we implement a Linux kernel module that simulates HPC reads and run it on a Raspberry Pi 3 Model B+. Counts and monitoring interval durations are written into an array for use by traditional weighted variance, and we use Tintin-Monitor's *update_variance* function to update the running variance attached to an *ePX* with Welford's method. Overheads, in processor cycles, are measured for each approach every time a sample is obtained. Results are shown in Fig. 7; unsurprisingly, the standard two-pass variance computation times grow linearly with the number of number of samples (about 69 cycles per sample), while Welford's method remains relatively constant (150 cycles on average). We note that the initial larger overheads for the first few samples are likely due to cache effects.

VII. Conclusions and Future Work

This paper presents Tintin [19], a new hardware event profiling infrastructure originally published at OSDI, to the real-time systems community. Tintin aims to schedule events to reduce errors due to multiplexing atop limited HPCs. Tintin is a general-purpose, Linux-based tool. Although it does not specifically target real-time systems, hardware profiling is nonetheless important to our community, as evidenced by the PMThread functionality in RT-Bench [23], [24]. Moreover, Tintin leverages the elastic scheduling model of Buttazzo et al. [45], [46] to minimize measurement uncertainty.

As future work, we intend to release Tintin-Miner, an offline analysis tool that will identify relationships among events so as to identify and remove redundant events from the monitoring pool. It will also perform the sensitivity analysis described in §IV-A to inform weights for Tintin-Scheduler. We will also extend the elastic scheduling policy with principled support for event groups and CPU platforms where certain HPC registers cannot accommodate certain event types.

ACKNOWLEDGMENT

This work was supported by the NSF (CNS-2154930, CNS-2229427, CNS-2141256, CNS-2403758, CNS-2229290), the ARO (W911NF-24-1-0155), the ONR (N00014-24-1-2663), a WashU OVCR seed grant, and Intel.

REFERENCES

[1] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic process groups in dos," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 177–191.

[2] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 377–389.

[3] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 97–108. [Online]. Available: https://doi.org/10.1145/1508244.1508256

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[5] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "Dmon: Efficient detection and correction of data locality problems using selective profiling," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. Virtual: USENIX Association, Jul. 2021, pp. 163–181. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/khan

[6] D. Chen, D. X. Li, and T. Moseley, "Autofdo: automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 12–23. [Online]. Available: https://doi.org/10.1145/2854038.2854044

[7] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson, "Lightweight, high-resolution monitoring for troubleshooting production systems," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 103–116.

[8] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 46–55, 2009.

[9] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen, "Power containers: An os facility for fine-grained power and energy management on multicore servers," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 65–76, 2013.

[10] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars, "Happy: Hyperthread-aware power profiling dynamically," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 211–218.

[11] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[12] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020.

[13] R. Gifford, N. Gandhi, L. T. X. Phan, and A. Haeberlen, "Dna: Dynamic resource allocation for soft real-time multicore systems," in *27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Nashville, TN, USA: IEEE, 2021, pp. 196–209.

[14] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–587. [Online]. Available: https://doi.org/10.1145/3575693.3578835

[15] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "Dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190555

[16] S. Das, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *Symposium on Security and Privacy (SP)*, IEEE. Oakland,CA,USA: IEEE, 2019, pp. 20–38.

[17] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 559–570. [Online]. Available: https://doi.org/10.1145/2485922.2485970

[18] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd international workshop on performance analysis of workload optimized systems, FastPath*, vol. 13. Austin, TX: -, 2013, p. 5.

[19] A. Li, M. Sudvarg, Z. Li, S. Baruah, C. Gill, and N. Zhang, "Tintin: A unified hardware performance profiling infrastructure to uncover and manage uncertainty," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025. [Online]. Available: https://sudvarg.com/publications/OSDI25_Tintin.pdf

[20] M. Sudvarg, C. Gill, and S. Baruah, "Linear-time admission control for elastic scheduling," *Real-Time Systems*, vol. 57, no. 4, pp. 485–490, Oct 2021. [Online]. Available: https://doi.org/10.1007/s11241-021-09373-4

[21] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan, "Time interpolation: So many metrics, so few registers," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. USA: IEEE Computer Society, 2007, p. 286–300. [Online]. Available: https://doi.org/10.1109/MICRO.2007.42

[22] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[23] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, "RT-Bench: An extensible benchmark framework for the analysis and management of real-time applications," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022, pp. 184–195.

[24] M. Nicolella, D. Hoornaert, and R. Mancuso, "RT-Bench: A long overdue update," in *Proc. of 19th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2025.

[25] A. Limited, "Arm cortex-a53 technical reference manual r0p4," https://developer.arm.com/documentation/ddi0500/j/Performance-Monitor-Unit/Events?lang=en, accessed 2023-11-16.

[26] ——, "Arm cortex-a78 technical reference manual," https://developer.arm.com/documentation/101430/0102/Debug-descriptions/Performance-Monitoring-Unit/PMU-events, accessed 2023-11-16.

[27] G. Zellweger, D. Lin, and T. Roscoe, "So many performance events, so little time," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2967360.2967375

[28] P. Wiki, "Tutorial: Linux kernel profiling with perf," https://perf.wiki.kernel.org/index.php/Tutorial, 2023, accessed on: 2023-11-16.

[29] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, p. 189–204, aug 2000. [Online]. Available: https://doi.org/10.1177/109434200001400303

[30] Intel Corporation, "Emon user's guide," https://www.intel.com/content/www/us/en/content-details/686077/emon-user-s-guide.html, 2023, accessed: 2023-12-01.

[31] J. Reinders, *VTune performance analyzer essentials*. Santa Clara: Intel Press, 2005, vol. 9.

[32] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[33] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 409–425.

[34] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian, "Counterminer: Mining big performance data from hardware counters," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. Fukuoka, Japan: IEEE Press, 2018, p. 613–626. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00056

[35] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*. IEEE, 2010, pp. 41–51.

[36] I. Corp., "Intel 64 and ia-32 architectures software developer manuals," https://software.intel.com/en-us/articles/intel-sdm, 2016, accessed 2019-03-05.

[37] M. Dimakopoulou, S. Eranian, N. Koziris, and N. Bambos, "Reliable and efficient performance monitoring in linux," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Salt Lake City, Utah: IEEE Press, 2016.

[38] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, "Bayesperf: Minimizing performance monitoring errors using bayesian statistics," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 832–844. [Online]. Available: https://doi.org/10.1145/3445814.3446739

[39] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[40] W. Mathur and J. Cook, "Toward accurate performance evaluation using hardware counters," pp. 23–32, 2003.

[41] ——, "Improved estimation for software multiplexing of performance counters," in *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '05. USA: IEEE Computer Society, 2005, p. 23–34.

[42] C. W. Gardiner, *Stochastic Methods: A Handbook for the Natural and Social Sciences*, 4th ed. Springer, 2009. [Online]. Available: https://link.springer.com/book/10.1007/978-3-540-70713-4

[43] I. M. Sobol, "Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates," *Mathematics and computers in simulation*, vol. 55, no. 1-3, pp. 271–280, 2001.

[44] T. Chantem, X. S. Hu, and M. D. Lemmon, "Generalized elastic scheduling," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, ser. RTSS '06. USA: IEEE Computer Society, 2006, p. 236–245. [Online]. Available: https://doi.org/10.1109/RTSS.2006.24

[45] G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Proceedings of the IEEE Real-Time Systems Symposium*, ser. RTSS '98. USA: IEEE Computer Society, 1998, p. 286.

[46] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management," *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289–302, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/12.990127

[47] M. Sudvarg, C. Gill, and S. Baruah, "Improved implicit-deadline elastic scheduling," in *2024 IEEE 14th International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2024, pp. 50–57.

[48] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–12.

[49] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 55–64.

[50] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in Multi-Core systems," in *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, Aug. 2007. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/memory-performance-attacks-denial-memory-service-multi

[51] S. A. Panchamukhi and F. Mueller, "Providing task isolation via tlb coloring," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 3–13.

[52] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 357–367.

[53] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 200–212.

[54] M. Bechtel and H. Yun, "Memory-aware denial-of-service attacks on shared cache in multicore real-time systems," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[55] A. Li, M. Sudvarg, H. Liu, Z. Yu, C. Gill, and N. Zhang, "Polyrhythm: Adaptive tuning of a multi-channel attack template for timing interference," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 225–239.

[56] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[57] man7.org Linux Man-pages project, *perf_event_open(2) - Linux Manual Page*, 2023, accessed: 2024-11-02. [Online]. Available: https://man7.org/linux/man-pages/man2/perf_event_open.2.html

[58] m0nad, "Diamorphine - lkm rootkit for linux kernels," https://github.com/m0nad/Diamorphine, 2023, accessed: 2025-04-06.

[59] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ser. ASIACCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 457–468. [Online]. Available: https://doi.org/10.1145/3196494.3196515

Towards a Linux-based Unikernel for Resource-Constrained Embedded Systems

Yoshifumi Shu * , Yutaka Matsubara * , Yixiao Li * , Hiroaki Takada *
* Graduate School of Informatics, Nagoya University, Aichi, Japan
E-mail: {shu.yoshifumi, yutaka, liyixiao, hiro}@ertl.jp

***Abstract*—Embedded Unikernel Linux (EUKL) is a Linux-based unikernel designed for resource-constrained embedded systems, which typically lack an MMU and hardware-assisted virtualization support. EUKL runs as a real-time operating system (RTOS) application. To the best of our knowledge, EUKL is the first approach that enables the simultaneous support of real-time applications and Linux applications in a single resource-constrained embedded system without compromising real-time capability or reliability.**

***Index Terms*—real-time systems, reliability.**

I. Introduction

There are demands to utilize feature-rich software assets already available in Linux (e.g., GUI frameworks, network stacks) to reduce development costs in embedded systems. However, embedded systems often interact with physical world and require hard real-time performance and high reliability. Ensuring real-time capability with Linux alone is challenging due to its enormous code base. Thus, the dual-OS approach by virtualization that runs Linux and a real-time operating system (RTOS) simultaneously (Fig.1) is commonly used to support both Linux applications and RTOS applications.

Resource-constrained embedded systems (e.g., Arm Cortex-M platform) are embedded systems with strict hardware resource constraints and typically lack both a Memory Management Unit (MMU) and hardware-assisted virtualization support. These significantly contribute to minimizing size, weight, power, and cost (SWaP-C), thus they are widely adopted in real-world products where SWaP-C is important.

If Linux applications are also available in resource-constrained systems, it is possible to develop feature-rich and complicated embedded applications with low SWaP-C and development costs. However, the previously mentioned virtualization solution is hard to apply to resource-constrained embedded systems due to the lack of hardware-assisted virtualization support. To the best of our knowledge, there is no existing solution to simultaneously support real-time applications and Linux applications in a single resource-constrained embedded system without compromising real-time capability or reliability of real-time applications. This situation forces developers either to compromise SWaP-C by adopting the virtualization approach with more expensive hardware where hardware-assisted virtualization is supported, or to incur high development costs.

Therefore, our goal is to (1) simultaneously support real-time applications and Linux applications (2) within a single resource-constrained embedded system (3) without compromising real-time capability or reliability, for the purpose of reducing both SWaP-C and development costs in feature-rich embedded applications.

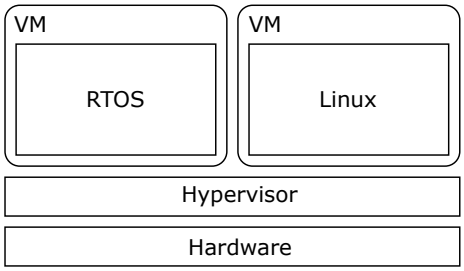


Fig. 1: Conventional approach: Dual OS by virtualization

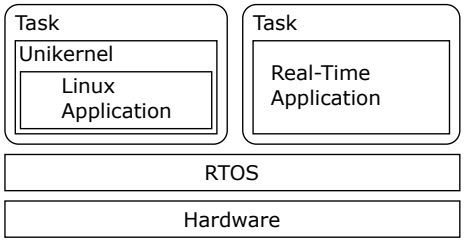


Fig. 2: Proposed approach: Dual OS by running a Linux-Based unikernel on an RTOS

To archive this goal, we propose Embedded Unikernel Linux (EUKL), a Linux-based unikernel suitable for resource-constrained embedded systems. EUKL is inspired by and a fork of Linux Unikernel (UKL) [1], evolved in the cloud-computing context.

II. Background

In this section, we describe the background technologies relevant to the key idea of EUKL.

A. Reliable RTOSes

Reliable RTOSes are RTOSes which provide strong spatial- and temporal- isolation to their user-space applications (e.g., TOPPERS/HRP3 [2], Zephyr [3]). These can support resource-constrained embedded systems equipped with a memory protection unit (MPU), used to spatial-isolation.

B. Unikernels

Unikernels [4], [5] are single-address-space OSes, highly optimized for specific applications. They have been primarily researched in the context of cloud- and edge-computing. Assuming the presence of a hypervisor, unikernels do not provide isolated environments to user applications, but provide rich features which are convenient to implement complex applications. Some of them provide POSIX-compatibility, even Linux-compatibility [1], [6]–[9]. This enables to highly optimize both kernel and application at the same time and realize high throughput and resource efficiency, while reusing numerous existing feature-rich and complicated application code.

Unikernels can be classified into two categories based on their implementations:

1) Clean-slate unikernels: Are unikernels whose a large portion of code is developed from scratch [6]–[8].

2) Existing GPOS-based unikernels: Are unikernels realized by modifying an existing general purpose operating system (GPOS) such as Linux and NetBSD [1], [9], [10].

C. UKL (Unikernel Linux)

UKL (Unikernel Linux) [1] is a prior work on a Linux-based unikernel. This converts Linux into a unikernel-style OS with a few modifications (1250 SLOC modifications against the Linux kernel). UKL provides kernel-level Linux compatibility, in addition to the characteristics observed in unikernels. The original implementation targets x86 bare-metal and virtual machine environments, which is not available in resource-constrained embedded systems.

D. Nommu Linux

Nommu Linux is a Linux kernel without using an MMU and available in the upstream Linux kernel source code with the `CONFIG_MMU=n` configuration. This is also known as μCLinux. Nommu Linux is capable of running in some resource-constrained embedded systems.

Typically, μClibc is used as a C standard library on Nommu Linux.

III. Key Idea of EUKL

The key idea of EUKL is to run a Nommu-Linux-based unikernel inside a user-space of a reliable RTOS (Fig.2).

This idea utilizes the strong isolation provided by the reliable RTOS instead of a hypervisor in the virtualization solution (Fig.1). Thus, the Linux-based unikernel provides a Linux execution environment, while the reliable RTOS protects native RTOS applications from the Linux kernel and its applications.

Owing to its design principle of single-address-space and typically single-privilege-level, a unikernel should be feasible within a user-space of an RTOS. Moreover, basing Nommu Linux for the unikerenl makes the Linux-based unikernel feasible in resource-constrained embedded systems itself. It also reduces the effort required to port the Linux-based unikernel onto an RTOS user-space, compared to the case of utilizing MMU-enabled Linux as UKL does. Thus, this idea is feasible.

IV. Design of EUKL

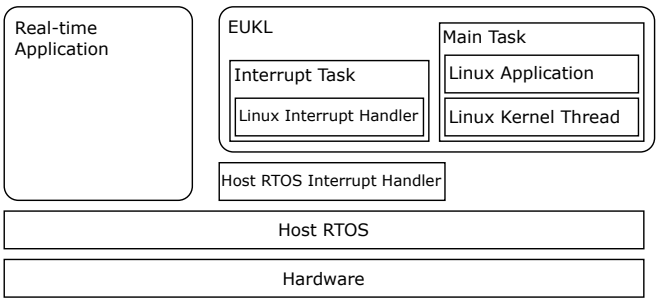


Fig. 3: Overview of EUKL architecture.

Fig.3 shows the overall architecture of a resource-constrained embedded system that utilizes EUKL with a reliable RTOS. All the Linux code completely runs inside an RTOS user-space to apply isolation by the RTOS. The Linux code runs inside two dedicated tasks. One (`Main Task`) is to run all the tasks in the Linux. The other (`Interrupt Task`) is to run interrupt handlers in the Linux. However, interrupt handlers (`Host RTOS Interrupt Handler`) run in the RTOS kernel, in order to virtualize interrupts assigned to the Linux.

In this section, we describe the detailed design of EUKL.

A. Nommu-Linux-based Unikernel

Nommu Linux is converted into a Linux-based unikernel in the same way as UKL:

1) Run both kernel and application at the same CPU privilege level: Mainly, the context switch part in the Linux kernel is modified.

2) Call a system call via a function call: The code calling a system call via a software-generated interrupt (e.g., an interrupt generated by the `svc` instruction in Arm Cortex-M) in μClibc is modified to call it via a simple function call.

3) Static link Linux kernel and user application into single binary: The program loader in the Linux kernel is modified to minimize dynamic loading applications. This change improves memory efficiency since memory fragmentation, including that caused by dynamic loading is a serious problem in resource-constrained embedded systems where MMU-based virtual memory is not available.

B. Unikernel on RTOS

Next, the Linux-based unikernel is ported to an RTOS user-space without modifying the RTOS. The main task of porting is to solve conflicts between Linux and the RTOS and apply isolation by the RTOS.

1) Port multi-tasking inside Linux: M:1 model and non-preemeptive scheduling is adopted to realize multi-tasking inside Linux. M:1 model (i.e., running all Linux tasks within a single task of the host RTOS) is adopted over 1:1 model (i.e., running each Linux task within a dedicated task of the host RTOS) since 1:1 model requires dynamic generating tasks, which poses additional requirements to the host RTOS.

Next, non-preemptive scheduling (also known as co-operative scheduling) is adopted since preemptive scheduing inside a single task also poses some requirements to the host RTOS.

2) Virtualize interrupts: Interrupts to Linux have to be carefully virtualized in order to avoid an effect against real-time performance of the host RTOS via those interrupts. Especially, all interrupt disabling used in Linux's mutual exclusion is performed by a special atomic operation (e.g., `cpsid` instruction in Arm Cortex-M) and, thereby, affect on interrupt latencies and responses of other RTOS applications.

Algorithm 1 describes the currently implemented logic in EUKL, in order to virtualize interrupts assigned to the Linux. In this algorithm, the operations of global interrupt disable from Linux are virtualized by disabling only triggered interrupts during the global disable (EUKLDisableAllInterrupt / EUKLEnableAllInterrupt). On the other hand, the interrupt handler in the host RTOS (HostInterruptHandler) manually disables all interrupts assigned to the Linux. By doing so, at most one interrupt assigned to the Linux can occur between the start of a high-priority task's execution and the resumption of the interrupt task's execution. Therefore this minimizes the impact on the high-priority task's worst-case response time (WCRT).

C. Unikernel Optimization

Link Time Optimization (LTO) is applied to both the Linux kernel and its applications to improve resource efficiency, which is important in resource-constrained embedded systems. LTO is a compiler optimization technique often adopted by Unikernels. This performs compiler optimizations against a whole program in a single time. We ported a GCC LTO patch [11] submitted to a Linux kernel mailing list.

In addition to LTO, the system call table implemented in Linux is removed in order to make LTO more effective. This table contains all addresses to a system call function even if it is not used by any user application. This is required to realize to call a system call via a software-generated interrupt. However, each system call is performed via a function call in EUKL, thus the table is not needed. By removing the table, system call functions which are not used by a user application are applied to LTO and expected to be removed.

V. Implementation

Table I lists the pairs of boards and RTOSes currently supported in EUKL. Firstly, EUKL is implemented on Arm v7-M (Cortex-M4) and TOPPERS/ASP3 for prototyping. TOPPERS/ASP3 is an RTOS that differs from TOPPERS/HRP3 in that it lacks isolation mechanisms. Thus, ASP3 is not a reliable RTOS. However, RTOSes without any isolation mechanism are also applicable to EUKL itself if the system doesn't require reliability of other RTOS applications. Now, we also support Arm v8-M (Cortex-M33) and TOPPERS/HRP3.

We believe EUKL is applicable to other RTOSes (e.g., Zephyr) with a few porting efforts since EUKL is designed carefully so that EUKL avoids using functions unique to TOPPERS/ASP3 or HRP3 for its portability.

Algorithm 1 Interrupt Virtualization Algorithm

```
1: function InitializeEUKLInterrupt
2:     for all irq ← irqs do
3:         eukl_irq_assigned[irq] ← Is irq assigned to the EUKL?
4:         eukl_irq_enabled[irq] ← 0
5:     end for
6:     eukl_all_irq_enabled ← 0
7:     eukl_active_irq ← −1
8:     eukl_pending_irq ← −1
9: end function
10: function EUKLDisableInterrupt(irq)
11:     if eukl_irq_assigned[irq] then
12:         HostLockCPU
13:         eukl_irq_enabled[irq] ← 0
14:         HostDisableInterrupt(irq)
15:         HostUnlockCPU
16:     end if
17: end function
18: function EUKLEnableInterrupt(irq)
19:     if eukl_irq_assigned[irq] then
20:         eukl_irq_enabled[irq] ← 1
21:         HostEnableInterrupt(irq)
22:     end if
23: end function
24: function EUKLDisableAllInterrupt
25:     eukl_all_irq_enabled ← 0
26: end function
27: function RaiseEUKLInterrupt(irq)
28:     EUKLDisableAllInterrupt
29:     eukl_active_irq ← irq
30:     HostActivateTask(EUKLInterruptTask)
31: end function
32: function EUKLEnableAllInterrupt
33:     eukl_all_irq_enabled ← 1
34:     if eukl_pending_irq >= 0 then
35:         irq ← eukl_pending_irq
36:         eukl_pending_irq ← −1
37:         RaiseGuestInterrupt(irq)
38:     end if
39: end function
40: function HostInterruptHandler(irq)
41:     for all irq ← irqs do
42:         if eukl_irq_enabled[irq] then
43:             HostDisableInterrupt(irq)
44:         end if
45:     end for
46:     if eukl_all_irq_enabled then
47:         RaiseEUKLInterrupt(irq)
48:     else
49:         eukl_pending_irq ← irq
50:     end if
51:     HostWakeupTask(EUKLMainTask)
52: end function
53: function EUKLInterruptTask
54:     EUKLInterruptHandler(active_irq)
55:     eukl_active_irq ← −1
56:     GuestEnableAllInterrupt
57:     for all irq ← irqs do
58:         HostLockCPU
59:         if eukl_irq_enabled[irq] and eukl_active_irq == −1 then
60:             HostEnableInterrupt(irq)
61:         end if
62:         HostUnlockCPU
63:     end for
64: end function
```

TABLE I: Currently supported boards and RTOSes in EUKL

Board	CPU	Host RTOS	Protection mechanism
STM32F429I-DISC1	Arm Cortex-M4	TOPPERS/ASP3	Not supported
MPS2+ AN505 on QEMU	Arm Cortex-M33	TOPPERS/HRP3	Supported
STM32U5G9J-DK1	Arm Cortex-M33	TOPPERS/HRP3	Supported

VI. Evaluation

We conducted an evaluation of EUKL implemented on STM32F429I-DISC1 [12](Table II) and TOPPERS/ASP3 by running a hello world application in C as a test Linux application.

The impact on the WCRT of a high-priority real-time application is calculated as 12.4 µs + 0.4µs × (number of interrupts assigned to EUKL). This shows that the impact on real-time capability is sufficiently minimal in many cases.

The ROM usage and RAM footprint are also evaluated for resource efficiency. The best ROM usage of EUKL(`eukl-lto-effective`) decreases by 30% compared to vanilla Nommu Linux(`vanilla`) (Table III). The RAM footprint (i.e., the minimum RAM size required to run the application) of EUKL(`eukl`) is also decreased by 30% compared to vanilla Nommu Linux(`vanilla`) (Table IV).

These results show that the impact on real-time capability is sufficiently minimal and resource efficiency is improved, demonstrating that our approach is practical and beneficial for resource-constrained embedded systems.

VII. Related Works

Table V shows a comparison table of methods that allow co-running RTOS and Linux while maintaining real-time capability.

As previously mentioned, hardware-assisted virtualization provides isolated execution environments for both Linux and RTOSes, thereby ensuring the reliability of RTOS applications. However, it is not suitable for resource-constrained embedded systems.

The Co-kernel approach [13]–[15] achieves a DualOS architecture by sharing the kernel space between the Linux kernel and an RTOS kernel. While it is feasible in resource-constrained embedded systems, it lacks isolation between the kernels, and therefore cannot ensure reliability of RTOS applications.

EUKL is the only approach that both ensures the reliability of RTOS applications(`Reliability`) and is feasible to resource-constrained embedded systems(`Feasibility`). Furthermore, by applying unikernel optimizations, it also achieves improved resource usage efficiency relative to other methods.

VIII. Conclusion

In this paper, we presented EUKL, a Linux-based unikernel running inside a reliable RTOS user-space. EUKL aims to allow co-existance of Linux applications and RTOS applications within a single resource-constrained embedded system while maintaining real-time capability and reliability of the RTOS applications. For resource efficiency, imporovements both in ROM usage and RAM footprint are also observed in the evaluation. EUKL is expected to reduce SWaP-C and development costs in feature-rich embedded applications.

As a future work, we will conduct additional detailed evaluation against real-product-aware and more complicated demo applications. Moreover, we are considering modifying the host RTOS in order to enhance the Linux-based unikernel (e.g., for more better real-time capability) since most RTOSes do not expect to execute an OS as their application. Finally, we have a plan to publish EUKL as an open-source project on the Internet targeting both research and industrial usage purposes.

References

[1] A. Raza, T. Unger, M. Boyd, E. B. Munson, P. Sohal, U. Drepper, R. Jones, D. B. De Oliveira, L. Woodman, R. Mancuso, J. Appavoo, and O. Krieger, "Unikernel linux (ukl)," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 590–605. [Online]. Available: https://doi.org/10.1145/3552326.3587458

[2] TOPPERS Project, Inc., "TOPPERS Project/HRP3." [Online]. Available: https://www.toppers.jp/hrp3-kernel.html

[3] Zephyr Project, "The Zephyr Project – A proven RTOS ecosystem, by developers, for developers." https://www.zephyrproject.org/.

[4] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 461–472. [Online]. Available: https://doi.org/10.1145/2451116.2451167

[5] A. Madhavapeddy and D. J. Scott, "Unikernels: The rise of the virtual library operating system," *Commun. ACM*, vol. 57, no. 1, p. 61–69, jan 2014. [Online]. Available: https://doi.org/10.1145/2541883.2541895

[6] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv—Optimizing the operating system for virtual machines," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 61–72. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[7] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 59–73. [Online]. Available: https://doi.org/10.1145/3313808.3313817

[8] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, "Unikraft: Fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 376–394. [Online]. Available: https://doi.org/10.1145/3447786.3456248

[9] H.-C. Kuo, D. Williams, R. Koller, and S. Mohan, "A linux in unikernel clothing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387526

TABLE II: Specifications of STM32F429I-DISC1

Component	Description
CPU	Arm Cortex-M4 (Arm v7-M) @ 168MHz
Number of Cores	Single Core
MPU	Implemented (PMSAv7)
MMU	Not implemented
Hardware-Assisted Virtualization	Not supported
RAM	Built-in SRAM: 245KB, External SDRAM: 8MB
ROM	2MB

TABLE III: ROM Usage

Target	Usage [KB]	Description of Target
vanilla	1353	vanilla Nommu Linux
vanilla-lto	990	vanilla + LTO applied
vanilla-lto-kernel-only	988	vanilla Nommu Linux kernel only with LTO applied
eukl	1506	EUKL without LTO
eukl-lto	963	eukl + LTO applied
eukl-lto-effective	886	eukl-lto + improved LTO efficiency by removing system call table

TABLE IV: RAM footprint

Target	Footprint [KB]	Description of Target
vanilla	2646	vanilla Nommu Linux
eukl	1860	EUKL without LTO

TABLE V: Comparison of approaches co-running RTOS and Linux

approach	**Feasibility**	**Reliability**	**Resource Efficiency**
EUKL (proposed)	✓	✓	✓
Hardware-Assisted Virtualization	-	✓	-
Co-kernel	✓	-	-

[10] A. Kantee and J. Cormack, "Rump Kernels: No OS? No Problem!" *;login: USENIX*, vol. 39, no. 5, p. 11–17, oct 2014. [Online]. Available: https://www.usenix.org/publications/login/october-2014-vol-39-no-5/rump-kernels-no-os-no-problem

[11] J. Slaby, "[patch 00/46] gcc-lto support for the kernel," https://lore.kernel.org/lkml/20221114114344.18650-1-jirislaby@kernel.org/.

[12] STMicroelectronics, "32f429idiscovery - discovery kit with stm32f429zi mcu * new order code stm32f429i-disc1 (replaces stm32f429i-disco)," https://www.st.com/en/evaluation-tools/32f429idiscovery.html.

[13] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "RTAI: Real Time Application Interface," *Linux J.*, vol. 2000, no. 72es, p. 10–es, apr 2000.

[14] P. Gerum, "Xenomai-implementing a RTOS emulation framework on GNU/Linux," *White Paper, Xenomai*, p. 81, 2004.

[15] H. Takada, S. Iiyama, T. Kindaichi, and S. Hachiya, ""Linux on ITRON": a hybrid operating system architecture for embedded systems," in *Proceedings 2002 Symposium on Applications and the Internet (SAINT) Workshops*, 2002, pp. 4–7.

Compute Kernels as Moldable Tasks: Towards Real-Time Gang Scheduling in GPUs

Attilio Discepoli
Vrije Universiteit Brussel
Brussels, Belgium
attilio.discepoli@vub.be
0009-0001-7100-3909

Mathias Louis Huygen
Vrije Universiteit Brussel
Brussels, Belgium
mathias.louis.huygen@vub.be

Antonio Paolillo
Vrije Universiteit Brussel
Brussels, Belgium
antonio.paolillo@vub.be
0000-0001-6608-6562

***Abstract*—We present a preliminary evaluation of real-time scheduling policies for GPU kernels modeled as moldable tasks. Our framework maps periodic real-time jobs to CUDA kernels and uses `libsmctrl` to assign Texture Processing Clusters (TPCs), enabling gang scheduling with variable parallelism. A calibration tool provides per-kernel WCET estimates across different TPC counts, allowing the scheduler to trade execution time for resource usage.**

We compare several scheduling strategies, including CUDA's default concurrent kernel execution ("all-out"), a sequential EDF policy using all TPCs per job, and a moldable EDF scheduler that dynamically allocates just enough TPCs to meet each job's deadline. Using static per-task memory allocation, we eliminate prior sources of interference and achieve WCET predictability.

Our results show that deadline-aware scheduling outperforms the default CUDA strategy in scenarios with urgency mismatches. Moreover, moldable EDF improves over sequential EDF by reducing deadline misses under non-preemptive execution, especially when long-running jobs could block shorter urgent ones.

***Index Terms*—GPU Partitioning, Gang Scheduling, Moldable Tasks, WCET Estimation, Non-Preemptive Scheduling, CUDA.**

I. Introduction

GPUs are increasingly integrated into real-time and safety-critical systems, including autonomous vehicles, robotics, and industrial automation. While their parallelism offers significant performance benefits, GPUs remain challenging to integrate in systems requiring predictability. One key obstacle is the lack of fine-grained control over how GPU resources are shared and scheduled across concurrent workloads.

In response to this, recent efforts have explored ways to partition GPU resources, such as using NVIDIA's Streaming Multiprocessor (SM) partitioning feature, exposed via tools like `libsmctrl` [1]. Partitioning promises isolation between workloads by assigning subsets of SMs—called Texture Processing Clusters (TPCs)—to different jobs. This opens the door to applying real-time scheduling techniques such as Earliest Deadline First (EDF), gang scheduling, or moldable parallelism on GPUs.

Following the terminology of Goossens and Berten [2], a parallel job in a gang scheduling system is said to be *rigid* if its processor allocation is fixed externally and never changes, *moldable* if the scheduler decides the allocation at release time, and *malleable* if the allocation can change during execution. In our case, the allocated resource under consideration is the number of TPCs assigned to a kernel launch.

In this work, we explore the feasibility and benefits of deadline-aware *moldable scheduling* for GPU kernels, leveraging runtime TPC assignment to adapt to system load. We develop a custom CUDA scheduling framework that models periodic real-time task sets, where each job corresponds to a GPU kernel execution. Using `libsmctrl`, we assign TPC masks at runtime based on the job's urgency and its worst-case execution time (WCET) profile, measured as a function of TPC count.

We implement several schedulers, including CUDA's default concurrent launch policy ("all-out"), a sequential EDF policy that assigns all TPCs to one job at a time, and a moldable EDF scheduler that dynamically adapts job parallelism based on deadline constraints. Our evaluation shows two key findings. First, the "all-out" strategy, while aggressive in parallelism, is unaware of deadlines and can underperform even simple sequential EDF in deadline-sensitive workloads. Second, the moldable EDF scheduler improves over both baselines by assigning only the resources necessary to meet each job's deadline, thus avoiding deadline misses—particularly in non-preemptive scenarios where greedy jobs might otherwise block shorter, urgent ones.

These results demonstrate that combining SM partitioning with moldable real-time scheduling can outperform default GPU execution strategies, and offer a promising direction for predictable GPU integration in real-time systems.

II. Framework Overview

We developed a custom runtime framework for evaluating GPU scheduling strategies under a real-time task model. Each task corresponds to a periodic stream of GPU jobs, implemented as CUDA kernels, and is characterized by a period, relative deadline, and kernel type. The kernel type of a task defines the specific workload executed by its jobs. It implicitly determines the Worst-Case Execution Time (WCET) of a job as a function of the GPU resources (*e.g.*, TPCs) assigned to it, as detailed below. All jobs are released on a fixed periodic schedule and are executed non-preemptively. The scheduler is invoked on each job release and completion event to determine which jobs to run next and how to assign GPU resources.

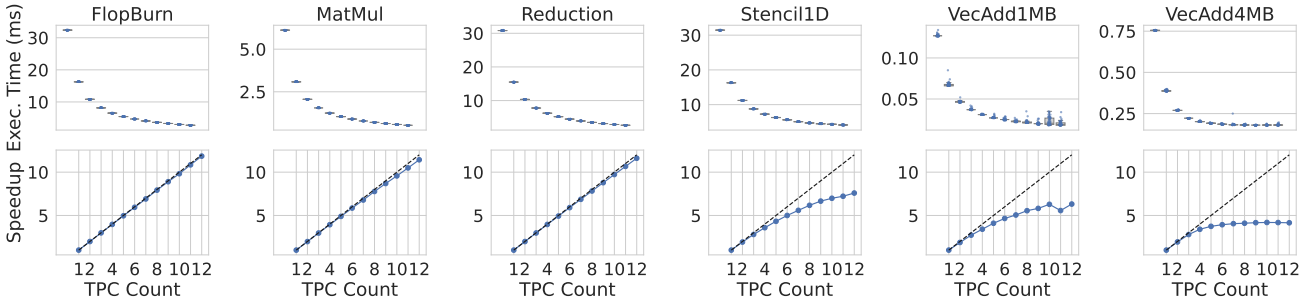


Fig. 1. Execution times (top row) and speedups (bottom row) of CUDA kernels under increasing TPC count (from 1 to 12). Compute-bound kernels (`FlopBurn`, `MatMul`, `Reduction`) show near-linear speedups, closely tracking the ideal scaling line (dashed). `Stencil1D` also scales well, but with diminishing returns due to memory reuse patterns (non-coalesced accesses). In contrast, memory-bound kernels (`VecAdd4MB`) saturate early, showing limited speedup beyond 4-6 TPCs. These results are used by the scheduler to estimate empirically the worst-case execution time as a function of TPC count.

SM Partitioning and TPC Masks. To control GPU resource usage, we use `libsmctrl`, a library that enables fine-grained partitioning of NVIDIA GPUs by restricting kernels to specific subsets of Streaming Multiprocessors (SMs). On our GPU model, a TPC corresponds to two SMs, and the TPC mask of a job defines the subset of SMs it is allowed to run on. This allows the scheduler to enforce isolation policies by selecting non-overlapping TPC masks for concurrent jobs.

Scheduler and Policy Interface. The framework supports pluggable scheduling policies. Schedulers implement a uniform interface and maintain their own internal state, such as the list of pending jobs. The interface consists of:

- `on_job_release(GpuJob*)`
- `on_job_complete(GpuJob*)`
- `std::vector<GpuJob*> select_jobs_to_run()`

For example, EDF-based schedulers use a priority queue internally, sorted on absolute deadlines, and allocate TPCs based on job urgency. All scheduling decisions are made online and are executed by a central loop that launches jobs using their assigned TPC masks. This design enables experimentation with various scheduling policies, including those supporting parallel jobs. We support:

- `seq-edf`: a sequential EDF policy where jobs run one at a time with all TPCs;
- `global-edf-1tpc`: each job runs with one TPC, allocated greedily;
- `moldable-edf`: each job is assigned the smallest TPC count that satisfies its WCET–deadline pair;
- `all-out` (baseline): every job is launched immediately, in a default-priority CUDA stream, with a full TPC mask (*i.e.*, no partitioning). On current NVIDIA GPUs, the Task Management Unit (TMU) and Work Distribution Unit (WDU) dispatch such kernels in *FIFO order within each priority level* [1]. Because all our streams use the default (equal) priority and no kernels spawn child kernels (*i.e.*, no CUDA Dynamic Parallelism), the dispatch order is reproducible across repeated runs of the same task set. This policy represents the default behavior of the CUDA runtime.

WCET Calibration. To estimate per-job execution time under different TPC configurations, we developed a WCET calibration tool. This tool runs each kernel type in isolation across 1 to N TPCs and records execution times. These measurements populate a per-kernel-type WCET vector made available to the scheduler. During execution, the scheduler can use this vector to reason about how many TPCs to assign to a job in order to meet its deadline.

Task-Based Memory Management. To ensure consistency and eliminate memory-related timing anomalies, each task acts as a "vessel" for its jobs, encapsulating both the kernel type and the memory buffers needed for execution. At program startup, each task allocates the required data structures in GPU device memory; these buffers are then reused by every job released by the task, avoiding any per-job allocation or deallocation during execution. This setup prevents host-to-device and device-to-host data transfers during execution, ensuring that all data resides in GPU memory throughout the experiment. It also isolates compute behavior from interference caused by shared memory traffic or copy engines—issues we leave to future work. This design significantly improves timing predictability. In earlier experiments, dynamic allocations and implicit data transfers introduced substantial timing noise and unintended synchronization effects, as also observed by Yang et al. [3]. With this simplification, execution times became remarkably more stable, and WCET overruns were largely eliminated.

Experimental Testbed. All experiments were conducted on an NVIDIA RTX 2000 Ada Generation Laptop GPU, a *discrete* GPU with 24 Streaming Multiprocessors (SMs), corresponding to compute capability 8.9. The system was configured with NVIDIA driver version 570.133.20 and CUDA toolkit version 12.8.

Frequency Control. To reduce variability and ensure repeatable WCET measurements, we disabled dynamic frequency scaling (DVFS) and fixed both the core and memory clocks using `nvidia-smi`. Specifically, we enabled persistence mode and locked the graphics and memory clocks to 2115 MHz and 7001 MHz, respectively: `nvidia-smi -pm 1`, `-lgc 2115`, `-lmc 7001,7001`. This eliminates one

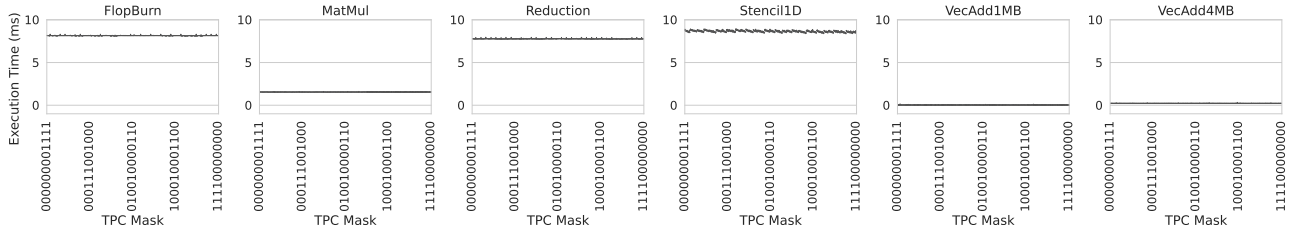


Fig. 2. Impact of TPC placement on execution time for a fixed TPC count ($k = 4$). Each subplot corresponds to a different CUDA kernel and shows execution time variation across all valid 4-TPC masks (out of 12 total). Only a few representative masks are shown on the x-axis for readability. Compute- and memory-bound kernels exhibit consistent runtimes regardless of which TPCs are selected, suggesting minimal sensitivity to placement.

major source of temporal jitter and ensures that observed WCET overruns are not artifacts of frequency throttling.

III. WCET Scaling and Placement Sensitivity

To support moldable scheduling and inform resource allocation decisions, we precompute the execution time of each CUDA kernel type across different levels of parallelism. Our WCET calibration tool executes each kernel in isolation using TPC masks ranging from 1 to 12 active SMs, and records the corresponding execution times. The kernels used in this study span a range of compute and memory intensities:

- **FlopBurn**: a compute-bound kernel that performs repeated trigonometric operations to saturate the floating-point ALUs,
- **MatMul**: a tiled matrix multiplication using shared memory, representative of structured compute workloads,
- **Reduction**: a parallel reduction that combines shared memory and atomic operations, with moderate control divergence,
- **Stencil1D**: a 1D stencil operation with neighborhood data dependencies and moderate memory reuse,
- **VecAdd1MB / VecAdd4MB**: simple element-wise vector additions over 1MB or 4MB arrays, limited by global memory bandwidth.

Figure 1 shows the measured WCETs of these kernels and the corresponding speedups as a function of TPC count. Compute-bound kernels such as `FlopBurn` and `MatMul` scale nearly linearly with the number of TPCs, achieving close to ideal speedup. Memory-bound kernels such as `VecAdd1MB` and `VecAdd4MB`, however, exhibit diminishing returns beyond 4–6 TPCs. `Stencil1D` kernels fall in between: they benefit from parallelism but scale sub-linearly.

These WCET profiles are exposed to the scheduler at runtime and enable reasoning about the trade-off between parallelism and execution time of a job, based on the number of TPCs it is assigned. They serve our moldable scheduler's decision process, which aims to assign the minimum number of TPCs needed to meet a job's deadline.

To assess the robustness of these measurements, we conducted a sensitivity study on TPC placement. Even when the number of assigned TPCs is fixed, the specific SMs selected (*i.e.*, the bit pattern of the TPC mask) could, in principle, affect execution time due to undocumented architectural asymmetries or locality effects. We fixed the number of active TPCs to $k = 4$ and systematically evaluated each kernel across all valid TPC masks with exactly four bits set, resulting in 495 unique placements out of 12 TPCs. Note that the bit logic in our masks is inverted relative to the convention used by `libsmctrl`: a bit set to 1 denotes an active SM in our encoding. Each mask configuration was executed 50 times following a warmup phase. Figure 2 summarizes the resulting distributions. We observe *no significant impact of TPC placement* on execution time. Compute-intensive kernels such as `FlopBurn`, `MatMul`, `Reduction`, and `Stencil1D` exhibit narrow and consistent distributions across all placements. Even memory-bound kernels like `VecAdd1MB` and `VecAdd4MB` do not exhibit systematic sensitivity to SM selection. These results confirm that, in isolation, WCET is primarily determined by the number of assigned TPCs rather than their physical placement.

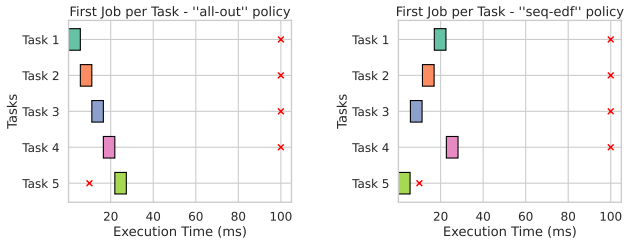
IV. Case Studies: When Scheduling Matters

To highlight the practical implications of deadline-aware scheduling on GPUs, we present two representative case studies drawn from our experiments. These scenarios illustrate how CUDA's default policy and simple sequential execution can both fail under real-time constraints, and how moldable scheduling can mitigate deadline misses.

A. Study 1: Deadline Awareness vs. Parallelism

This scenario compares `seq-edf` with the baseline `all-out` strategy. The taskset consists of five identical jobs. Four of them are tasks with a period and deadline of 100 ms, while one is an "urgent" task with a 10 ms deadline. All jobs execute the same kernel and share the same WCET profile, but the urgent task has significantly tighter timing constraints.

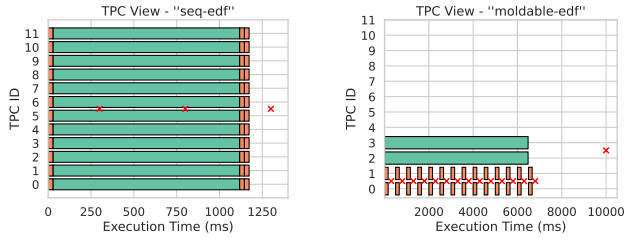
Under `all-out`, all five jobs are launched concurrently at each release. CUDA's default scheduling policy, as noted by Bakita et al. [1], admits kernels in FIFO order with limited prioritization. This means that once a kernel is submitted, all of its blocks must complete before newer kernels can begin execution—even if those newer kernels correspond to more urgent tasks. As a result, the urgent job may be delayed by earlier, less time-sensitive work, leading to deadline misses despite ample compute resources.



(a) `all-out`: jobs are launched concurrently. CUDA queues and schedules them in FIFO order. Urgent tasks are not prioritized.

(b) `seq-edf`: jobs are scheduled one at a time in EDF order. Urgent tasks are executed early and meet their deadlines.

Fig. 3. Case study 1: Comparison between deadline-unaware (`all-out`) and deadline-aware (`seq-edf`) scheduling. Each job uses all TPCs (12), and the taskset contains one urgent task (Task 5) with a tighter deadline. Red crosses mark job deadlines. Under `all-out`, Task 5 misses its deadline due to lack of prioritization. In contrast, `seq-edf` prioritizes it correctly, and all deadlines are met.



(a) `seq-edf`: urgent jobs (orange) are delayed by the greedy job (green), leading to deadline misses for urgent jobs.

(b) `moldable-edf`: only minimal TPCs are assigned to the greedy task, enabling concurrent execution. All deadlines are met.

Fig. 4. Case study 2: Job execution timelines across TPCs. Each horizontal bar represents a kernel execution on a specific TPC. Red crosses mark job deadlines. Moldable scheduling avoids deadline misses by spatially multiplexing tasks.

In contrast, `seq-edf` enforces a strict priority order based on absolute deadlines, running one job at a time using all TPCs. Although this strategy reduces concurrency, it ensures that the most urgent job is executed first, preventing unnecessary deadline violations.

Figure 3 shows the resulting Gantt chart: while `all-out` suffers from misprioritized execution, `seq-edf` guarantees the timely completion of the urgent job.

B. Study 2: Resource-Aware Scheduling under Non-Preemption

This scenario compares `seq-edf` and `moldable-edf` on a two-task workload with non-preemptive execution. The first task is a "greedy" long-running job with a WCET of 1250 ms when using all 12 TPCs, and 7500 ms when limited to 2 TPCs. It has a period of 10 s and a relaxed deadline of 10 s. The second task releases short, urgent jobs with a WCET of approximately 265 ms on 2 TPCs and 25 ms on 12 TPCs, and with a period of 500 ms and a relative deadline of 300 ms.

Under `seq-edf`, jobs are executed sequentially using all 12 available TPCs. At the start of the hyperperiod, both tasks release their first jobs. Since the urgent job has an earlier deadline, it is correctly scheduled first. However, subsequent jobs of the urgent task arrive while the greedy job is executing and are blocked until it completes. Because execution is non-preemptive, the urgent jobs cannot reclaim GPU resources mid-execution and thus miss their deadlines. In contrast, `moldable-edf` dynamically estimates the number of TPCs required to meet each job's deadline based on the precomputed WCET table. It assigns only the necessary TPCs to the greedy job, leaving enough TPCs free to schedule the urgent job concurrently. As a result, `moldable-edf` avoids blocking and ensures both tasks remain schedulable.

Figure 4 shows the Gantt chart highlighting this difference: only `moldable-edf` successfully schedules both tasks without deadline misses.

These results show that moldable scheduling, when combined with SM partitioning, has the potential to enable higher schedulability even in non-preemptive systems.

V. Limitations

Our evaluation is preliminary and subject to several limitations that we plan to address in future work.

Resource isolation. While our framework allows fine-grained TPC partitioning, kernel executions are calibrated and evaluated in isolation, without concurrent memory traffic or host-side interference. Real-world deployments may experience shared resource contention—*e.g.*, across L2 cache slices, DRAM channels, or copy engines—which could impact both execution time and predictability, particularly for memory-bound kernels. As future work, we plan to extend our WCET analysis with interference-aware calibration, using co-scheduled background kernels to quantify slowdowns under contention. This will complement the results shown in Figure 2, where execution times were measured in isolation.

CUDA baseline behavior. We compare against CUDA's default kernel execution behavior, which typically launches kernels in FIFO order across streams. We do not explore alternative configurations enabled by the CUDA runtime, such as per-stream priorities or CUDA Graphs, though these could influence deadline satisfaction under certain workloads.

Application coverage. The current evaluation is limited to synthetic kernels with predictable structure and compute-dominated profiles. While useful to showcase scheduling behavior, real-world GPU workloads—such as DNN inference or sensor fusion—will be explored in future studies.

Formal analysis. Finally, no formal schedulability analysis is provided in this work. We focus on empirical evaluation; integrating analytical models (*e.g.*, response-time analysis) and characterizing worst-case interference under shared resource contention are important next steps.

VI. Related Work

Gang scheduling has long been explored in real-time systems to coordinate parallel workloads under predictable exe-

cution models. Nelissen et al. [4] introduced a response-time analysis framework for non-preemptive, periodic moldable gang tasks, establishing schedulability bounds under job-level fixed-priority (JLFP) policies.

Bakita et al. [1] proposed `libsmctrl`, a user-space mechanism for SM-level partitioning on NVIDIA GPUs, enabling spatial isolation via explicit control over the TPC mask. Our framework builds on this mechanism to support real-time schedulers that explicitly allocate GPU resources to individual jobs at runtime. Follow-up work [5] showed that SM partitioning alone does not ensure isolation, due to shared resource interference from memory controllers, copy engines, and internal arbitration mechanisms. These insights informed our framework design, particularly the need for static memory allocation. Ali et al. [6] recently introduced the Streaming Multiprocessor Locking Protocol (SMLP), which supports predictable intra-component GPU access by dynamically resizing GPU workloads to fit available SMs. SMLP is designed for component-based systems scheduled with JLFP inside time-sliced partitions, and offers analytical bounds on priority-inversion blocking. Their evaluation couples simulation-based blocking analysis with a brief hardware sweep—using `libsmctrl` to measure how one kernel's WCET scales with the number of SMs—to motivate the resizing model. Our work is complementary: we implement a moldable EDF scheduler on real CUDA hardware, calibrate WCETs for several kernels across TPC counts, and measure deadline behaviour under multiple scheduling policies. This empirical perspective lets us quantify the practical benefits of moldable gang scheduling for task sets with heterogeneous deadlines.

VII. Conclusion

We presented a preliminary evaluation of real-time scheduling strategies for GPUs using SM partitioning enabled by `libsmctrl`. Our framework models CUDA kernel executions as periodic tasks, performs WCET calibration across TPC counts, and supports a variety of scheduling strategies, including deadline-unaware (`all-out`), sequential EDF, and a moldable EDF policy. Through controlled case studies, we demonstrated that deadline-aware scheduling can outperform CUDA's default strategy, and that moldable EDF further improves schedulability by minimizing blocking under non-preemptive execution.

This work opens the path toward non-preemptive gang scheduling on GPUs by leveraging SM partitioning. As future work, we plan to develop a full-fledged moldable scheduler tailored to GPU architectures, along with formal schedulability analysis. This includes extending the moldable model to sporadic workloads by tracking per-task cooldown intervals—*i.e.*, known idle phases after job completion that allow safe, temporary reuse of reserved resources.

To further improve timing predictability, we will extend our WCET analysis to evaluate the impact of inter-kernel interference. Such analysis should also be conducted on real-world GPU applications; For instance, Bakita et al. [1] reported results for TPC partitioning on a YOLO workload.

A deeper investigation into memory-related effects—such as copy engine contention and device memory allocation overhead—is an important direction for future work.

Finally, we aim to explore the feasibility of true malleable scheduling, where TPC allocations can be updated dynamically during kernel execution—provided such capabilities are supported on current or emerging GPU architectures.

References

[1] J. Bakita and J. H. Anderson, "Hardware Compute Partitioning on NVIDIA GPUs," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)* . Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 54–66. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/RTAS58335.2023.00012

[2] J. Goossens and V. Berten, "Gang FTP scheduling of periodic and parallel rigid real-time tasks," Jun. 2010, arXiv:1006.2617 [cs]. [Online]. Available: http://arxiv.org/abs/1006.2617

[3] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," *LIPIcs, Volume 106, ECRTS 2018*, vol. 106, pp. 20:1–20:21, 2018, artwork Size: 21 pages, 880171 bytes ISBN: 9783959770750 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[4] G. Nelissen, J. Marcè i Igual, and M. Nasri, "Response-Time Analysis for Non-Preemptive Periodic Moldable Gang Tasks," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Maggio, Ed., vol. 231. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 12:1–12:22.

[5] Bakita, Joshua and Anderson, James H., "Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2024, pp. 294–305.

[6] S. W. Ali, Z. Tong, J. Goh, and J. H. Anderson, "Predictable GPU Sharing in Component-Based Real-Time Systems," in *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Pellizzoni, Ed., vol. 298. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 15:1–15:22.

SentryRT-1: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception

Yuwen Shen*†‡, Jorrit Vander Mynsbrugge*‡, Nima Roshandel*†‡, Robin Bouchez*‡, Hamed FirouziPouyaei*‡, Constantin Scholz*‡, Hoang-Long Cao*§, Bram Vanderborght*‡, Wouter Joosen†, Antonio Paolillo*

*Vrije Universiteit Brussel, Belgium, †KU Leuven, Belgium
‡imec, Belgium, §Can Tho University, Vietnam

***Abstract*—Ensuring timely and safe operation in robotics remains a challenge, especially in systems combining high-throughput perception with real-time safety constraints. SentryRT-1 is a minimal C++/CUDA runtime that integrates multi-camera sensing, GPU-accelerated human detection, and a safety module enforcing Speed and Separation Monitoring (SSM) robot control. In this case study, we model the runtime as a real-time task set and evaluate its behavior under various Linux kernel configurations. Using synthetic interference and replayable camera inputs, we benchmark the latency and determinism of the safety module. Our results show that real-time scheduling policies such as SCHED_DEADLINE significantly reduce both average and worst-case reaction times, and that a real-time kernel with PREEMPT_RT provides further—though less pronounced—improvements. These findings demonstrate the capabilities of Linux-based configurations for safety-critical robotic workloads.**

***Index Terms*—Real-Time Systems, Collaborative Robotics, Safety-Critical, Human-Robot Interaction, Real-Time Linux, Perception Pipeline, Speed and Separation Monitoring**

I. Introduction

Driven by global labor shortages and the accelerating automation trend, robotics is rapidly expanding across industries, particularly in manufacturing [1], [2]. Historically, most robotic deployments occurred in greenfield installations—factories built for automation, typically with fenced-off robotic cells where robots operate at high speed and isolated from humans to ensure safety and throughput [3]. However, many production tasks still rely on manual labor—so-called *brownfield environments*—where automation is introduced incrementally and workspace is often limited [4]. Examples include pharmaceutical packaging, machine tending, and kitting. In these cases, traditional caged robots are impractical due to spatial constraints and the need for continuous human cooperation.

Collaborative robots (cobots) equipped with Power and Force-Limiting (PFL) capabilities can safely operate alongside humans without physical barriers [5]. However, PFL relies on contact-based stopping, which may still cause injuries and imposes strict speed limitations. To overcome these constraints, robots are increasingly equipped with on-robot perception sensors to enable safety through Speed and Separation Monitoring (SSM) [6], standardized by ISO10218 [5], allowing dynamic speed adjustment based on human proximity.

Corresponding author: antonio.paolillo@vub.be

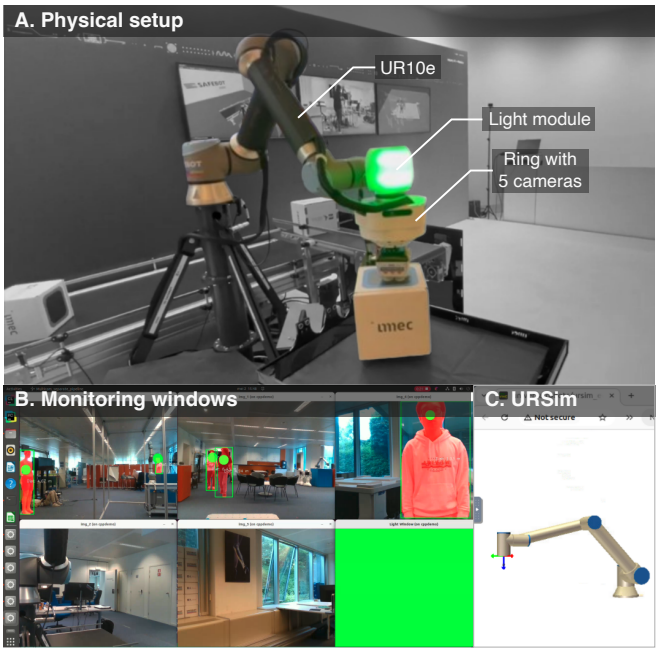


Fig. 1. Overview of the SentryRT-1 platform. (A) The physical setup including a UR10e robotic arm, a ring of five cameras, and a light signaling module. (B) Monitoring windows displaying 5 replay data streams and the virtual light module. (C) URSim, the robot simulation environment.

Achieving such responsiveness requires low-latency, predictable execution across sensing, inference, and actuation pipelines. Linux-based systems—widely used in modern robotic stacks—are attractive due to their ecosystem and hardware support, but it remains unclear how well they meet real-time constraints in these safety-critical scenarios. In particular, there is limited understanding of how different Linux configurations affect end-to-end timing in perception-driven safety loops. This paper addresses this gap through an application-driven analysis of real-time performance under representative a robotic workload.

Our prototype system, illustrated in Figure 1A, consists of a Universal Robots UR10e manipulator, five Intel RealSense D435i depth cameras mounted around the end-effector, an industrial x86 PC with a discrete NVIDIA GPU, and a light signaling module [7] controlled via the robot's digital I/O.

The hardware is able to operate both in live mode – using the setup of Figure 1A – and in simulation – using URSim [8] (Figure 1C) and replayable camera streams. To drive this setup, we develop **SENTRYRT-1**, a C++/CUDA software runtime that integrates multi-camera perception with GPU-accelerated human detection and a safety module enforcing proximity-based speed control. The system avoids middleware such as ROS, providing direct control over threading, memory, and data flow. A screenshot of the live perception and safety feedback is shown in Figure 1B.

In this paper, we model this runtime as a real-time task set and evaluate it under different Linux configurations—including `PREEMPT_RT` and `SCHED_DEADLINE`—to assess their ability to meet real-time constraints under synthetic interference that models high computational demand. Our benchmarking framework supports replayable inputs and repeatable stress conditions, allowing systematic evaluation of safety responsiveness under load. Our results show that `SCHED_DEADLINE` significantly improves reaction time compared to baseline Linux, while the real-time kernel with `PREEMPT_RT` provides further improvements.

II. Related Work

Significant research has been conducted within the real-time computing and robotics communities to design and evaluate the performance of robotic software frameworks.

ROS. The most widely adopted framework is ROS2 [9], which promotes modularity in robotic system design and uses Data Distribution Service (DDS)-based inter-process communication to transfer data between components. ROS2 adopts an event-driven callback mechanism, making timing analysis based on processing chains of components a natural approach [10]–[12]. Tang et al. [10] analyze response times across processing chains, modeling and improving the behavior of ROS2 executors to optimize task scheduling and system-level response time. Teper et al. [11] evaluate timing performance using two metrics—*maximum reaction time* and *maximum data age*—based on cause-effect chain analysis in ROS2-based autonomous robotic systems. However, prior work [13]–[16] has shown that the DDS-based IPC in ROS2 can introduce millisecond-level delays in data transmission between nodes. In contrast, shared-memory communication in pure C++ implementations incurs significantly lower transmission delays. Moreover, bypassing the ROS2 framework substantially reduces software footprint and debugging complexity. Developers gain finer control over execution paths and avoid the performance overheads and abstraction layers introduced by middleware [17], [18]. Teper et al. [19] show that the ROS2 multithreaded executor is prone to starvation, leading to unbounded response times. These limitations motivate our design independent of the ROS2 framework.

Benchmarking. Bakhshalipour et al. present `RoWild` [20], a comprehensive cross-platform performance benchmark for various mobile robotic systems. It reports end-to-end execution times and identifies algorithmic bottlenecks. `RobotPerf` [21] introduces a benchmarking framework tailored to robotic workloads implemented as ROS2 computational graphs. It supports both black-box and grey-box testing methodologies to evaluate real-time performance across diverse hardware platforms. However, the framework is tightly coupled with the ROS2 ecosystem, limiting its use in non-ROS2 systems. Both studies do not consider the impact of scheduling policies or kernel configurations, nor do they provide a timing model.

Soft real-time scheduling. To address this gap, Sifat et al. [12] propose a safety-performance metric that explicitly incorporates timing considerations in real-time robotic systems. Their approach uses heterogeneous processing units (e.g., CPU and GPU) modeled through a stochastic heterogeneous parallel DAG (SHP-DAG). They evaluate their method using both FIFO and CFS schedulers. However, these schedulers are not designed to meet the hard real-time requirements of safety-critical robotic systems. In contrast, our work explores the use of the real-time scheduling policies and preemptive kernels, which is more suitable for ensuring real-time guarantees.

Evaluate real-time constraints and kernels. Tools such as `cyclictest` [22] and `Timerlat` [23] have been developed to measure execution latency and trace its root causes. The recent tool `LiME` [24] automatically derives task models from real Linux workloads. We previously evaluated the impact of the `PREEMPT_RT` patch using a Raspberry Pi 5 [25], showing its benefits in reducing latency and improving determinism, which motivated further investigation in robotic settings.

III. System Design Overview

Our system is designed to enable cage-free human-robot workspaces by combining GPU-accelerated perception with reactive SSM safety control in a tightly integrated runtime. The hardware includes the following components (Fig. 1A):

- A UR10e [26] industrial robotic arm with an OnRobot VG10 suction gripper [27], connected via LAN to the central computer;
- Five Intel RealSense D435 [28] depth cameras, mounted around the robot Tool Center Point (TCP) on a custom 3D-printed fixture, connected via USB-C (3.2) to the central computer;
- A central computer equipped with an Intel i9-14900KF processor (32 CPUs), an NVIDIA GeForce RTX 4060 Ti (8 GB VRAM), and 2 TB solid-state storage;
- An Atmel ATmega-based Antropo light signaling module [7], [29], connected to the robot's 24 V digital I/O, providing status feedback: safety stop (red), slowed motion (orange), normal operation (green).

The software is implemented in C++ and CUDA, without ROS or external middleware, in order to reduce latency and maintain full control over scheduling, memory allocation, and data exchange. This design also facilitates fine-grained debugging, step-through inspection, and performance monitoring, which are often obscured by middleware like ROS [17]. Figure 2 illustrates the structure of the system software and its data flow, highlighting the timing-critical path from camera acquisition to safety actuation. Below, we describe the key software components of our runtime environment.

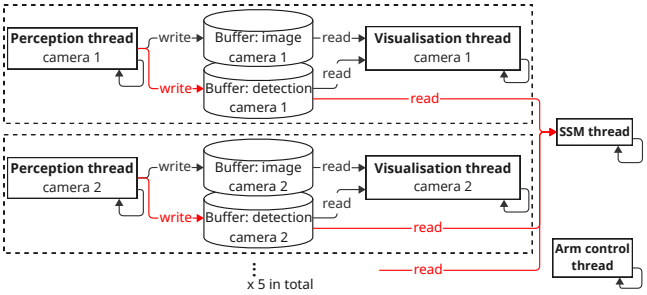


Fig. 2. Task graph of the SENTRYRT-1 runtime. Perception threads process camera input and write detection results to shared buffers, which are consumed by the SSM thread. The SSM and Arm control threads issue commands to the UR robot for motion and signaling. Visualization threads read images and detections to render annotated views. The critical path is shown in red.

Camera acquisition module. Each RealSense camera is polled at 30 FPS to retrieve aligned depth and RGB frames.

Perception module. A neural network detects and segments humans in the RGB frames, then computes their 3D positions by projecting the segmentation masks onto the corresponding depth images. The positions of humans and distances to the robot's Tool Center Point (TCP) are produced and passed to the SSM module for evaluation.

Trajectory control module. The robot executes predefined missions (e.g., picking and placing boxes with the suction gripper), following motion trajectories programmed via the Real-Time Data Exchange (RTDE) protocol [30] provided by Universal Robots, using the UR RTDE library [31].

SSM module. A continuously running safety loop monitors the distance between detected humans and the robot's Tool Center Point (TCP). When a predefined safety threshold is breached, the module dynamically reduces the robot's speed or halts its motion if the separation distance becomes too small. To implement this behavior, SENTRYRT-1 currently uses the RTDE speed slider interface [32][1]. Once the minimum required distance is re-established, the robot resumes its normal operating speed [5].

All components run as C++ `std::threads`, exchanging data via shared memory buffers. Inter-thread communication follows a double-buffering pattern, where one thread writes to a back buffer while another reads from a front buffer. This design enables non-blocking data exchange with minimal locking, reducing synchronization overhead and jitter. Although the SSM loop is the final enforcement point for safety decisions, its effectiveness depends on timely and reliable data flowing through the perception pipeline—including camera acquisition and inference threads that lie on the critical path. This motivates a real-time analysis of the system as a whole.

Why real-time matters. Under high system load, race conditions or scheduling delays in the acquisition threads can lead to stale or empty frames entering the pipeline. From the perspective of the SSM logic, this is functionally equivalent to real-world occlusion: in both cases, the robot loses visibility of its environment. To ensure safety, the system must detect such degraded input and trigger a precautionary stop (i.e., red light), but this requires that the SSM thread itself maintain real-time guarantees. If it too is delayed or starved, these safety violations may go undetected, resulting in unbounded latency or unsafe robot behavior. These observations highlight the need for real-time task modeling and motivate our experiments, which quantify reaction times under stress conditions.

[1]Future work will integrate certified safety interfaces, such as PROFIsafe.

TABLE I
SENTRYRT-1 TASK SET CHARACTERIZATION.

Task	Function	Period	Deadline	Criticality
τ_{ssm}	SSM	2 ms	2 ms	High
τ_{p_i}	Perception (cam. i)	33 ms	33 ms	High
τ_{arm}	Arm mission control	Seconds	N/A	Low
τ_{v_i}	Visualization (cam. i)	Best-effort	N/A	Low

Implementation. The current prototype is structured around a single `main()` function, which launches threads for perception, visualization, arm motion, and safety monitoring. Each perception thread handles both camera acquisition and inference for one sensor (i.e., combining camera acquisition and perception modules), writing its results to shared detection buffers. The SSM thread runs periodically, reading the latest detections from all detection buffers, and issues speed updates to the robot, status signals to the LED module. Visualization threads render annotated RGB-frames to the screen for debugging and user feedback but are not latency-critical.

IV. REAL-TIME MODELING

To evaluate the real-time behavior of our robotic perception and control system, we model the runtime as a set of interacting real-time tasks, each corresponding to a core thread in the implementation. Our goal is to understand how different Linux configurations impact the end-to-end responsiveness of the SSM module—the latency-critical component responsible for enforcing safety constraints. This analysis must also consider upstream dependencies on perception threads (camera acquisition and GPU-accelerated inference), which share compute resources with the SSM loop and may introduce contention under load. Table I summarizes the system's tasks and their timing characteristics, which we detail below.

Perception Tasks (τ_{p_i}). For each camera i, a dedicated thread performs image acquisition and human detection inference. These tasks are periodic, with an intended execution rate of 30 Hz (i.e., period of 33.3 ms). Each perception thread writes to a shared double buffer consumed by the SSM and visualization threads. Inference execution time varies depending on the GPU and frame content, but results must be produced within one frame interval to maintain pipeline stability.

SSM Task (τ_{ssm}). This task runs periodically with a period and deadline of 2 ms (i.e., 500 Hz). It reads the latest detections from all perception buffers, computes human-to-robot distances, and updates the robot's speed and LED signals. This task represents the final safety-critical decision point and must complete execution within each period to ensure timely

intervention in case of human proximity. The 2 ms budget reflects the worst-case latency for effective speed modulation. Since the task polls perception outputs every 2 ms, a new detection may wait up to one period before being processed by SSM thread; combined with a 2 ms execution deadline, this results in a worst-case reaction time of 4 ms. Failure to trigger a safety control input to stop the robot in the provided time constraint significantly increases the risk of injury.

Arm Control Task ($\tau_{\mathbf{arm}}$). This task executes a pre-defined pick-and-place routine via the robot's controller interface. It is not latency-sensitive and is scheduled as best-effort.

Visualization Tasks ($\tau_{\mathbf{v}_i}$). Each perception thread is paired with a UI thread that overlays human detection results on RGB frames and displays annotated output for monitoring and debugging purposes (see Figure 1B). These threads are non-critical and excluded from our real-time evaluation. However, they share buffers with both perception and SSM threads, introducing potential contention in a mixed-criticality setting—a topic we leave for future work.

V. Evaluation

A. Goals and Methodology

Our evaluation focuses on how Linux kernel configurations and scheduler policies affect the real-time behavior of SentryRT-1 under load. We aim to understand the responsiveness and determinism of the perception and SSM loops, which are critical for enforcing human-robot distance constraints. We structure our investigation around the following research questions:

- **RQ1:** How does the system's reaction time degrade under increasing CPU interference?
- **RQ2:** How do different scheduling policies (`CFS`, `RR`, `SCHED_DEADLINE`) impact latency guarantees?
- **RQ3:** What influence does the choice of Linux kernel (`generic`, `lowlatency`, `realtime`) have on worst-case and average latency?

To answer these, we run a series of stress tests on the system while varying scheduler policies and kernel builds. Experiments are repeated across two modes: *Virtual camera mode*, pre-recorded camera streams replayed from disk to simulate identical sensor input, and *Physical camera mode*, live streams from five Intel RealSense cameras attached via USB-C. Each run lasts **30 seconds** and is repeated **3 times** per configuration to capture variability. The experiments process is automated using the **benchkit** open source tool [33].

B. Task Mapping and Scheduling Policies

Each functional module in SentryRT-1 (e.g., perception, safety monitoring) is implemented as a dedicated C++ thread using `std::thread`. We refer to these as *main threads*. However, they rely on several external libraries—such as Intel RealSense, TensorRT, OpenGL, and UR RTDE—which internally spawn additional *subthreads* to handle frame acquisition, inference execution, LED control, and robot actuation.

This architecture introduces a challenge: assigning a scheduling policy to the main thread alone does not guarantee real-time behavior if its subthreads continue to run under the default `SCHED_OTHER` policy, which lacks real-time guarantees. In the presence of CPU interference, these subthreads may be preempted, delaying or even blocking the main thread and breaking end-to-end timing guarantees.

To address this, we implement a mechanism to dynamically identify all threads spawned by each main thread and apply a user-specified fallback scheduling policy to their subthreads. While we cannot assign `SCHED_DEADLINE` to subthreads—since doing so requires explicit knowledge of their execution parameters (e.g., runtime, deadline, period)—we evaluate a fallback option throughout the following combinations.

DL+CFS. Main threads use `SCHED_DEADLINE`; subthreads remain under the default `SCHED_OTHER` policy.

DL+RR. Main threads use `SCHED_DEADLINE`; subthreads are assigned `SCHED_RR` with a fixed static priority of 50.

RR+RR. Both main and subthreads are assigned `SCHED_RR` with fixed priority of 50.

CFS+CFS. Both main and subthreads use the default `SCHED_OTHER` policy. This reflects the unmodified behavior of standard Linux deployments.

This workaround—falling back on the `SCHED_RR` priority class for opaque subthreads—highlights a key limitation of the current Linux real-time scheduling API: assigning a scheduling policy to a parent thread does not automatically propagate to its child threads, necessitating manual intervention to ensure consistent timing behavior across all execution contexts.

For all threads assigned `SCHED_DEADLINE`, the `runtime`, `deadline`, and `period` parameters are configured based on the expected execution rates and constraints of each task, as summarized in Table I. Subthread priorities are equal within each configuration, and thread pinning is not enforced, allowing the Linux scheduler to dynamically manage core assignment. The arm mission control and visualization tasks are considered non-critical and remain scheduled under the default `SCHED_OTHER` policy.

To simulate non-critical interfering workloads—such as OS background tasks or network stack activity—we launch a configurable number of *noise threads* (from 0 up to 128, to test the limit of the system under high load) using the default `SCHED_OTHER` policy. Each thread executes a tight loop of relaxed atomic increments and decrements on a dummy counter, designed to saturate CPU pipelines. On our system with 32 logical CPUs, launching more than 32 such threads guarantees oversubscription and exposes the impact of CPU interference on real-time tasks.

C. Kernel Variants

In addition to scheduling policy, the kernel configuration plays a role in determining the responsiveness and latency behavior of real-time applications. We compare three Linux kernel variants provided by Ubuntu 22.04 [34], all based on version `5.15.0`:

- **`generic`** (`5.15.0-138-generic`): The standard Ubuntu kernel with voluntary preemption. It is optimized

TABLE II
EXPERIMENTAL VARIABLES AND THEIR EXPLORED VALUES.

Variable	Values
Camera mode	`Physical` (D435i) / `Virtual` (dataset)
Linux kernel	`generic` / `lowlatency` / `realtime`
Main threads policy	`SCHED_DEADLINE` (DL) / `SCHED_RR` (RR) / `SCHED_OTHER` (CFS)
Subthreads policy	`SCHED_RR` (RR) / `SCHED_OTHER` (CFS)
Noise thread count	0 / 16 / 32 / 64 / 128

for throughput and general-purpose workloads, but lacks guarantees on worst-case latency.

- **`lowlatency`** (`5.15.0-138-lowlatency`): A soft real-time kernel variant enabling more aggressive preemption to reduce interrupt handling latency and jitter.
- **`realtime`** (`5.15.0-1082-realtime`): A more preemptible real-time kernel distributed via Ubuntu Pro. It includes the `PREEMPT_RT` patchset, which converts most interrupt handlers into schedulable threads and supports bounded latencies.

D. Metric Collected

To ensure the robot does not collide with humans, the primary metric evaluated is the *reaction time* of the system's timing-critical path. We define *reaction time* as the duration between the arrival of a new input frame and the completion of the **first job** of the SSM task τ_{ssm} that processes this frame. Since τ_{ssm} runs at a significantly higher frequency than the perception tasks τ_{p_i}, the system's reaction to an input frame is effectively determined by this first polling instance of τ_{ssm} that corresponds to that frame; later instances for the same frame only maintain the current safety status until a new frame is available. This definition allows us to isolate the latency of the system's critical path—from sensor input to safety actuation—and to measure how kernel and scheduling decisions affect timely responsiveness under varying interference levels.

We evaluate both the **worst-case** and **average-case** reaction times across all frames in each run. Each experimental configuration is repeated 3 times, and we report the aggregated statistics (max and mean across repetitions) to capture variability and ensure repeatability. Table II summarizes the experimental variables and their explored values.

E. Results and Discussion

RQ1: Reaction time under interference. We begin by analyzing how the system's reaction time is affected by increasing CPU load, using configurations that vary the number of noise threads. Figure 3 shows that under the default `CFS+CFS` policy, both average and worst-case reaction times degrade rapidly once the number of noise threads exceeds the number of logical CPUs (32 on our platform). This is expected: critical tasks such as τ_{p_i} and τ_{ssm} receive no prioritization and must contend equally for CPU time.

In contrast, policies that assign real-time priorities to these threads (e.g., `RR+RR`, `DL+RR`, `DL+CFS`) remain resilient even under high contention. This confirms that prioritization—especially for perception and safety-critical threads—is essential to maintain bounded latency.

RQ2: Impact of scheduling policy. Across all interference levels, both `DL+RR` and `RR+RR` configurations show significant improvements in worst-case and average-case reaction time compared to `CFS+CFS`. These two policies are largely on par in our experiments, maintaining stable latency under load and shielding the critical path from CPU interference. The lack of a clear performance gap between `DL+RR` and `RR+RR` is explained by the system's ample resources: with 32 logical CPUs and relatively low per-task utilization, each real-time thread can be effectively isolated. In more constrained systems, where real-time tasks must share cores or operate closer to full CPU utilization, we expect the benefits of EDF-based scheduling (e.g., deadline enforcement and bandwidth guarantees in `SCHED_DEADLINE`) to become more pronounced.

We observe outliers even under real-time configurations. For example, under the `DL+RR` policy on the `generic` kernel with 0 noise thread and virtual cameras, the worst-case reaction time spikes to 81 ms. This is attributable to the unpredictability of GPU inference workloads, which in this case consumed up to 77 ms—far beyond the nominal 33 ms perception period. This underscores a key limitation: while the CPU scheduling is protected, tasks offloaded to the GPU may still introduce non-determinism.

RQ3: Kernel comparison and jitter. To examine the impact of kernel variants on timing variability, we zoom in on individual experimental runs. Figure 4 shows per-frame reaction times for a single run of each kernel configuration under fixed load (128 noise threads) and constant scheduling policy (DL+RR). This fine-grained view shows that the `realtime` kernel (with `PREEMPT_RT`) provides lower jitter and tighter latency bounds than both the `generic` and `lowlatency` kernels. However, its impact is smaller than that of the scheduling policy itself. This suggests that most benefits stem from correct prioritization and task isolation rather than kernel-level preemption improvements alone.

Highlighted setting. Under a top-performing setup—real camera input, high interference (128 noise threads), and `DL+RR` scheduling on a `realtime` kernel—the system reacts within **44-70 ms worst-case** and **15-17 ms average**. The default (`CFS+CFS`, `generic` kernel) shows 353 ms worst-case and 117 ms average. This yields about 80% and 85% reductions in worst-case and average times, respectively.

Summary of findings. These results support the conclusion that real-time Linux configurations—with explicit prioritization of safety-critical tasks and runtime visibility into subthreads—can significantly improve latency determinism in robotics workloads. Yet challenges remain: in particular, GPU tasks such as human detection are opaque to the scheduler and can lead to deadline violations even without CPU interference. This motivates further work in designing GPU-aware scheduling models, group-based deadline policies, or extending Linux with budget inheritance mechanisms across threads and heterogeneous resources.

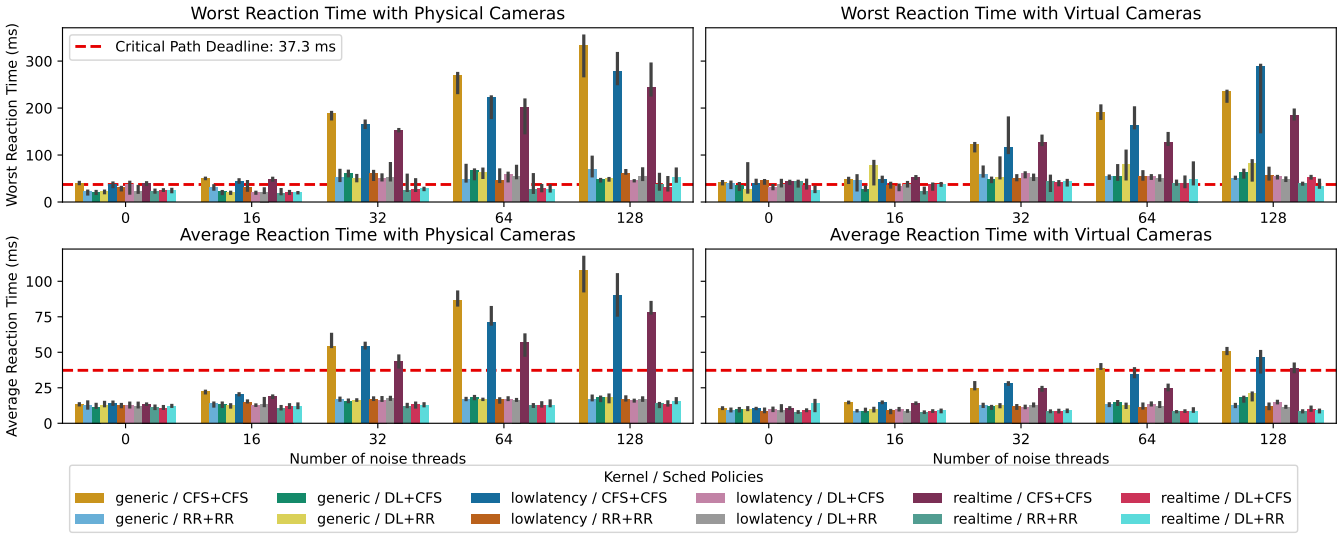


Fig. 3. Reaction times using physical and virtual cameras under different scheduling policies on various Linux kernels. Each 30-second experiment collects all reaction times (worst and average) per configuration. We repeat each experiment 3 times; bars represent the median value across the 3 runs, with error bars showing the minimum and maximum. Using the critical path deadline (37.3 ms) as a reference, the DL+RR policy with physical cameras meets the deadline as long as the number of noise threads does not exceed 32. Average reaction times for RR+RR, DL+CFS, and DL+RR also remain below this threshold. Real-time scheduling policies yield significantly lower reaction times than the default CFS+CFS policy under system interference.

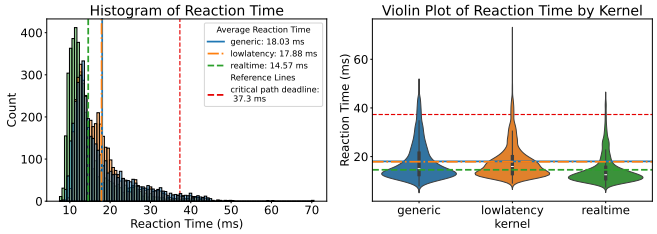


Fig. 4. Sample distribution of system reaction times on `generic`, `lowlatency`, and `realtime` Linux kernels. Each distribution corresponds to a single 30-second run using physical cameras, the DL+RR policy, and 128 noise threads. While the average reaction times on the `generic` and `lowlatency` kernels are nearly identical, the `realtime` kernel reduces the average by 3 ms. Moreover, the reaction time distribution under `realtime` shifts leftward, with a higher concentration of samples below the critical path deadline. Despite this improvement, all kernels experience deadline misses, primarily due to the unpredictable execution time of GPU tasks.

These experiments support the design direction of SentryRT-1, which aims to serve as a lightweight safety perception layer for robotics. An runtime capable of supporting real-time embedded systems with GPU acceleration and modular real-time scheduling is a key requirement for deployments.

VI. Conclusion

This paper presented SentryRT-1, a minimal C++/CUDA runtime that integrates multi-camera sensing and GPU-accelerated human detection for safety-critical robot control without middleware. By modeling the system as a real-time task set and evaluating its responsiveness under various Linux kernel and scheduler configurations, we demonstrated that proper use of `SCHED_DEADLINE` combined with a real-time kernel (with `PREEMPT_RT`) can significantly improve both average and worst-case reaction times. Compared to the default setup (`CFS` scheduler, `generic` kernel), our configuration reduced worst-case reaction time by **80%** and average time by **85%**, even under high CPU interference. This underscores the viability of Linux-based systems for enforcing Speed and Separation Monitoring (SSM) constraints.

To further improve performance under high interference, we plan to use the `timerlat` [23] tool to analyze deadline miss conditions, and use the `LiME` [24] tool to further refine the task model. Our results expose limitations in the current Linux real-time scheduling API, especially its inability to manage multi-threaded real-time tasks as unified entities. Ensuring consistent scheduling across main threads and library-spawned subthreads is a manual and error-prone process. This motivates the need for future work on group-level scheduling, graceful `SCHED_DEADLINE` inheritance, and deeper integration of scheduling policies with real-world runtime dependencies.

Beyond scheduler design, several other factors warrant investigation: the impact of thread pinning, I/O and memory contention, GPU sharing between critical and non-critical tasks, and interference from network traffic or unpredictable thread placement. Newer kernel features such as EEVDF scheduling, or alternative platforms including embedded SoCs like Jetson or heterogeneous CPU architectures with Performance- and Efficient-cores (e.g., Intel Alder Lake), offer further opportunities to test and refine our assumptions. Finally, extending SentryRT-1 to support heterogeneous sensor fusion, both rule-based and AI-driven scene interpretation, and modular perception pipelines will help evaluate its applicability in increasingly complex collaborative robotic scenarios.

ACKNOWLEDGMENT

This work was supported by imec through the SAFEBOT research program. The authors also acknowledge Universal Robots for their in-kind contribution of hardware components, which facilitated the system integration and evaluation.

REFERENCES

[1] W. Dauth and S. Findeisen and Jens Suedekum and Nicole Woessner, "The Adjustment of Labor Markets to Robots," *Journal of the European Economic Association*, 2021.

[2] Shahab Sharfaei and Jan Bittner, "Technological employment: Evidence from worldwide robot adoption," *Technological Forecasting and Social Change*, 2024.

[3] M. Vasic and A. Billard, "Safety issues in human-robot interactions," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 197–204.

[4] T.-A. Tran, T. Ruppert, G. Eigner, and J. Abonyi, "Retrofitting-based development of brownfield industry 4.0 and industry 5.0 solutions," *IEEE Access*, vol. 10, pp. 64 348–64 374, 2022.

[5] International Organization for Standardization, *Robotics — Safety requirements — Part 2: Industrial robot applications and robot cells*, International Organization for Standardization Std. ISO 10 218-2:2025, 2025, second edition, published February 2025. [Online]. Available: https://www.iso.org/standard/73934.html

[6] C. Scholz, H.-L. Cao, E. Imrith, N. Roshandel, H. Firouzipouyaei, A. Burkiewicz, M. Amighi, S. Menet, D. W. Sisavath, A. Paolillo, X. Rottenberg, P. Gerets, D. Cheyns, M. Dahlem, I. Ocket, J. Genoe, K. Philips, B. Stoffelen, J. Van den Bosch, S. Latre, and B. Vanderborght, "Sensor-Enabled Safety Systems for Human–Robot Collaboration: A Review," *IEEE Sensors Journal*, vol. 25, no. 1, pp. 65–88, 2025.

[7] C. Scholz, H.-L. Cao, I. El Makrini, and B. Vanderborght, "Antropo: An open-source platform to increase the anthropomorphism of the Franka Emika collaborative robot arm," *Plos one*, vol. 18, no. 10, p. e0292078, 2023.

[8] "Offline Simulator - e-Series and UR20/UR30 - UR Sim for Linux," https://perma.cc/8DVY-6RG9, accessed: 2025-05-01.

[9] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

[10] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 231–243.

[11] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-To-End Timing Analysis in ROS2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 53–65.

[12] A. H. Sifat, X. Deng, B. Bharmal, S. Wang, S. Huang, J. Huang, C. Jung, H. Zeng, and R. Williams, "A Safety-Performance Metric Enabling Computational Awareness in Autonomous Robots," *IEEE Robotics and Automation Letters*, vol. 8, no. 9, pp. 5727–5734, 2023.

[13] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1–10.

[14] Y. Ye, Z. Nie, X. Liu, F. Xie, Z. Li, and P. Li, "ROS2 Real-time Performance Optimization and Evaluation," *Chinese Journal of Mechanical Engineering*, vol. 36, no. 1, p. 144, 2023. [Online]. Available: https://doi.org/10.1186/s10033-023-00976-5

[15] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 264–277.

[16] H. Abaza, D. Roy, B. Trach, W. Chang, S. Saidi, A. Motakis, W. Ren, and Y. Liu, "Managing End-to-End Timing Jitters in ROS2 Computation Chains," in *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*, ser. RTNS '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 229–241. [Online]. Available: https://doi.org/10.1145/3696355.3696363

[17] S. Barut, M. Boneberger, P. Mohammadi, and J. J. Steil, "Benchmarking Real-Time Capabilities of ROS 2 and OROCOS for Robotics Applications," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 708–714.

[18] A. Alhonen, "Why don't we use ROS?" http://pulurobotics.fi/blog/pulurobotics-blog-1/post/why-don-t-we-use-ros-7, 2017, pulurobotics Blog.

[19] H. Teper, D. Kuhse, M. Günzel, G. v. d. Brüggen, F. Howar, and J.-J. Chen, "Thread Carefully: Preventing Starvation in the ROS 2 Multithreaded Executor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3588–3599, 2024.

[20] M. Bakhshalipour and P. B. Gibbons, "Agents of Autonomy: A Systematic Study of Robotics on Modern Hardware," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 3, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3626774

[21] V. Mayoral-Vilches, J. Jabbour, Y.-S. Hsiao, Z. Wan, M. Crespo-Álvarez, M. Stewart, J. M. Reina-Muñoz, P. Nagras, G. Vikhe, M. Bakhshalipour, M. Pinzger, S. Rass, S. Panigrahi, G. Corradi, N. Roy, P. B. Gibbons, S. M. Neuman, B. Plancher, and V. J. Reddi, "RobotPerf: An Open-Source, Vendor-Agnostic, Benchmarking Suite for Evaluating Robotics Computing System Performance," 2024. [Online]. Available: https://arxiv.org/abs/2309.09212

[22] L. T. Foundation, "Linux Foundation Realtime Wiki - HowTo - Cyclictest," https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start, accessed: 2025-05-01.

[23] D. B. D. Oliveira, D. Casini, J. Lelli, and T. Cucinotta, "Timerlat: Real-time Linux Scheduling Latency Measurements, Tracing, and Analysis," *IEEE Transactions on Computers*, pp. 1–13, 2025.

[24] Björn B. Brandenburg and Cédric Courtaud and Filip Marković and Bite Ye, "LiME: The Linux Real-Time Task Model Extractor," in *2025 IEEE 31th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025.

[25] W. Dewit, A. Paolillo, and J. Goossens, "A Preliminary Assessment of the real-time capabilities of Real-Time Linux on Raspberry Pi 5," in *Proceedings of the 18th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2024, pp. 7–12. [Online]. Available: https://www.ecrts.org/wp-content/uploads/2024/07/ospert24-proceedings.pdf

[26] "UR10e, Medium-sized, versatile cobot," https://perma.cc/4PCV-C9WL, Accessed: 2025-05-01.

[27] "VG10 – Flexibler, Einstellbarer Vakuumgreifer," https://perma.cc/US4B-ZLDK, Accessed: 2025-05-01.

[28] "Intel RealSense D435," https://perma.cc/K5EN-7WHV, Accessed: 2025-05-01.

[29] H.-L. Cao, S. A. Elprama, C. Scholz, P. Siahaya, I. El Makrini, A. Jacobs, A. Ajoudani, and B. Vanderborght, "Designing interaction interface for supportive human-robot collaboration: A co-creation study involving factory employees," *Computers & Industrial Engineering*, vol. 192, p. 110208, 2024.

[30] "Real-Time Data Exchange (RTDE) Guide," https://perma.cc/6VCE-9KD6, Accessed: 2025-05-01.

[31] A. P. Lindvig, I. Iturrate, U. Kindler, and C. Sloth, "ur_rtde: An Interface for Controlling Universal Robots (UR) using the Real-Time Data Exchange (RTDE)," in *2025 IEEE/SICE International Symposium on System Integration (SII)*, 2025, pp. 1118–1123.

[32] "RTDE IO Interface API," https://sdurobotics.gitlab.io/ur_rtde/api/api.html#rtde-io-api, accessed: 2025-05-01.

[33] "Benchkit: Performance Evaluation Framework," https://github.com/open-s4c/benchkit/tree/main, accessed: 2025-05-01.

[34] "Ubuntu Kernel Database," https://kernel.ubuntu.com/mainline/, accessed: 2025-05-01.

IRx: RTOS-Aware Abstract Interpretation using an LLVM-based Interpreter

Andreas Kässens, Vitali Fendel, Daniel Lohmann
Leibniz Universität Hannover
Hannover, Germany
{kaessens, lohmann}@sra.uni-hannover.de, vitali.fendel@gmail.com

***Abstract*—By tailoring *Real-Time Operating Systems (RTOSs)* to the specific real-time application using abstract interpretation, one can significantly improve nonfunctional properties, such as dependability or slack time. To analyze the interactions with OS objects and lift this optimization potential, the system call semantics of the RTOS must be derived, either from a specification or from the RTOS implementation, which in both cases is a tedious process. Previous work has demonstrated this for OSEK/AUTOSAR systems.**

To allow the abstract interpretation of applications for *arbitrary* RTOSs, we suggest analyzing system calls with a concrete LLVM-based interpreter: IRx. Compared to previous approaches, it is not necessary to reimplement the RTOS interface as an abstract OS model which can be tedious and error-prone for configurable and rapidly evolving RTOSs like Zephyr. Instead, we extract the effect of the system call from the resulting concrete kernel state of the *actual execution* of system calls. We evaluate the applicability of IRx using abstract interpretation of benchmark applications and a complex embedded firmware application.

I. Introduction

Embedded real-time systems typically serve a dedicated and highly specialized purpose. To meet the specific requirements of such applications, many embedded operating systems with different focuses have been designed in the past. For critical applications in automotive or avionics industry, constraints regarding deterministic behavior and timing predictability with respect to deadlines demand using an *Real-Time Operating System (RTOS)*, usually with fixed-priority-based scheduling and static allocation of system objects [1]. For such real-time applications, a control-flow–sensitive abstract interpretation can extract deep static knowledge about the system interactions at runtime. Previous research on static analysis and optimization of such statically configured systems is often limited to OSEK/AUTOSAR applications [2], [3], [4] as it requires an abstract OS model, which is difficult to derive for less strictly specified RTOSs.

For example, the *System State Enumeration (SSE)* in *d*OSEK [3] and the MultiSSE in *Automatic Real-Time System Analyzer (ARA)* [4] use a `SystemSemantics` function that interprets the system call against the abstract OS model. The result of the analysis, the *State Transition Graph (STG)*, contains all possible system states, including the properties of all system objects, such as resource allocation or event states.

This work was partly supported by the German Research Foundation (DFG) under grant no. LO 1719/4-1

```
1  struct sensors s;
2  void thread_flight_stabi() { // PRIO_HIGH
3    while(true) {
4      set_actuators(s);
5      k_sleep(K_MSEC(5));
6    }
7  }
8  void thread_read_sensors() { // PRIO_MED
9    while (true) {
10     s = read_sensors();
11     if threshold(s)
12       k_wakeup(flight_stabi);
13     k_sleep(K_MSEC(1)); // low prio tasks
14   }
15 }
```

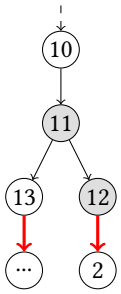


Fig. 1: Simplified GCFG of a Zephyr application: A thread reads sensor values (#10) and wakes a high-priority task (#12) ahead of schedule when a threshold is exceeded.

By transforming the state graph into a *Global Control-Flow Graph (GCFG)*, cross-kernel optimizations such as per-call specialization of system interactions, assertions for improved dependability, or inlining of complete tasks are possible [3].

Drawing from our experience in developing abstract OS models within the static analysis framework ARA [5], the process of adding support for control-flow–sensitive abstract interpretation is tedious, especially for more complex and evolving RTOS interfaces, such as Zephyr. Nevertheless, also these more complex (and, compared to AUTOSAR, less static) systems would profit from RTOS-aware analyses and optimizations: The GCFG in Fig. 1 describes a small Zephyr application. Next to local transitions (black), it contains global cross-kernel transitions (red), which could be optimized considering the RTOS semantic. The `k_wakeup` system call can be replaced with a direct dispatch to the high-priority task. If there is no further higher-priority thread in the system, the whole-system optimizer could even inline the respective parts of the thread at the callsite.

This Paper

We describe a new approach for bringing such analyses and optimizations to a broader practicability by supporting (almost) *arbitrary* RTOS, for which besides the syscall interface no formal specification or semantic model exists. The core idea is to automatically derive the semantic model with respect to the *actual application* from the syscall interface and the RTOS *implementation*, by directly executing the RTOS system calls using an LLVM IR interpreter. To show the viability of

our approach, we have implemented it for the (comparatively complex) Zephyr RTOS into the ARA analysis framework [5],[1] and run the SSE on several benchmark applications and a real-world example. In particular, we claim the following contributions:

- We describe the abstract interpretation of cross-kernel control flows using IRx, an LLVM-based interpreter.
- Using this approach, we implement the SSE for Zephyr.
- Finally, we apply the SSE to benchmark applications and a real-world Zephyr application.

The remainder of this paper is structured as follows: We describe the background of the control-flow–sensitive analysis in Sec. II. Sec. III covers the fundamental approach compared to previous work and Sec. IV explains the implementation details. We evaluate IRx in Sec. V and discuss limitations and future work in Sec. VI.

II. Background

To gain compile-time knowledge about the embedded real-time system, as shown in Fig. 1, it is necessary to apply static analyses such as the abstract interpretation [6] to the application. By traversing the *Control-Flow Graph (CFG)* of the application, the effects of the application and system call logic on an abstract system state can be computed.

As an instance of this abstract interpretation, the SSE [3] and the MultiSSE [4] require the `SystemSemantics` function implemented by an abstract OS model, which computes the subsequent *Abstract System States (AbSSs)* of the current system state at the granularity of *Atomic Basic Blocks (ABBs)* [2]. These abstract OS models including all system calls must be derived manually — either based on an explicit specification in the case of OSEK/AUTOSAR or based on the *implementation* of the RTOS. As a result, all possible AbSS are connected in a STG, which enables the creation of a GCFG [3] as shown in Fig. 1. Next to the optimizations described in the following section, it is possible to use such graphs to compute tight *Worst-Case Execution Time (WCET)* bounds [7].

A. Whole-System Optimization

Static analysis of embedded real-time control systems does not, by itself, lead to improvements in non-functional properties. One of the benefits is the ability to perform system-wide optimizations based on the knowledge gained from the interactions between scheduled units within the application. Using control-flow–sensitive abstract interpretation, all possible subsequent states can be computed in advance for each interaction between the application and the RTOS (i.e., system call). This allows the synthesis of specialized system call *implementations*, which may include assertions to improve dependability [3], reduce system call overhead by replacing the generic implementation with a fast path [4], or allow inlining of complete tasks. By tailoring the RTOS to the application, it is possible to trade unnecessary functionality and flexibility of the RTOS for higher predictability and dependability, faster execution with less kernel overhead, and less pessimism in the timing calculation.

[1]All code will be published at https://github.com/luhsra/parrot

B. Interrupts and state explosion

In real-time control systems, tasks can be executed according to a schedule (time-triggered) or based on interrupts (event-triggered) [1], [2]. Without expert knowledge of execution timing or interrupt inter-arrival times, an abstract interpretation like the SSE must emit additional states for each interrupt source at every possible system state, leading to an explosion of possible states. For hard real-time systems, inter-arrival times are required for schedulability analysis [8] and are often known by application developers in practice [9]. This information can be taken into account during the construction of the STG, which can drastically reduce the number of possible states [4]. The example application in Fig. 1 requires timer interrupts, which leads to state explosion if not considered during STG creation. The original implementation of the SSE uses an additional notation of TaskGroups and suggests incorporating explicit cause-effect knowledge to further reduce the number of unwanted states [3].

C. The Zephyr RTOS

Zephyr is a thread-based open source RTOS that supports many architectures and target platforms [10]. It is supported by the Linux Foundation and shares some similarities with the Linux kernel [11], such as the development process, monolithic system design, configurability, and tooling. KConfig allows application developers to configure the scheduling strategy, supported device drivers, system interfaces, etc. In Zephyr, system objects can be created statically or dynamically, but static allocation and initialization is preferable for embedded applications [12].

In recent years, the Zephyr RTOS has gained traction in the open source world and with industrial stakeholders [10]. Zephyr is also an increasingly popular target for evaluation in research [11], [13].

III. The IRx Approach

In Fig. 2 we show conceptually how IRx is integrated into the control-flow–sensitive abstract interpretation of the ARA framework for Zephyr. While traversing the ABBs in the application's local CFG, all system calls need to be interpreted. The list of system calls is determined beforehand, but could be extracted automatically from the *parse_syscalls.py* build step in Zephyr. The entire RTOS kernel is precompiled to LLVM IR code and therefore includes the compile-time application-specific configuration. Next to the system call to be interpreted, static analysis must consider the current abstract state in the IRx context. By the nature of control-flow–sensitive analysis, states are not ordered by their chronological order, but by the order imposed by the search algorithm. Therefore, the interpreter must be *prepared* for each system call. First, it is reset to a predefined state (i.e., all global objects are initialized, like a fresh boot), and then the correct state of the abstract system objects is replicated in the concrete kernel

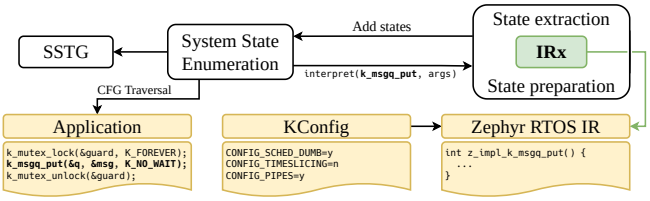


Fig. 2: Overview of the SSE with IRx: While traversing the application's CFG, IRx interprets the system calls using the precompiled Zephyr IR.

state. In the example from Fig. 2, in particular the scheduler, message queue (`msgq`) and mutex states need to be prepared. To return after the system call is finished, IRx can stop on a predefined function call. For Zephyr, this termination point is the `arch_swap` function call, which dispatches to the next thread at the end of the system call. After the system call interpretation, the RTOS-specific adapter extracts for example the message queue and scheduler ready-list from the concrete kernel state and updates the abstract system state.

IV. Implementation

IRx is based on the upstream LLVM-14 *Intermediate Representation (IR)* interpreter[2]. To support the most common embedded realtime systems, we have extended it to allow executing 32-bit IR code. This requires changes in memory management, global variable initialization, and load/store instruction callbacks. However, only a one-line modification to one LLVM header file is sufficient to enable our custom interpreter implementation, allowing the use of standard, unmodified LLVM libraries and eliminating the need for a custom LLVM build. The ARA framework is fundamentally based on LLVM IR, so using the LLVM interpreter allows seamless integration between analysis, optimization, and synthesis, as the same IR code is used throughout the entire framework.

A. Initialization and Memory Mapping

To enable 32-bit RTOS execution within the 64-bit interpreter, we modified global variable initialization and load/store instructions to intercept memory accesses and use a memory map to translate between the LLVM-allocated memory space and the RTOS application's memory space.

Initialization of global variables posed a challenge due to circular dependencies where variable addresses were referenced in the initialization of other variables. In the upstream interpreter, these initializers are executed during construction, but in IRx they required delaying execution until the memory map is fully established to ensure correct address references. After that, initialization can proceed with the correct 32-bit address space setup, allowing referencing the 32-bit addresses by the initializers of the global variables. We expect that the addition of native 32-bit interpretation within LLVM would eventually eliminate the need for these modifications. Since the LLVM IR does not contain definitions for external symbols provided by the linker script, we generate the missing and necessary symbols based on the linker script specifications after IRx loads the IR code.

[2]https://github.com/llvm/llvm-project/tree/llvmorg-14.0.6/llvm/lib/ExecutionEngine/Interpreter

B. Hardware specifics

Compiled for a specific target system (e.g., Intel ECFW in Sec. V-B targeting ARM), the RTOS contains hardware-specific assembly code that the LLVM IR interpreter cannot execute directly. However, further execution is not necessary, as we can already extract sufficient information regarding its scheduling decision and provide the information to the static analysis. The hardware implementation of the context switch is never reached within the interpreter, as we automatically stop the interpreter at predefined function calls, specifically `arch_swap` for Zephyr. Instead, the RTOS adapter uses the extracted scheduling decision to update the running thread of the abstract system state, effectively executing the context switch within our RTOS adapter and recording the decision for later analysis. To enable precise interpreter flow control and isolation of hardware-specific parts at predefined locations, the RTOS's IR code is compiled without optimization to prevent instruction reordering and inlining. Importantly, this does not prevent enabling compiler optimization flags for the final generated binary after the static analysis.

The ARA framework triggers interrupts within its SSE implementation. The corresponding *Interrupt-Service-Routines (ISRs)* are then executed in the same way as system calls are executed by IRx. User-defined callback routines can be called by the ISR, which are stored as function pointer addresses within the kernel objects. The RTOS adapter translates these function pointers into LLVM IR function names to allow extraction of the subsequent AbSS.

C. Kernel state preparation and extraction

Using a separate tool, we automatically extract relevant data structures from the compiled LLVM IR RTOS code to be accessible for the RTOS adapter. Combined with struct (*SType*) and debug information (*DIType*), the data structures ensure accurate type detection, querying of the struct member names, and calculating the address offsets for conversion to and from Python.

We manage concrete kernel objects through a proxy that tracks their address. Each object's Python object ID is mapped to its 32-bit interpreter address, and the Python representation of that object holds the corresponding 32-bit memory address. Additionally, we maintain a map to cache the Python objects for subsequent requests, and to ensure that the corresponding concrete kernel data has a unique Python object representation.

For each system call, the IRx interpreter prepares a context as described in Sec. III and shown in Fig. 2. For RTOS-specific initialization of concrete kernel state, the Zephyr adapter uses the existing RTOS initialization functions to properly prepare the statically and dynamically allocated

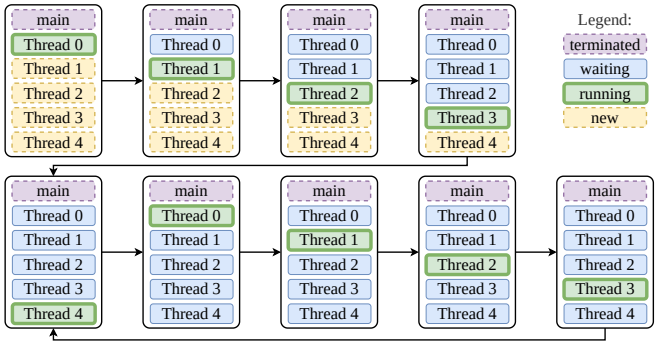


Fig. 3: Simplified STG for the Thread Metric Cooperative Scheduling benchmark: Five threads with same priority yield the CPU, resulting in round-robin scheduling. All other system calls before thread 0 are omitted.

Benchmark/Application	#AbSSs	#Transitions	Runtime [min]
Basic processing	18	27	
Memory allocation	23	34	
Synchronization	23	34	
Message passing	26	39	
Preemptive interrupts	49	72	
Interrupt processing	52	77	
Preemptive scheduling	73	113	
Cooperative scheduling	98	148	
ECFW, ISR A	82	127	0.5
ECFW, ISR A – B	276	428	1.5
ECFW, ISR A – C	1 995	3 088	8.2
ECFW, ISR A – D	19 833	30 634	72.9
ECFW, ISR A – E	234 506	361 634	824.8

TABLE I: Statistics of SSE analysis on the Thread Metric benchmark suite and Intel Embedded Controller Firmware

specific concrete kernel structures such as the scheduler ready-list. The remaining data is then restored from the abstract system state into the interpreter's memory to reassemble the concrete native kernel state. Finally, after execution, this mapping allows the retrieval of all corresponding and possibly modified data to update the abstract system state.

V. Evaluation

To evaluate the IRx implementation and the state extraction in ARA, we run the SSE on several benchmark applications as well as on a large real-world example. In addition to some small hand-crafted examples, we have ported the Thread Metric RTOS benchmark suite [14] to the current Zephyr LTS (3.7.0)[3].

A. Thread Metric Benchmark Suite

This benchmark suite was originally designed for the ThreadX RTOS to compare the performance of core RTOS functionality such as scheduling and interrupt handling [13]. Since we are only interested in RTOS semantics in the benchmark, we disable the result collection thread that wakes up every 30 seconds to summarize the results at specified intervals. For example, the Cooperative Scheduling benchmark consists of five threads with the same priority. After the *System Setup Point (SSP)* is reached (i.e., all threads are initialized), they are scheduled in a round-robin fashion. Using IRx to interpret system calls in ARA, we perform the SSE on these benchmark applications. We manually confirmed that the resulting STGs reflect the expected system behavior, as exemplified in Fig. 3 for the Cooperative Scheduling benchmark.

Tab. I collects the STG statistics for each test application. The algorithm runtimes for the Thread Metric benchmarks are in the range of seconds. Since the scheduling tests have the highest number of threads, more system calls occur and the number of states increases. The Thread Metric interrupt processing benchmarks use *Software-Generated Interrupts (SGIs)* to trigger the interrupt callback function. In our Zephyr adapter implementation, we map the SGIs to normal external interrupts, which are only enabled for a single ABB when they are triggered. As external interrupts are non-deterministic by nature, the SSE emits two states — either the interrupt was triggered or not. For the Thread Metric benchmarks, where we can ensure that SGIs is always triggered immediately, this handling leads to an overestimation of possible system states. On the other hand, real systems could disable such interrupts, so we cannot model these SGIs as dispatch to the interrupt handling thread.

[3]The latest non-LTS (4.1) now includes an official port as well.

B. Intel Embedded Controller Firmware

To evaluate the scalability of our implementation, we took the largest open source Zephyr application we could find, the Intel *Embedded Controller Firmware (ECFW)* [12]. The firmware can control low-level functionality of Intel notebooks and is targeted at Zephyr 3.4. After porting the ECFW to Zephyr 3.7 LTS, we disable the driver logic to limit the amount of control-flow traversal. In particular, some drivers, such as I^2C devices, use system mutexes and sleep system calls. Since these system calls in a deep call stack hierarchy lead to state explosion [3], we ignore them and assume those low-level interactions to be atomic related to the system state and the rest of the application logic. For real-time scenarios, Zephyr provides the application with the RTIO peripheral API, which can be used instead of blocking calls.

The application contains up to eight configurable threads for power sequencing, peripheral, system, keyboard, and POST management. These threads interact with six ISRs via message queues and semaphores as shown in Fig. 4.

Since the application has no hard real-time constraints [12] and no timing information is available, we need to reduce the number of possible interrupts in the application, otherwise the SSE will not terminate due to the state explosion described in Sec. II-B. Therefore, we disable interrupts until the SSP is reached, which in this case means that all threads are initialized and block until the corresponding interrupt is triggered. Next, as this event-triggered system does not have

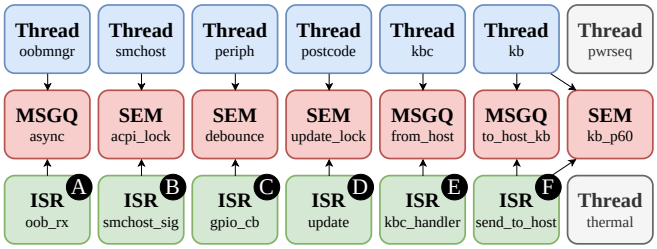


Fig. 4: System objects in Intel ECFW: 6 Threads and ISRs are analyzed with IRx. Two time-triggered threads are deactivated. Semaphores and message queues are used for communication.

a defined hyperperiod and interrupt inter-arrival times, we introduce an artificial ISR activation limit, otherwise the SSE cannot terminate because a new interrupt can occur at any time. For a system with N threads and M interrupt sources, we define the hyperperiod as the time from the SSP to just before any interrupt is triggered a second time. The resulting system states in the hyperperiod then include all possible activation pattern permutations of $0 \leq m \leq M$ interrupt sources. We run the SSE analysis using IRx with $N = 6$ threads and $1 \leq M \leq 5$ ISRs enabled in the ECFW, named from Ⓐ to Ⓔ in Fig. 4. The two time-triggered threads (i.e., repeatedly sleeping in a loop) are disabled because they lead to the same state explosion as external interrupts when no timing information is considered. The Tab. I shows the STG results for the Intel ECFW with increasing number of ISRs. With each additional interrupt, the number of states and transitions — which directly affect the runtime — increases exponentially. With interrupt Ⓕ, the SSE analysis would take about an order of magnitude longer and was not performed because it does not provide any knowledge gain.

VI. Discussion

In the evaluation, we have shown that an LLVM-based interpreter can replace an abstract OS model for static control-flow–sensitive RTOS interaction analysis. Compared to previous work, we do not need to create an RTOS model with system semantics for each system call from the specification or implementation of the RTOS interface. With IRx, we *execute* the RTOS implementation and use an adapter to prepare the concrete kernel state and extract only the state needed for the flow-sensitive abstract interpretation.

A. Scalability and Performance

Although there is some overhead in preparing the interpreter and extracting the kernel state, the scalability issues discussed above arise from the SSE analysis, not from the IRx approach. In addition, the ARA framework for CFG traversal uses Python, making the entire analysis single-threaded. Currently, we are already automatically analyzing smaller applications using IRx in our CI/CD environment [5]. For more complex applications, we plan to use timing information to reduce the state graph, and also to remove the Python GIL to parallelize the CFG traversal and RTOS state analysis. Furthermore, we could partially and iteratively implement a specialized and faster abstract OS model from the results of the initial analysis with IRx, continuously monitoring potential differences between model and interpreter results.

B. Limitations

To address the scalability issues of the SSE mentioned above, we impose several restrictions on the SSE analysis for Zephyr. First and foremost, we ignore time-triggered tasks and alarms in our evaluation, since otherwise the state explosion cannot be handled without additional timing analysis of the rest of the ECFW application. We also introduce an artificially shortened hyperperiod. Although the analysis is complete for this period, we ignore states that may occur later, such as a message queue buffer that might run full after several consecutive interrupts. For some kernel abstractions, such as message queues, semaphores, and mutexes, we have analyzed additional hand-crafted examples and can correctly detect such cases. However, there are some unsupported configuration features. For example, the Zephyr adapter currently only implements reading the simple linked-list ready queue. We do not yet support multicore analysis, but it is possible to extend the Zephyr adapter for this case, and IRx can be invoked once per core. In general, if the system call implementation branches depending on any concrete kernel state that has not been properly prepared, the system call analysis may return an incorrect abstract state. We avoid this by preparing and extracting the entire kernel state that might be modified, not just the obviously affected system objects, for each system call.

C. Future Work

With IRx, we are able to statically analyze interactions for arbitrary embedded operating systems. We have performed the SSE analysis for the Zephyr RTOS, which allows the automatic static optimization of non-functional properties such as dependability or predictable timing behavior. Adapters for other specialized real-time or safety-critical operating systems such as μC/OS-III, ThreadX or ARINC 653 systems are left for future work. Next to interaction-specific optimizations in the application, it is also possible to compare and verify the system call behavior of different implementations of the AUTOSAR specification. For example, the STGs of the existing ARA AUTOSAR model can be compared to (open source) AUTOSAR-compliant RTOSs like ERIKA [15] or Trampoline [16]. This way we could statically verify the implementations against the abstract system model or find specific differences related to the analyzed application.

VII. Related Work

Years ago, static analysis and WCET calculation for RTOSs was already performed on the RTEMS kernel without taking interactions into account [17]. Newer analysis frameworks such as the RTSC [2], *d*OSEK [3], and ARA [5] have analyzed the application together with the RTOS semantics to compute WCET or optimize interactions. However, they use an abstract RTOS model to determine system call semantics and are limited to the OSEK/AUTOSAR specification.

For reachability or vulnerability analysis, symbolic execution engines such as KLEE [18] work on an intermediate representation of the application such as LLVM IR. Going a step further, approaches like SYMCC [19] have moved from interpreting an intermediate representation to instrumenting and compiling that IR to execute the resulting binary. Such tools can be used to improve the security and safety of the code under test, but do not incorporate knowledge of the RTOS semantics.

The analysis tool Astrée applies abstract interpretation to check for runtime errors in concurrent C applications and understands the RTOS semantics of ARINC 653 and AUTOSAR [20]. Similarly, the GOBLINT framework can be used to analyze data races in concurrent programs [21] and is continuously extended with new features. These and similar tools use abstract interpretation to analyze critical embedded software and prove the absence of run-time errors, but are not designed for static optimization of the RTOS to the application.

With Trampoline, application-specialized embedded systems can be generated based on an OS model that the authors have created for their OSEK libraries [22]. They use formal methods to verify that the specialized generated system exhibits the expected behavior. The specialization focuses on system-wide dead code elimination and does not consider specialization of RTOS interactions.

VIII. Conclusions

Static analysis of embedded real-time applications enables whole-system optimization, including the interactions between RTOS and application. By applying this static knowledge and tailoring the RTOS exactly to the needs of the application, non-functional properties such as dependability, predictability, and timing behavior of the overall system can be improved. To enable control-flow–sensitive abstract interpretation of arbitrary RTOSs, we propose to interpret the interactions between application and RTOS at the IR level. This way, we make use of the inherent specialized use case of the application and do not need to derive a generic and abstract system model for each RTOS interface. With IRx, we exemplarily execute Zephyr system calls using the Zephyr LLVM IR code to enable static analysis such as the SSE. An RTOS-specific adapter prepares the concrete kernel state and extracts the abstract state of relevant OS objects such as the scheduler ready-list or semaphores after interpreting the system calls. We demonstrate the applicability of this approach by analyzing the Thread Metric benchmarks and a real-world Zephyr-based embedded controller firmware.

References

[1] AUTOSAR. "Specification of Operating System (R24-11). " https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_OS.pdf.

[2] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '10)*, 2010.

[3] C. Dietrich, M. Hoffmann, and D. Lohmann, "Cross-kernel control-flow-graph analysis for event-driven real-time systems," in *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*, 2015.

[4] G. Entrup, A. Kässens, B. Fiedler, and D. Lohmann, "Applied static analysis and specialization of cross-core syscalls for multi-core AUTOSAR OS," *Real-Time Systems*, 2024.

[5] G. Entrup, B. Steinmeier, and C. Dietrich, "Ara: Automatic instance-level analysis in real-time systems," in *Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '19)*, 2019.

[6] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1977.

[7] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, "Syswcet: Whole-system response-time analysis for fixed-priority real-time systems," in *Proceedings of the 23rd IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '17)*, 2017.

[8] N. Audsley et al., "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, 1993.

[9] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded Real-Time Systems (WATERS '15)*, 2015.

[10] *Zephyr Project homepage.* https://www.zephyrproject.org/.

[11] B. Wang and M. Seltzer, "Tinkertoy: Build your own operating systems for IoT devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[12] Intel. "Embedded controller firmware documentation. " https://intel.github.io/ecfw-zephyr.

[13] M. Silva et al., "ChamelIoT: A tightly- and loosely-coupled hardware-assisted OS framework for low-end IoT devices," *Real-Time Systems*,

[14] Microsoft Corporation. "Thread-Metric RTOS Test Suite. " https://github.com/eclipse-threadx/threadx/blob/master/utility/benchmarks/thread_metric.

[15] *ERIKA Enterprise.* http://erika.tuxfamily.org.

[16] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline: An OpenSource implementation of the OSEK/VDX RTOS specification," in *IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06.*, 2006.

[17] A. Colin and I. Puaut, "Worst-case execution time analysis of the RTEMS real-time operating system," in *Proceedings 13th Euromicro Conference on Real-Time Systems*, IEEE, 2001.

[18] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th Symposium on Operating System Design and Implementation (OSDI '08)*, 2008.

[19] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[20] A. Miné, "Static analysis of embedded real-time concurrent software with dynamic priorities," *Electronic Notes in Theoretical Computer Science*, 2017.

[21] V. Vojdani and V. Vene, "Goblint: Path-sensitive data race analysis," *Annales Univ. Sci. Budapest., Sect. Comp.*, 2009.

[22] K. Tigori, J.-L. Béchennec, S. Faucou, and O. Roux, "Formal model-based synthesis of application-specific static RTOS," *ACM Transactions on Embedded Computing Systems*, 2017.

Bounded Resource Reclamation

Viktor Reusch
Barkhausen Institut
Dresden, Germany

***Abstract*—Resource allocation is well-studied in operating systems, but resource reclamation remains largely underexplored. This paper investigates the impact of unpredictable resource reclamation latency on system behavior, particularly in resource- and time-constrained environments like Open-RAN and serverless functions. We study scenarios of high reclamation times across various systems. Under adverse conditions, reclaiming resources can delay process termination by multiple seconds on both Linux and the L4Re microkernel. We propose a design for accounting and bounding resource reclamation latency to enable predictable system operation and mitigate potential denial-of-service scenarios. We also advocate for optimizing for the case of bulk reclamation — reducing worst-case reclamation latency by multiple orders of magnitude.**

***Index Terms*—Operating Systems, Real-time systems and embedded systems, Reliability, Allocation/deallocation strategies**

I. Introduction

Resource management is a key concern when seeking *predictable system behavior*. Processes frequently allocate and release resources during their lifetime. Thus, the performance and timeliness of resource allocations through the OS is critical for stable application performance. This includes the consideration of tail latencies and worst-case execution times. Consequently, existing work focuses on improving the performance of allocation operations.

In contrast, the behavior of resource reclamation is understudied. The operating system has to manage a variety of limited resources like user memory (e.g., for heaps), kernel memory (e.g., for page tables), or more specific resources like available TCP port numbers. Because these resources are limited, the OS has to reclaim them to make them available to new allocations. These reclamation operations are especially prevalent during the termination of processes. Reclamations occur in bulk as resources held by the terminated processes are released. This can lead to unpredictable system behavior mainly due to two effects: First, the reclamation operations themselves need to be executed and thus they occupy the CPU. This is often reflected by terminations of processes taking longer than usual. Meanwhile, the operating system collects the released resources. Second, if the whole system is running resource constrained, consecutive allocations (e.g., during startup of a new process) have to wait for resources to be released. Thus, new processes have to wait on the termination of previous processes to fully release resources. In conclusion, the latency of release operations affects the subsequent system behavior causing a threat to predictability.

There are prominent use cases that could benefit from a predictable resource reclamation behavior. Predictability can be ensured by enabling the operating system to proactively enforce a time bound on reclamation operations. We call this concept *bounded resource reclamation* (*BRR*). A first possible use case of BRR is a resource-constrained multi-user system as found in software-defined radio solutions like *Open-RAN*. Such Open-RAN units are numerously deployed in the field with limited system resources due to, e.g., power and cost constraints. Thus, applications of different stakeholders need to share resources. These systems also need to be dynamic, allowing termination and recreation of applications. Resource reclamations of terminating processes could interfere with newly created processes thus delaying application startup. This shows the necessity of BRR in resource-constrained systems. Second, also cloud settings, like function-as-a-service environments, can benefit from bounded resource reclamation. Cloud functions are typically assigned a fixed main memory allocation and a hard time limit. This allows the cloud provider to cost-effectively maximize system utilization. However, this utilization management is in vain if resource reclamation is not considered. As shown in Fig. 1, a rogue function of an untrusted customer could deliberately prolong resource reclamation and thus delay system operation. Again, the concept of BRR is needed to solve this issue.

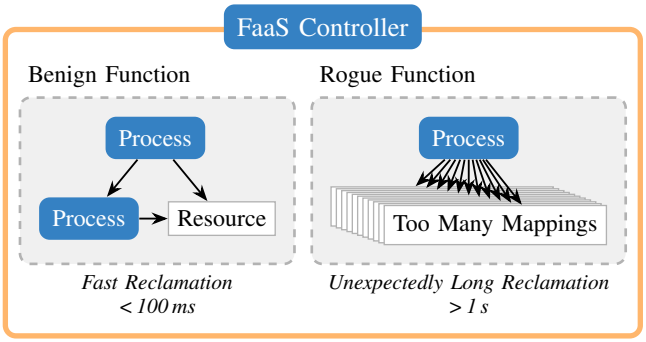


Fig. 1. While most benign functions in a FaaS environment terminate quickly, some other functions might behave unexpectedly and allocate many system resources in rather complicated ways. Reclaiming these resources takes unexpectedly long.

To tackle the issue of unpredictable reclamation behavior, we propose to plan ahead for resource reclamation. We especially focus on the bulk reclamation of resources during application termination. This seems to be the most prevalent scenario, at least in the described use cases. A predictable system should implement accounting of the time needed to reclaim resources and thus allow setting strict time bounds. This enables controller services or orchestrators to guarantee timely reclamation of resources even when untrusted applications exceed their

time budget and have to be forcefully terminated. We also propose to optimize for bulk reclamations by employing arena allocation for management data structures, which drastically reduces the cost of reclaiming whole process trees.

This paper starts with an analysis of the worst-case reclamation time of process resources on various systems (Section II). We present a design of a system that accounts for the latency of resource reclamations (Section III). Additionally, we advertise for grouping resources in prospect of reclamation to accelerate system operation. The implementation of a prototype of this design is then presented in Section IV and evaluated in Section V.

II. Study of Reclamation Latency

To show the necessity for bounded resource reclamation, we examine whether unbounded resource reclamation can lead to unexpected delays in system operation. This section first demonstrates that resource reclamation can actually delay process termination by noticeable amounts. In the examined scenario, a process allocates an unusually large amount of resources in a rather complex way, e.g., by allocating a lot of memory as singular pages. These allocations are correctly accounted towards the CPU time quota of the process. However, when the process gets terminated, e.g., due to exceeding its FaaS time limit, resource reclamation will take unusually long. This reclamation time is not accounted for and might lead to unexpected system behavior by delaying further application startups.

This study of reclamation latency looks at the resource reclamation of kernel memory under three different operating systems: Linux, L4Re [1], and M^3 [2]. On all these platforms, a single process is spawned. It requests the allocation of many page mappings to the same physical page. So no additional user memory is needed to back the page mappings. However, the management structures for these mappings require lots of kernel memory. Later, the process is forcefully terminated. The termination latency is measured to see how resource reclamation prolongs termination latency.

Another, similar benchmark is conducted to assess the potential of the reclamation process to slow down the progress of the whole system. During the termination of the process with the mappings, another workload is started. This workload consists of spawning processes in a tight loop, stressing the kernel subsystems. By measuring how much this workload is slowed down, one can asses how much the operation of a, e.g., FaaS system would be slowed down by long reclamation operations.

Linux is a very common platform for running FaaS workloads, which could experience unbounded resource reclamation. The Linux benchmarks of Fig. 2 are run using kernel version 6.7.4 on an Intel Xeon Platinum 8358 CPU. The governor is set to performance, SMT is disabled, and benchmark programs are pinned to a single CPU core. As the plot on the top left shows, the termination of a *small* Linux process normally takes only around 72.7 µs. However, when the process has one thousand memory mappings, the termination latency already increases to 334 µs. In its extreme, the latency can reach up 13.8 seconds for cleaning up 32 million mappings. Of course, having these many mappings also requires to a lot of kernel slab memory — around 8.9 GiB more. The bottom left plot in Fig. 2 highlights how a workload running during the termination is affected. For this, the plot depicts the runtime of the concurrently-running workload relative to its base runtime in a quiescent system. Thus a relative runtime of one would imply that termination and reclamation have no effect on the workload runtime. When the terminating process only has a single mapping, the concurrent workload is only about 12.2 % slower than its baseline. If the termination latency is high, we see a slow down of up to 115 %. So the workload execution time is roughly doubled. The Linux scheduler seems to equally distribute CPU time between the reclamation operation and the workload. This behavior is, of course, not applicable to all workloads and scheduling strategies but gives one concrete example of termination latency affecting system performance.

The second column of plots in Fig. 2 depicts the results under the L4Re system. The system hardware is the same as previously under Linux. L4Re is a small, microkernel-based operating system with real-time capabilities. Discussing resource reclamation latency on L4Re is not only interesting when discussing real-time but also in the light of recent work that is exploring function-as-a-service workloads on L4Re [3]. In the top L4Re plot, we again see a correlation between mapping count and termination latency. The latency varies from 9.7 ms up to 3.0 s depending on the number of mappings. There is again a big increase in kernel memory consumption of 724 MiB. Concurrent workloads on the L4Re platform are slowed down by the reclamation operations as shown by the lower plot. At 100 000 mappings, the concurrent workload runs 29 % slower. When reclaiming 32 million mappings, the execution time of the workload even becomes 106 times longer. The likely cause of this large slowdown is the *helping* strategy of the L4Re kernel locks. The reclamation code path in the kernel often has to acquire locks. Other kernel threads that try to acquire the already-taken locks will lend the CPU to the lock-holding thread so that it can make progress and release the lock eventually. This leads to the termination operation receiving more CPU time than the concurrent workload.

The third system examined in Fig. 2 is M^3, which is a hardware-software co-design that focuses on security-critical use cases, such as telecommunication infrastructure [4]. M^3 has unique hardware resources, like hardware-based communication channels, that make it interesting for a study on resource reclamation. The results under M^3 are obtained using the gem5 [5] system simulator.[1] Due to the comparatively slow speed of simulation, we limited our M^3 scenario to thousands of kernel structures. The M^3 system not only supports large numbers of memory mappings but also of semaphore kernel objects. Thus, we additionally examined semaphores under M^3. The plots show a similar overall trend for M^3 as already for L4Re. For example, termination latency increases from 313 µs to 16.2 ms when reclaiming 10 000 semaphore objects. Because the count of kernel objects is lower in the M^3

[1] A simulator is necessary because of the custom hardware components in an M^3 system.

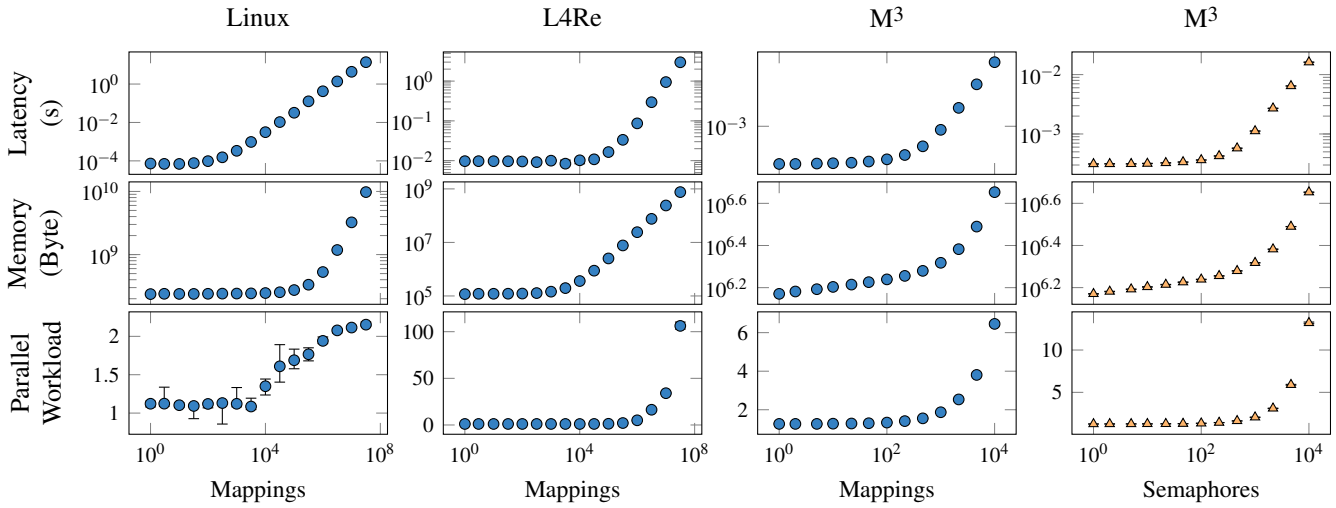


Fig. 2. These plots evaluate our study on reclamation latency. The columns represent different OSes and types of kernel objects. The first row shows the latency of process termination depending on the number of kernel objects allocated. The kernel memory footprint before termination is shown in the second row. The third row shows the relative execution time slowdown of a workload running concurrent with the termination. All depicted values are medians with error bars showing the fifth and 95th percentile. Most axes have a logarithmic scale to better show correlations across orders of magnitude.

benchmarks, the increase in kernel memory consumption is only about 2.88 MiB. Similar to the L4Re scenario, a concurrent workload is heavily slowed down by concurrent reclamations. The likely cause is that the scheduling of system calls in the M³ kernel does not make fairness considerations.

Overall, this study on termination latency reveals that reclaiming resources — at least in the kernel — can take a considerable amount of time. Of course, allocating high numbers of kernel objects also entails consuming larger amounts of kernel memory. Thus, one could try to limit the latency of memory reclamations by using existing mechanisms for kernel memory accounting, e.g., via Linux cgroups [6]. However, limiting the kernel memory usage of specific processes is only a proxy metric for reclamation time. There is currently no holistic approach for enforcing a time bound on resource reclamation. Thus, the next section will discuss how a system for explicitly accounting reclamation time can be designed. This holistic approach also gives the opportunity to optimize resource management for the case of bulk reclamation, e.g., on process termination. This might be particularly interesting for short-lived function is FaaS settings.

III. Design for Bounded Resource Reclamation

The analysis in the previous section has shown that reclamation latency can be a threat to predictable system behavior. Hence, there arises the need for a design ensuring bounded resource reclamation. As motivated by the use cases, we assume that there is always some controller in the system that creates and terminates (groups of) child processes. This controller is also the entity that should enforce a time bound on the reclamation of resources. For example, a FaaS provider might want to limit the latency of processes termination to at most 100 ms. This allows for predictable high-level scheduling of customer functions. Hence, a controller should be able to set a bound of 100 ms on the reclamation latency of each child

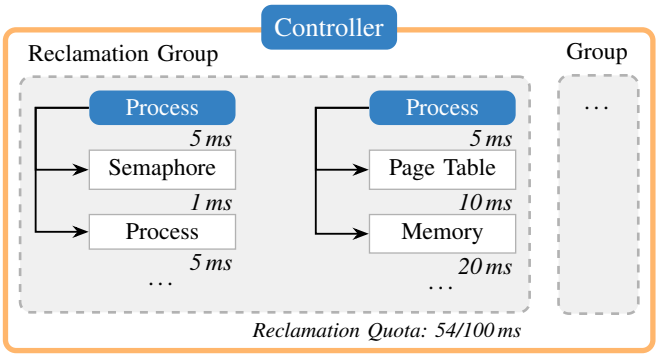


Fig. 3. The abstract design of BRR assigns each reclamation group a reclamation time budget. Whenever a kernel object is allocated, the expected time needed to reclaim this object is subtracted from the group's quota. If the quota reaches zero, no further allocations can be performed.

group of processes. Reclamation latency thus has to become a virtual resource on its own with budgets and accounting.

For the simplest form of BRR, the controller would assign every (group of) processes with a specific reclamation time budget. Then, whenever a resource is allocated the OS checks if this resource will be reclaimed when the group is terminated. For such allocations, the OS subtracts the anticipated reclamation time from the group's quota. The combined reclamation latency of all allocations made by the group must thus not exceed its budget. We call a group of processes and resources sharing the same reclamation quota a *reclamation group*. An example of this approach is shown in Fig. 3. In summary, the OS accounts for the reclamation time of processes in advance.

Of course, there are some complications in real-world operating systems. Most prominently, processes can nest by forking of child processes. These should also be subject to the same budget constraints. Thus, reclamation budgets have to be inherited.

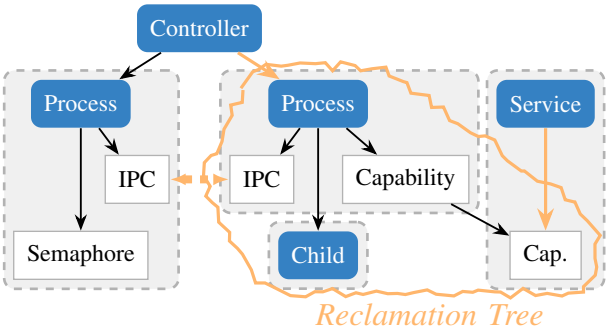


Fig. 4. Whenever a process group or a process hierarchy is terminated, many kernel resources get freed. This includes child processes and derived capabilities. All these released kernel objects form the reclamation tree. There are also connections from the outside to objects inside the reclamation tree, e.g., IPC channels to system services.

Furthermore, the reclamation of a process might trigger the deletion of other OS objects like network sockets or capabilities. Capabilities on microkernels, like L4Re and M^3, are especially interesting as they can be delegated and derived. This essentially creates inheritance trees inside the kernel that also need to be honored for BRR. Overall, these interdependencies in the different OS objects mean that upon termination of a process, a whole subtree of objects will be reclaimed. We call this subtree the *reclamation tree* starting from some root process. An example is given in Fig. 4. The reclamation latency of all the objects in the reclamation tree of a reclamation group has to be accounted in the associated reclamation quota for accurate accounting.

The observation of the reclamation tree leads to a possible optimization. For the use cases of FaaS and Open-RAN, one can assume that there are few interconnections between the reclamation tree of a customer process and the rest of the system. These interconnections are likely limited to user memory and networking. Presumably, the bulk of reclamation operations for these customer processes are internal to the reclamation tree. Examples for such reclamation operations are deleting user memory mappings, terminating inter-process-communication channels, or deallocating process objects in the kernel. These cleaned-up objects are only linked internally with respect to the reclamation tree. So all objects in the reclamation tree will be reclaimed during a bulk reclamation. Henceforth, there is actually no need to unlink individual objects. Both sides of these object-to-object links will be freed and reclaimed. Thus, only interconnections to the rest of the system that link outside of the reclamation tree need to be unlinked.

Going a step further, one can optimize the deallocation of all these objects inside the reclamation tree by using some form of arena allocator for all objects inside the reclamation tree. The reclamation group would get a single, large chunk of memory (the arena) in advance. Afterwards, all objects in the reclamation tree of the group are allocated in memory inside of the arena. This grouping of objects allows the reclamation of a whole tree by simply reclaiming one large chunk of memory and unlinking a couple of outside interconnections — opposed to deallocating each object individually. This optimization should greatly reduce reclamation time and thus help to keep bounds on reclamation time low.

IV. Prototypical Implementation

To test the effectiveness of our approach, we have implemented a prototype of bounded resource reclamation. We chose the M^3 operating system as our implementation platform. M^3's microkernel design fits well to the idea of a reclamation tree. Kernel objects and capability inheritance already span a graph inside kernel memory. Furthermore, M^3 is a hardware/operating-system co-design with interesting hardware resources that need to be considered during reclamation. The fundamental design idea of M^3 is to isolate individual CPU cores using custom hardware-based isolation units. This tiled architecture already sketches boundaries for process groups that can be leveraged for BRR. In this first prototype, we limit our implementation to the grouping mechanism in the kernel itself including arena allocation. The actual accounting of the reclamation time, especially for connections leading outside of the reclamation tree, will be added in future work.

First, one has to consider how a reclamation tree looks like in the M^3 kernel. The M^3 microkernel is capability-based and thus the kernel manages a list of capabilities for each process. This means that processes can only access the kernel functionalities/abstractions they have a capability to. In the kernel, these capabilities point to kernel objects like semaphores, process structures, or page mapping entries. Processes on M^3 can collaborate by exchanging capabilities pointing to these objects. Thus, an inheritance graph of capabilities and kernel objects is created. When a process is terminated and the process structure is reclaimed in the kernel, all capabilities held by the process will be freed as well. Consequently, all inherited capabilities (even when exchanged with other processes) are reclaimed. Kernel objects that are no longer referenced by any capability are reclaimed. Because kernel objects can be process structures themselves, reclamation can recurse into child processes. This can lead to the cleanup of whole process trees in a single system call under M^3.[2] All this recursion has to be reflected in the reclamation tree of processes and thus has to be respected by our implementation of BRR.

In this implementation of bounded resource reclamation, the kernel enables controllers to assign processes to reclamation groups. Whenever then such a process creates or inherits capabilities/kernel objects, the group affiliation is inherited too. This ensures that all objects in the reclamation tree are assigned to the same reclamation group and can thus be accounted for. Additionally, the kernel allocates each object belonging to a group in a group-local memory arena as shown in Fig. 5.

Some objects in the reclamation tree might also point outside of the reclamation group. These could be, e.g., semaphore objects that are shared with an outside, system-wide network service. The BRR implementation addresses these outside connections by referencing the offending objects in a group-local *scrub list*. When reclaiming a group, the objects in the

[2]This is similar to what can be achieved with PID namespaces under Linux.

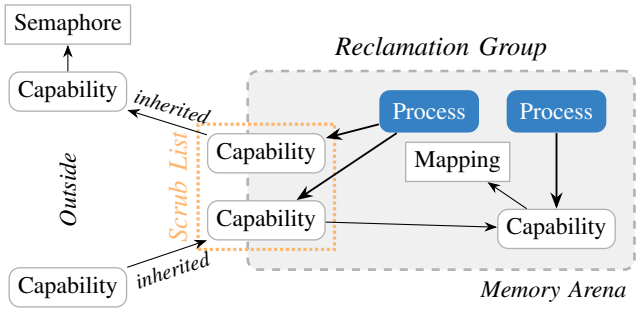


Fig. 5. Kernel objects of the same reclamation group are allocated together inside the same memory arena to optimize for the case of bulk reclamation. Connections leading out of the reclamation group need to be handled individually and are thus noted in a scrub list.

scrub list are unlinked first before the whole arena allocation of the group is deallocated at once. The individual objects inside of the group do not need to be individually unlinked or deallocated, which greatly reduces reclamation latency.

In the future, the arena allocation and each element in the scrub list need to be accounted for in a reclamation time quota. Furthermore, the implementation is currently only fully realized for semaphore kernel objects — just to serve as a proof-of-concept. Nevertheless, we think this implementation can serve for an initial evaluation of the feasibility and effectiveness of the approach as shown in the next section.

V. Evaluation

For the evaluation, we perform measurements on M³ using the gem5 simulator for RISC-V. We compare three different system configurations. First, we measure on a baseline M³ system that does not contain any custom modifications for BRR. Second, we use a modified kernel that implements the changes outlined in the previous section. However, the processes under test are not added to any reclamation group. The results of this configuration show the overhead of the performed code changes to the overall system. For the third configuration, the created processes are actually put inside of a reclamation group to take advantage of the reclamation optimizations.

Figure 6 shows the influence of the modifications on the latency of application startup. The median startup latency increased by only 4.69 µs or 0.81 % from a baseline of 581 µs due to the modifications to the M³ kernel. The increase in latency is higher when starting up the process inside of a group. It increases by 15.3 µs or 2.6 %. This shows that the overhead for the additional bookkeeping of kernel objects inside of reclamation groups and additional branch instructions in system call handlers is small but noticeable.

The overhead of the M³ modifications are also noticeable when creating new kernel objects via system calls. Figure 7 shows that the overhead of grouping is 377 ns or 3.8 % from a baseline of 9.9 µs for creating a single memory mapping. The overhead is more pronounced when creating a semaphore object in the kernel. With the baseline being 3.10 µs, the increase in latency is 1.20 µs or 39 %. This shows that the

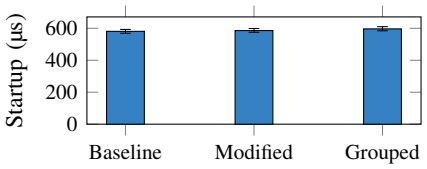


Fig. 6. This plots shows the latency of application startup under an unmodified version of M³, the modified one, and the modified one when starting the process inside of a reclamation group.

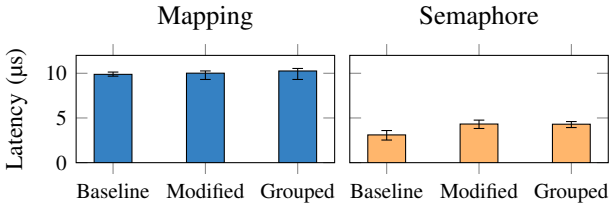


Fig. 7. This shows the latency of allocating a single kernel object under the three tested system settings.

modification to the kernel, which are needed for bounded resource reclamation, can have an effect on short, simple system calls. However, this effect disappears for more involved operations like application startup in Fig. 6.

The preliminary implementation of BRR is already able to drastically speed up termination as shown in Fig. 8. The left most data points show the termination latency of a process with only a single semaphore object allocated inside the kernel. The measured latency is very similar between not using reclamation groups (398 µs) and using the feature (370 µs). Without the group feature and with 10 000 allocated semaphores, the reclamation latency reaches 17 ms. However, with the group feature, the latency increases by only 12 % to 414 µs. This is achieved by allocating kernel objects of a single reclamation group inside a memory arena and freeing them in bulk. The semaphore objects allocated for Fig. 8 do not have any outside connections and can therefore be freed without unlinking individual outside connections. In contrast, the figure clearly shows that without using groups, the reclamation latency increases as all semaphore objects need to be cleaned up individually. Overall, our preliminary implementation already highlights the potential time savings that are possible when optimizing for the case of bulk resource reclamation as often seen in time-constrained FaaS settings.

VI. Discussion

The proposed grouping technique — as a sideeffect — makes memory accounting in the kernel easier and more precise. Normally, kernel memory budgeting can only be an estimate because of fragmentation between different-sized slab allocators and general kernel allocators. Because of this fragmentation, a kernel, like the L4Re microkernel, could run out of memory even though no processes exceeds its kernel memory budget. This imprecise accounting could be avoided by using the proposed grouping approach using an

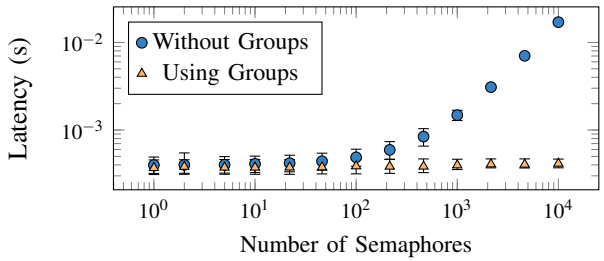


Fig. 8. This plots depicts the time needed to terminate a process depending on the number of semaphore objects it allocated via the kernel. The usage of reclamation groups keeps the termination latency low even for high numbers of objects.

arena allocator. All memory is accounted for in advance and fragmentation issues are contained to the individual groups. Although this comes at the cost of flexibly and dynamically using kernel memory, one could argue that this flexibility is not needed in settings like FaaS. Resources are typically assigned in advance to the various customer's processes to allow partitioning the multi-tenant cloud machines.

Though, there are challenges when elevating BRR to the whole operating system. System services also need to reclaim resources once client processes terminate. This includes network services closing sockets and memory services freeing user memory. One possible solution could include some interplay between OS services and the kernel to manage reclamation time quotas. Alternatively, the controller could allocate separate reclamation budgets at every service. To keep the first prototype simple, this paper focuses on the kernel-internal reclamations of kernel memory.

VII. Related Work

To the best of our knowledge, there is no previous work about a holistic approach to account for reclamation time and optimize for it. Nevertheless, there are related works that partially overlap with the goals of bounded resource reclamation.

Blackham et al. [7] analyze the worst-case execution time of operations in the seL4 kernel. They make sure that non-interruptible sections in the kernel are bounded — thus making it feasible to swiftly react to interrupts. A similar timing analysis would also aid BRR as it is important to estimate the latency of individual reclamation operations. The analysis of Blackham et al. does not put a bound on the latency of reclamations as a whole, only on the individual non-interruptible sub-operations of which there can be unboundedly many. This gap can be filled with by the described design of bounded resource reclamation.

There are quite a few works that are concerned with real-time memory management using dynamic garbage collection [8], [9], [10]. This also entails reclaiming memory with predictable latency to make it available for future allocations. For example, Baker [8] designs a real-time garbage collector that reclaims memory during subsequent allocations. Thus, system progress is not stalled when releasing memory or during periodic scans. Every operation is rather bounded by constant time. However, it is unclear how this concept could be practically transferred to operating systems. First, resources need to be handled by a global garbage collection algorithm. This does not fit the common design of manual resource management typically found in OS kernels. Second, application performance would depend on the amount of resource reclamations that have to happen during allocation operations. Thus, the performance of, e.g., FaaS functions would still depend on the behavior of the previous tenant of the system — just with better predictable time bounds. In contrast, bounded resource reclamation eliminates this correlation by reclaiming all resources directly on termination in bounded time.

The *Thundering Herd* attack by Mergendahl et al. [11] is an example of why predictable system behavior is important. Mergendahl et al. are concerned with attacks that use the kernel in unusual ways to introduce unexpected timing behavior in victim threads. The described attacks make use of the inner workings of the seL4 scheduler implementation to break temporal isolation. This allows many low-priority threads to arbitrarily delay scheduling of a high-priority thead.

Furthermore, there are existing OS mechanisms to enforce restrictions on resource consumptions. Both Linux' *cgroups* and L4Re's *factories* allow processes to restrict (kernel) memory consumption of children. However, simply restricting the maximum allocation of memory does not serve as an adequate proxy for limiting reclamation latency on termination. For example, the cgroup owning shared memory is in-deterministic when multiple cgroups are involved [6]. The factory abstraction in the L4Re microkernel is only concerned with accounting kernel memory. Thus, memory consumption in services on behalf of applications needs to be handled separately. In general, using existing memory accounting to enforce a bound on reclamation latency is imprecise as the latency to reclaim different objects varies. For example, reclaiming a single, large allocation for a task stack might be a lot faster than reclaiming individual, small semaphore objects. BRR offers a holistic solution by accounting for reclamation latency in advance.

VIII. Conclusion and Future Work

This paper has demonstrated the critical impact of resource reclamation latency on system predictability. Our study showed that uncontrolled reclamation can significantly prolong process termination. The observed delays under Linux, L4Re, and M^3 underscore the need for proactive management of reclamation behavior. Our proposed design, focused on accounting and bounding reclamation latency, allows for more reliable systems. Additionally, the possibility of optimizing for bulk reclamations emerges with the potential for orders-of-magnitude latency reductions.

The current implementation is only in a prototypical state with lots of improvements for future work. We would like our implementation to support the actual, precise accounting of reclamation time and to work with more kinds of kernel objects (especially memory mappings). In the end, BRR needs to be expanded to the whole operating system, including services, to fully bring predictable reclamation behavior to real-world scenarios.

REFERENCES

[1] A. Lackorzynski and A. Warg, "Taming subsystems: Capabilities as universal resource access control in L4," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, IIES '09, Nuremburg, Germany, March 31, 2009*, M. Engel and J. Nolte, Eds., ACM, 2009, pp. 25–30. DOI: 10.1145/1519130.1519135. [Online]. Available: https://doi.org/10.1145/1519130.1519135.

[2] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. P. Fettweis, "M3: A hardware/operating-system co-design to tame heterogeneous manycores," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, T. Conte and Y. Zhou, Eds., ACM, 2016, pp. 189–203. DOI: 10.1145/2872362.2872371. [Online]. Available: https://doi.org/10.1145/2872362.2872371.

[3] T. Miemietz et al., "A perfect fit? - towards containers on microkernels," in *Proceedings of the 10th International Workshop on Container Technologies and Container Clouds, WOC 2024, Hong Kong, Hong Kong, December 2-6, 2024*, ACM, 2024, pp. 1–6. DOI: 10.1145/3702637.3702957. [Online]. Available: https://doi.org/10.1145/3702637.3702957.

[4] S. Haas et al., "Trustworthy computing for O-RAN: security in a latency-sensitive environment," in *IEEE Globecom 2022 Workshops, Rio de Janeiro, Brazil, December 4-8, 2022*, IEEE, 2022, pp. 826–831. DOI: 10.1109/GCWKSHPS56602.2022.10008543. [Online]. Available: https://doi.org/10.1109/GCWkshps56602.2022.10008543.

[5] N. L. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011. DOI: 10.1145/2024716.2024718. [Online]. Available: https://doi.org/10.1145/2024716.2024718.

[6] T. Heo. "Control group v2," The Linux Kernel documentation, Accessed: Apr. 16, 2025. [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v2.html.

[7] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, IEEE Computer Society, 2011, pp. 339–348. DOI: 10.1109/RTSS.2011.38. [Online]. Available: https://doi.org/10.1109/RTSS.2011.38.

[8] H. G. Baker Jr., "List processing in real time on a serial computer," *Commun. ACM*, vol. 21, no. 4, pp. 280–294, 1978. DOI: 10.1145/359460.359470. [Online]. Available: https://doi.org/10.1145/359460.359470.

[9] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Commun. ACM*, vol. 26, no. 6, pp. 419–429, 1983. DOI: 10.1145/358141.358147. [Online]. Available: https://doi.org/10.1145/358141.358147.

[10] P. Cheng and G. E. Blelloch, "A parallel, real-time garbage collector," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, M. Burke and M. L. Soffa, Eds., ACM, 2001, pp. 125–136. DOI: 10.1145/378795.378823. [Online]. Available: https://doi.org/10.1145/378795.378823.

[11] S. Mergendahl, S. Jero, B. C. Ward, J. Furgala, G. Parmer, and R. Skowyra, "The thundering herd: Amplifying kernel interference to attack response times," in *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022*, IEEE, 2022, pp. 95–107. DOI: 10.1109/RTAS54340.2022.00016. [Online]. Available: https://doi.org/10.1109/RTAS54340.2022.00016.

RT-Bench: A Long Overdue Update

Mattia Nicolella
Boston University
Boston, Massachusetts, U.S.A.
mnico@bu.edu

Denis Hoornaert
Technical University of Munich
Munich, Germany
denis.hoornaert@tum.de

Renato Mancuso
Boston University
Boston, Massachusetts, U.S.A.
rmancuso@bu.edu

***Abstract*—RT-Bench is a framework and community project that aims to establish a unified set of benchmarks with a homogeneous launch and result reporting interface, and with a simple build system. RT-Bench targets academic researchers and industry practitioners interested in understanding the performance characteristics of embedded/real-time systems when tested over realistic use-case applications. To facilitate real-time systems research, RT-Bench is designed from the ground up to include a set of fundamental capabilities such as periodic execution, selectable OS scheduler, and native and multi-architecture performance counters support, to name a few.**

RT-Bench has undergone continuous improvements and extensions. This paper reviews the most recent additions and features of the framework. Most prominently, these include heap migration, synchronized benchmark release, and experimental support for multi-threaded applications. This contribution includes a tutorial session with template benchmarks to showcase the new features and illustrate the process of integrating new benchmark suites.

***Index Terms*—Benchmarking, Real-time, Profiling, Periodic benchmarks**

I. Introduction

For practitioners and academics, benchmarking constitutes an essential step in testing and validating their systems regardless of the application domain. Naturally, many domain-specific benchmark suites have emerged over time as a result of independent efforts. On the one hand, this organic growth has offered a broad and diversified portfolio of benchmarks. On the other hand, however, it has caused an inherent fragmentation w.r.t. the set of features, launch/command interfaces, and supported metrics. This lack of a *de facto* standard primarily hinders reproducibility, as well as productivity and adoption, since practitioners must manually adapt each suite of interest to their needs via a repetitive and time-consuming process.

Since its release [1], RT-Bench [2] has aimed to reduce these frictions through a rich standardized interface that enables compatible benchmarks to tap into its feature set seamlessly. For instance, the framework allows users to swiftly leverage common timing features, such as periodic execution and reporting of elapsed time, in all the included benchmarks. Not only does RT-Bench focus on features relevant to real-time system benchmarking, but it also provides native support for resource profiling, which has proven helpful in understanding run-time resource requirements and pinpointing performance bottlenecks in more general settings.

The framework also strives to combine user-friendliness, requiring minimal effort when adapting to new benchmarks. To that end, RT-Bench's contributors have maintained an extensive documentation [3] of the framework, ranging from the supported benchmark suites to the framework's APIs.

This article and its associated tutorial session aim to reiterate RT-Bench's core concepts, mechanisms, and goals while formally introducing the newly ratified features.

II. What's new?

Since its first release in 2022 [1], RT-Bench has known a relative success within the real-time community [4]–[10][1]; calling for many improvements, feature additions, feature revision, bug fixes and code consolidation. This section describes the most prominent changes brought to RT-Bench.

A. Continuous Back-to-back Executions.

One of the first features introduced in RT-Bench in an effort to more realistically reflect the behavior of real-time applications was the enforcement of periodic execution. As RT-Bench increased in adoption, however, it was brought to our attention that many scenarios and use cases exist where this mode of operation is not ideal and, in fact, undesirable.

Typically, these correspond to cases where the role of the benchmark is to create pressure on the system's resources (i.e., bandwidth from IsolBench [11]). As such, the intermittent activity of interfering workloads that comes from periodic releases and missed deadlines (and consequent job skipping, showed in Fig. 1, top part) creates *"pressure gaps"* that can make empirical *worst-cases* harder to observe and reliably reproduce. Such gaps are also inconvenient when attempting to collect stable readings of the performance counters.

Hence, a *"continuous back-to-back execution"* mode has been introduced (Fig. 1, bottom part). It can be enabled simply by omitting a period (`-p`), ensuring retro-compatibility with existing RT-Bench command options. Note that nothing else changes as the benchmark execution routine is still executed in a loop until a `SIGINT` is received or until the specified number of tasks instances (`-t`) has been completed.

B. Heap Migration

The push toward heterogeneous System-on-Chips is not just confined to processing elements. Modern embedded platforms, in particular, also tend to feature a diversified array of on- and off-chip memories with different sizes and temporal

[1]For sake of transparency, cited papers exclude research items involving the authors of the original paper [1] and their close collaborators.

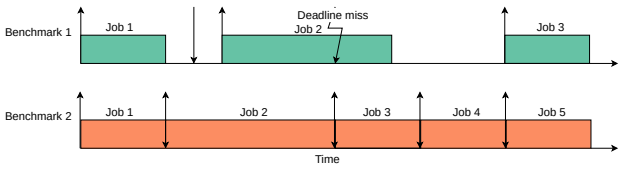


Fig. 1: RT-Bench execution modes: Periodic execution (top) and continuous back-to-back (bottom).

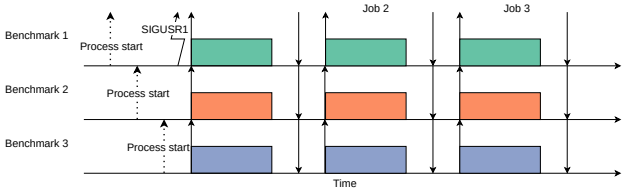


Fig. 2: RT-Bench synchronized release example with three benchmarks.

characteristics. On Linux-based systems, these memories are often treated as reserved memory nodes. Thus, unless system designers add ad-hoc support, these nodes remain natively inaccessible to standard user-space applications.

To enable the study of benchmarks' memory affinity, RT-Bench integrates a user-friendly *heap migration* mechanism. When heap migration is enabled, RT-Bench switches to a custom heap allocator over an internally allocated memory region. The user can control the type of allocation via command-line parameters. Specifically, when enabling heap migration (`-H`), users can indicate the target memory in two ways: (1) by providing a physical memory address; or (2) by providing a file pathname.

In the former case, RT-Bench will perform a non-cacheable memory mapping using `/dev/mem` and will use said mapping as the applications' heap memory region. In the latter case, RT-Bench will memory-map the provided file. If this is a normal file (e.g., in a `tempfs` filesystem), the resulting memory region will be cacheable. This option is handy as reserved heterogeneous memory nodes can be exposed to user space by exporting them as block devices that can be memory-mapped. Kernel-level support for the creation of said interface, however, is out of the scope of RT-Bench. When using the (`-H`) option in either mode, the end-users are responsible for providing adequate access to the underlying memory (e.g., by ensuring it is a usable physical address/file) to avoid bus errors.

Note that a heap size limit parameter (`-m`) is required because RT-Bench must terminate the target benchmark if the working set size exceeds the size of target memory region.

C. External Synchronized Release

As RT-Bench runs atop of a rich full-fledged operating system (i.e., Linux), running several benchmarks concurrently to observe and study their interactions is already possible. However, no control over the release time of the workloads is usually possible. This makes the simulation of specific conditions, such as the worst-case job release pattern, hard to perform.

Thus, an initial release (offset) synchronization mechanism has been added to RT-Bench. This mechanism, shown in Fig. 2, allows different benchmarks or different instances of the same benchmark to be released together via the newly introduced (`-s[=TASKSET_NAME]`) parameter. When the parameter is used, the released benchmarks behave as if they belong to the same task set with synchronized release offset. Specifying a task set name is optional. If an explicit name is not provided, RT-Bench will default to an implicit name.

Practically speaking, RT-Bench's synchronization unrolls in three phases. Initially, all workloads initialize themselves until they are ready to execute for a first iteration. There, they wait to receive a synchronization signal (i.e., `SIGUSR1`). This behavior allows users to add an arbitrary number of benchmarks to the set by simply starting them with the same (`-s`) parameter value. Once all the desired benchmarks have been launched, the user can start the full experiment by sending the `SIGUSR1` signal to any one of the benchmarks in the set. Once any of the benchmarks receives the signal, said benchmark becomes the *synchronization manager*. The manager is endowed with the responsibility to (1) determine a synchronized release time in the near future and (2) send a signal to wake up all the other benchmarks (subordinates). Upon reception of the signal, each subordinate benchmark, as well as the manager, will configure an absolute release timer with the exact coordinate time determined earlier.

Note that the mechanism still involves the end-users as they must manually start the required benchmarks and choose a unique task set name (`TASKSET_NAME`). They also must send the first `SIGUSR1` signal once all the desired benchmarks have been launched. Alternatively, a `synch-helper` tool is also provided which allows users to specify how many benchmarks to wait for before the start `SIGUSR1` signal is sent.

D. Extended Reporting of Benchmark-specific Metrics

For some benchmarks, the default performance metrics reported by RT-Bench are only a subset of the information useful for the experiment at hand. Since its initial release, RT-Bench offers the possibility to report one benchmark-specific metric for each benchmark. This feature is used, by the applications in the IsolBench suite [11], which also report measured memory latencies and bandwidths. However, in a number of cases, limiting benchmark-specific metric reporting to a single value was deemed too restrictive. For instance, a benchmark performing neural network inference for object identification might want to simultaneously report the number of detected objects *and* the corresponding confidence scores.

For this reason, the benchmark-specific reporting mechanism has been revamped. In this revision, benchmark-specific metrics and metadata are stored in a struct named `extra_measurement` aggregating pairs of headers and metrics. At run-time, the struct is populated by the user-

defined `benchmark_log_data()` routine with the desired metrics and their human-readable name. Accordingly, the `EXTENDED_REPORT` compilation flag has been deprecated.

E. Performance Monitoring Thread Improvements

Recall that RT-Bench included the option to spawn a performance monitoring thread *PMThread*. The PMThread runs in parallel with a given benchmark, and its goal is to perform high-frequency sampling of architectural performance counters to more accurately profile the benchmark under analysis.

An interesting challenge in the design of the PMThread is how to store the trace of performance samples efficiently. In the initial version, sample measurements[2] were stored in one contiguous container (i.e., a buffer) where each entry corresponded to a timestamp in the execution. For a time bucket Δ_t, the i^{th} entry contains the performance measurements sampled at instant $i \times \Delta_t$ of the benchmark run-time. Such pre-allocated container would limit the execution time of the benchmark and eventually lead to a buffer overflow.

The reworked implementation, instead, features support for a list of buffers that can hold these measurements. This is achieved dynamically during the execution phase as follows. Whenever a buffer is filled, a new buffer is allocated and added to the list, with a size that is twice as large as the previous buffer in the list. The approach limits the number of memory allocations during execution while not copying/moving any previously acquired data samples. Finally, any memory used to hold performance counters is not tracked by the memory watchdog, and thus, it does not count against the benchmark's memory limit (`-m`).

F. Experimental Multi-thread Support

The feedback following the previous release of RT-Bench lamented the lack of support for multithreading. The absence was justified by several—feasible but tricky—implementation hurdles such as synchronization and core affinity assignment.

Since then, steady progress has led to the introduction of an experimental support for multithreading. It is now possible to easily use the framework with a multithreaded benchmark since new APIs to (1) spawn new threads and (2) define a parallel computation section with these threads are available. With these APIs the main execution thread can operate before, during and after the parallel section; with minimal changes to RT-Bench's execution logic as shown in Fig. 3.

Note that, at the time of writing, multi-threading cannot be used in conjunction with other options. For instance, as the subsystem used for heap migration and Working Set Size (WSS) reports are not thread-safe, multi-threading can only be used provided no dynamic memory allocations are performed during the execution phases. Likewise, performance counter reports generated by the PMThread only cover the main thread.

III. Demonstration session outline

This section provides a step-by-step overview of the tutorial session accompanying this paper. The objective is to demonstrate the user-friendliness of RT-Bench and provide the audience with key pointers to get started with their projects.

[2]i.e., a `struct` aggregating a selected set of PMCs such as `L2_refills`.

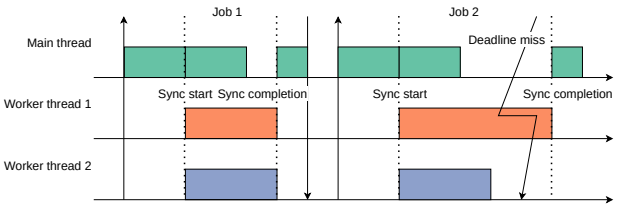


Fig. 3: Execution model for multithreaded benchmarks.

1. Building and Launching a Benchmark. Using a project template [12] specifically made for this occasion, we will showcase how RT-Bench-adapted benchmarks can be compiled and executed. In particular, we will focus on the required command line parameters, scenario configuration via `.json` files, and where to find built binaries. This part will cover how to release a taskset, a heap migration example, and how to retrieve performance measurements, including an example using the PMThread.

2. Adding a Benchmark. This part will showcase how to extend RT-Bench with additional benchmarks [13], whether single- or multi-threaded. The inspection of the single-threaded benchmark from the template will serve as a stepping stone to revise RT-Bench's basics. Specifically, it will include how to split a `main` function into the three required harness functions (i.e., `init`, `execute`, and `teardown`) and tips on how to manage any inputs modified in place during execution. This part of the demo will also cover how to add custom logged metrics. Finally, the steps that must be undertaken to tap into the *experimental* multi-thread support will be shown by adapting the benchmark at hand.

IV. Conclusion & Future Work

RT-Bench benefits from a continued effort in maintaining the project and adding new features. Future work is directed towards (1) adding multithreaded benchmarks and improving the support for multithread execution, (2) supporting C++ benchmarks, and (3) the ability to define and release a taskset from a single `.json` specification. Other future work includes the construction of a RT-Bench database, which contains metrics of different benchmarks categorized by platform.

The authors are certainly committed to providing community support, maintaining, and extending the RT-Bench project. Nonetheless, they also welcome inputs and contributions by the community, such as new benchmarks, features, and support for additional platforms: these are fundamental to keep the project relevant, up to date, and ultimately useful.

V. Acknowledgments

This research was supported by the National Science Foundation (NSF) under grant number CSR-2238476. Denis Hoornaert was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

REFERENCES

[1] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, "Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022, pp. 184–195.

[2] Rt-bench repository. [Online]. Available: https://gitlab.com/rt-bench/rt-bench/-/tree/OSPERT25

[3] Rt-bench documentation. [Online]. Available: https://rt-bench.gitlab.io/rt-bench/branch/OSPERT25/index.html

[4] C.-F. Yang and Y. Shinjo, "Compounded real-time operating systems for rich real-time applications," *IEEE Access*, vol. 13, pp. 26 079–26 104, 2025.

[5] A. Oliveira, G. Moreira, D. Costa, S. Pinto, and T. Gomes, "IA&AI: interference analysis in multi-core embedded AI systems," in *Data Science and Artificial Intelligence*, C. Anutariya, M. M. Bonsangue, E. Budhiarti-Nababan, and O. S. Sitompul, Eds. Singapore: Springer Nature Singapore, 2025, pp. 181–193.

[6] M. A. Soomro, A. Nasrullah, and F. Anwar, "Poster abstract: Time attacks using kernel vulnerabilities," in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 626–627. [Online]. Available: https://doi.org/10.1145/3715014.3724040

[7] M. A. Khelassi and Y. Abdeddaïm, "Impact of compilation optimization levels on execution time variability," in *2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2024, pp. 1–4.

[8] M. A. Khelassi, "Using statistical methods to model and estimate the time variability of programs executed on multicore architectures," Theses, Université Gustave Eiffel, Dec. 2024. [Online]. Available: https://hal.science/tel-04921969

[9] W. Dewit, A. Paolillo, and J. Goossens, "A preliminary assessment of the real-time capabilities of real-time Linux on Raspberry Pi 5," in *18th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*. Alexander Zuepke and Kuan-Hsun Chen, 2024.

[10] K. Hosseini, "Real-time system benchmarking with embedded Linux and RT-Linux on a multi-core hardware platform," Master's thesis, Linköping University, Department of Computer and Information Science, 2024.

[11] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–12.

[12] Rt-bench benchmark suite template. [Online]. Available: https://gitlab.com/rt-bench/templates/benchmark-template/-/tree/OSPERT25

[13] Benchmark template to extend rt-bench. [Online]. Available: https://gitlab.com/rt-bench/templates/adapt-bmark/-/tree/OSPERT25

Real-Time Virtualization on Heterogeneous MPSoCs: A Hands-On Tutorial with Jailhouse and Omnivisor

Daniele Ottaviano

Technical University of Munich
daniele.ottaviano@tum.de

Modern Cyber-Physical Systems (CPS)—such as those in automotive, avionics, and railway domains—increasingly rely on complex, heterogeneous MultiProcessor Systems-on-Chip (MPSoCs) to consolidate workloads with diverse real-time and safety requirements. Combining general-purpose applications and critical control tasks within a single platform presents significant challenges in maintaining isolation and predictability. Static partitioning hypervisors provide a robust solution to address these challenges. However, traditional hypervisors are typically designed for homogeneous multi-core platforms and lack native support for managing heterogeneous cores—such as microcontroller-class CPUs and soft cores on FPGAs—which are becoming increasingly common in commercial MPSoCs.

This tutorial introduces real-time static partitioning virtualization using Jailhouse, a minimalist hypervisor, and its extension Omnivisor, which generalizes static partitioning to heterogeneous ISA cores. Building on recent work presented at ECRTS 2024[1] , the tutorial offers a hands-on exploration of how these tools support the deployment and management of isolated real-time virtual machines (VMs) across different processor types within a single platform.

The session begins with a technical overview and live demonstration of the Jailhouse architecture. We will explore Jailhouse's minimalist design, which leverages the existing Linux system to bootstrap the hypervisor and create isolated "cells" (VMs) with statically assigned resources, using shell commands and configuration files.

Next, we will dive into the Omnivisor extension. As an open-source enhancement, Omnivisor extends Jailhouse to support remote cores with different ISAs (e.g., ARM Cortex-R, RISC-V) on the same SoC, implemented in hardware and on programmable logic (e.g., soft-core on FPGA). It enables unified VM management across these cores, providing mechanisms for spatial and temporal isolation that abstract away the complexity of the underlying heterogeneity. Participants will learn how Jailhouse, with the Omnivisor extension, automates the use of platform-specific isolation features—including MMU, SMMU, and SMPU—and integrate them into the VM lifecycle.

We will then experiment with Omnivisor's dynamic FPGA fabric management. This includes loading the corresponding FPGA bitstream, setting up the necessary device-tree overlays, and injecting the required drivers into the Linux root-cell. This functionality enables the flexible instantiation of isolated VMs on newly deployed soft-cores, fully integrated into the static partitioning framework. Finally, we will focus on interference mitigation and memory bandwidth regulation. We will examine how mechanisms like MemGuard, QoS, and MemPol are integrated into the hypervisor to provide temporal isolation guarantees across VMs, even under high system contention.

Participants will learn how to deploy and manage real-time workloads on heterogeneous MPSoCs using Jailhouse and Omnivisor. They will gain insights into the design challenges and solutions for virtualization on platforms with asymmetric cores and will be provided with access to open-source code, tooling, and documentation to reproduce the setup and experiments on their own.

[1] D. Ottaviano, F. Ciraolo, R. Mancuso, and M. Cinque, "The Omnivisor: A Real-Time Static Partitioning Hypervisor Extension for Heterogeneous Core Virtualization over MPSoCs," in Proceedings of the 36th Euromicro Conference on Real-Time Systems (ECRTS 2024), vol. 298. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024, pp. 7:1–7:27.

Call for Collaboration: Contributing to Multi-Messenger Astrophysics

Marion Sudvarg*†, Ye Htet*, Roger D. Chamberlain*, Jeremy D. Buhler*, James H. Buckley†
*Department of Computer Science and Engineering, †Department of Physics
Washington University in St. Louis
{msudvarg, htet.ye, roger, jbuhler, buckley}@wustl.edu

***Abstract*—In multi-messenger astrophysics, signals of multiple types (e.g., gravitational waves, neutrinos, electromagnetic waves) are combined in an effort to learn more about the observed phenomena of interest. The Advanced Particle-astrophyics Telescope (APT) is a mission concept for a space-borne instrument that detects gamma-ray bursts (GRBs) omnidirectionally, facilitating multi-messenger observations by identifying and localizing celestial events of interest. In this presentation, we describe the current status of on-instrument computations for the Antarctic Demonstrator for APT (ADAPT) pursuant to guiding prompt follow-up observations of transient events. We also describe open problems and the challenges of extending ADAPT's computation to the future APT instrument. We encourage contributions and collaboration from members of the real-time systems community.**

I. Introduction

Background and Motivation. The astrophysics community has a strong interest in observing transient astrophysical phenomena using multiple modalities. This *multi-messenger* approach may include, e.g., gravitational waves, electromagnetic waves, neutrinos, and cosmic rays [1], [2]. Because these transients can be short-lived [3], fast detection and localization is key to supporting cooperative multi-modal observation.

The Advanced Particle-astrophysics Telescope (APT) [4] mission concept is a proposed gamma-ray and cosmic-ray observatory that will orbit the Sun-Earth L_2 Lagrange Point, which avoids obstruction by the earth and ensures a nearly omnidirectional, 4π-steradian field of view (FoV). Its goals include prompt detection and localization of gamma-ray bursts (GRBs), which are early indicators of, e.g., neutron star and black hole mergers, blazar and magnetar flares, and supernovae. APT's localizations will permit follow-up observation of such events by optical telescopes, which typically have quite narrow FoVs. It is predicted to localize GRBs with better than 1° accuracy and computational latency under 1 second [5].

Nonetheless, APT may supplement point localization with more detailed *likelihood maps* that provide a spatial probability distribution over possible locations. This is especially important for its Antarctic Demonstrator (ADAPT), which has greater localization uncertainty. The most likely regions of the map can then be searched by fast-slewing optical telescopes to localize a source for subsequent observations.

Factors contributing to delays between detection and secondary observations include time to localize/map the transient, communication latency to follow-up telescopes (which is particularly challenging for space-based instruments like APT), and the time for these telescopes to physically search the sky as directed by a likelihood map. The computational pipeline that transforms raw sensor data into localizations (see Fig. 1) must therefore execute on the instrument (in space) [4] and meet significant size, weight, and power (SWaP) constraints.

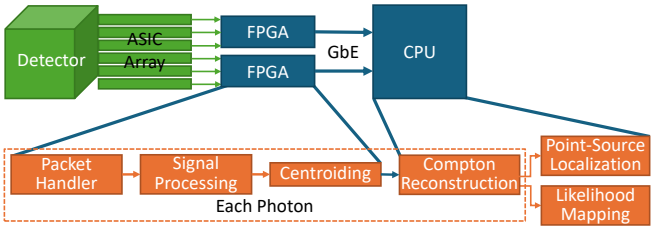


Fig. 1. ADAPT's computational pipeline for GRB localization.

Current Status. This presentation describes several important computational elements associated with APT and their ongoing development for ADAPT. §II outlines our progress on improved trajectory reconstruction for individual gamma rays, a process which uses FPGAs to read out and process signals from digitizer ASICs. From resulting estimates of the positions and energies of interactions in the detector, each gamma-ray photon's initial trajectory is constrained to a ring in the sky. §III details our method for point-source localization using the resulting collection of rings. While our original method based on least-squares refinement is effective in simulation for APT, ADAPT's smaller size and exposure to atmospheric background radiation give rise to greater uncertainty. To address this, we augment our iterative approach with machine learning. Furthermore, in §IV, we also plan to generate likelihood maps of the GRB's location; these will then be transmitted to optical telescopes for subsequent physical search.

Open Problems. Challenges persist in developing our computational pipeline for ADAPT, and there remain open problems related to deployment on the future APT instrument, which is larger, has more sensors, and demands higher readout rates. In §II, we motivate the need for efficient, FPGA-based noise-suppression and photon-counting algorithms. The future APT mission will need improved capabilities to transmit result data from several dozen FPGAs for aggregation on a CPU. In §III, we describe ADAPT's iterative ML-based reconstruction and localization loop, and connect it to existing work on concurrent, real-time execution of neural network models and recent work on IDK cascades [6]. In §IV, we discuss opportunities for

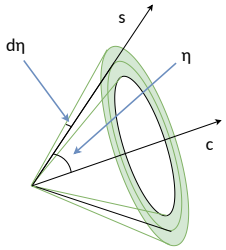


Fig. 2. A reconstructed Compton ring (from [11]).

tradeoffs between computational workload and precision of the probability maps that ADAPT will generate and consider the implications when portions of the spectral fitting and mapping algorithms are offloaded to GPU or FPGA accelerators, which may need to be shared with mission- or safety-critical recurrent tasks. For all of these problems, we welcome contributions and collaboration with other research groups in the real-time systems and operating systems communities.

Attribution. Many of the details of ADAPT's current status closely follow an invited paper at this year's CompSpace special session at the Computing Frontiers conference [7].

II. Trajectory Reconstruction

The upcoming ADAPT and proposed APT telescopes infer a GRB's direction by combining the trajectories of individual gamma-ray photons that interact with them. Here we give an overview of the instrument designs and methods to reconstruct gamma-ray trajectories from their raw sensor data.

The APT and ADAPT Detectors are constructed with layers of scintillating tiles that emit visible light when incoming gamma-ray photons scatter within them. This light is first captured by perpendicular arrays of wavelength-shifting (WLS) optical fibers that line the top and bottom surfaces of the tiles, then measured by silicon photomultipliers (SiPMs) placed at their ends [8], [9]. This overlay of 1-dimensional fiber arrays into a 2-dimensional mesh, with the relative position of the tile, allows us to resolve the 3-dimensional position $\mathbf{r} = (x, y, z)$ of each interaction. Additional SiPMs, placed around the edges of the tile layers, improve light collection and provide an estimate of the energy E deposited with each interaction.

Both will fly with onboard computational hardware, including waveform digitizer ASICs to sample and digitize analog signals from the SiPMs [10]. FPGAs process the ASIC data, reducing it to spatial coordinates and energy measurements. A processor builds the final set of interactions $(\mathbf{r}_i, E_i)$ associated with each gamma ray, then uses these to perform *Compton reconstruction*, constraining the gamma-ray photon's initial trajectory to a ring of the form illustrated in Fig. 2.

FPGA Pipeline Prototype. ADAPT's digitizer ASICs continuously sample the output voltages from the SiPM preamplifiers and store the values in a ring buffer with an analog memory depth of $\geq$256 entries. When a gamma ray is detected, all ASICs are triggered simultaneously to digitize and read out these values. Given the speed at which the gamma-ray photon moves within the detector, all of its interactions are captured in a single readout and cannot be temporally disambiguated.

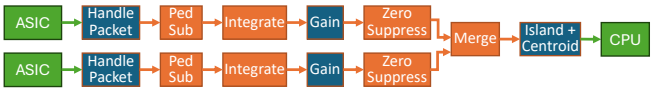


Fig. 3. ADAPT's FPGA pipeline.

We refer to the collection of data from a single gamma ray's interactions as an "event." To handle streaming event arrivals, a finite state machine (FSM) **packet handler** reads out the serial interface from the digitizer ASICs, providing flexibility to handle multiple types of waveform digitizers. ADAPT is expected to demonstrate the capabilities of two ASICs: the ALPHA [12], developed by collaborators at the University of Hawai'i at Mānoa, and the HDSoC from Nalu Scientific, both based on the TARGET ASIC [13].

Our pipeline first performs **pedestal subtraction**, removing the unique capacitive charge pedestal inherent to each of the ASIC's analog memory cells from the digitized readouts to yield the true sampled signal values. To infer the number of photons captured by each fiber or edge detector's SiPMs, a **signal integration** stage sums over digitized output values. To estimate the number of photons captured, a **gain correction** stage multiplies the integrated value by a per-channel fractional gain, then subtracts the expected *dark count* (spontaneous SiPM impulses due to thermally-generated electrons) for the duration of the integration window. **Zero-suppression** then sets sufficiently low photon counts to zero, under the assumption that these values may be caused by noise effects in the circuit or variation in dark counts.

Zero-suppressed photon counts from multiple ASICs are merged into a single array for each WLS fiber plane. To then derive interaction coordinates from each array, **island detection and centroiding** take the mean of WLS fiber positions, weighted by photon intensity, over islands of adjacent non-zero channels. The complete pipeline is illustrated in Fig. 3.

In [14], we described several HLS-based optimization techniques for an earlier version of this pipeline, achieving a throughput of $>2 \times 10^5$ events per second in simulation, even with a conservative 100 MHz system clock. New (or modified, in the case of island detection and centroiding) components since that work are marked in dark blue in Fig. 3.

Compton Reconstruction. Using a CPU, we next build a set of interactions, or *hits*, $(\mathbf{r_i}, E_i)$ for each individual event. From these, we reconstruct the gamma-ray photon's trajectory to constrain the burst's direction in a process referred to as *Compton reconstruction* [15]. For a gamma-ray photon that scatters following an interaction with an electron, the Compton law gives the relationship between the cosine η of its scattering angle and its energy before and after the interaction. Given the vector $\mathbf{c}$ between its two interactions and this η value, we can constrain the gamma ray's source direction $\mathbf{s}$ to a circle projected on the unit sphere, as illustrated in Fig. 2. Spatial and energy measurement errors spread the circle into a ring, or *annulus*; by propagation of error, we can estimate the uncertainty $d\eta$ in its radius [16], [17].

Reconstruction is challenging because the set of hits is temporally unordered due to the gamma ray's speed-of-light travel within the instrument. We therefore use the methods described

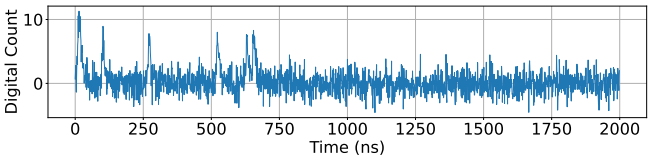


Fig. 4. 2 μs digitized waveform from an optical fiber that captured 5 photons.

in [15] to infer the most likely ordering of hits. We developed an accelerated branch-and-bound tree search algorithm [10], [16] that achieves a throughput of around 3×10^5 events per second when utilizing all 4 cores of ADAPT's 1.92 GHz Intel Atom 3845 CPU-based flight computer [18].

Open Challenge: Photon Counting. The energy deposited by a scattering gamma ray is low enough that only a handful of optical photons (typically $<$20) are transmitted by each optical fiber. For example, Fig. 4 shows a simulated waveform read out from a fiber; of the six peaks, five correspond to captured photons, while one is a dark count. Photon counts inferred from signal integration are thus inaccurate given the low SNR. Alternative FPGA-based techniques for noise suppression and photon counting — e.g., combinations of threshold-based methods [19], filters, and deep-learning based approaches — remain an open challenge. In particular, we would like to explore combinations of such methods, including synthesis of multiple kernels with different combinations of filters to be dynamically swapped out depending on resource availability and latency/throughput requirements.

Open Challenge: Data Rates. ADAPT will produce around 200 KB–1 MB of raw data for a single event; during a burst, several thousand events trigger every second. To sustain these high data rates, a hierarchy of FPGAs implements the preprocessing pipeline to handle thousands of readout channels. Ultimately, 13 FPGAs transmit reduced data to a CPU via Gigabit Ethernet. The larger APT instrument will produce $\sim$100$\times$ more data per event while triggering $\sim$10–100$\times$ faster. We anticipate that at least 60 FPGAs will send data, though perhaps not all of them for every event. Addressing this challenge within the constraints imposed by the space-based computational environment may require Time-Sensitive Networking (TSN). We will alternatively consider a dedicated FPGA with access to unified CPU memory or cache fabric that performs event building and dispatches Compton reconstruction workloads to CPU cores directly, building upon the principles of CAESAR [20].

III. Point-Source Localization

Localization aims to determine the most likely source direction for a GRB using the Compton rings from reconstruction.

Approach. Localization fixes a GRB's source direction $\mathbf{s}$ by "intersecting" multiple Compton rings. In principle, three rings suffice to fix $\mathbf{s}$; however, we must contend with both the uncertainties $d\eta$ of each ring and the fact that many observed rings ($\geq$50% for ADAPT) arise from unrelated, diffuse *background radiation*. As described in [5], ADAPT's localization operates in two stages. The first stage, *approximation*, selects the most likely direction $\mathbf{s_0}$ from a set of candidates. The second stage, *refinement*, uses an iterative least-squares approach to adjust $\mathbf{s_0}$ until it converges to a maximum-likelihood estimate $\mathbf{s}$.

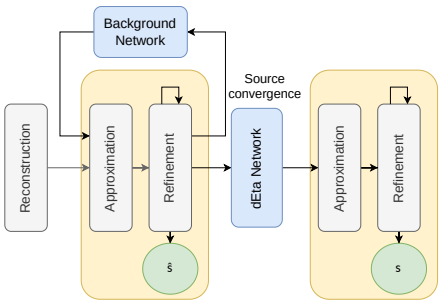


Fig. 5. ADAPT GRB localization pipeline (from [11]).

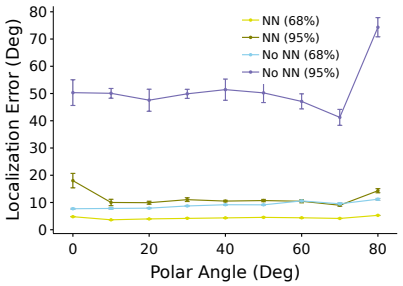


Fig. 6. Localization accuracy vs. polar angle with/without ML (from [11]).

Use of Machine Learning. Our recent work [11] supplements ADAPT's localization pipeline with machine learning inference designed to address background noise and uncertainty estimation. We introduce two multilayer perceptron models, the *background network* and the *dEta network*. The first classifies a Compton ring as originating from either the GRB or the background, allowing background suppression; the second more accurately estimates $d\eta$ for the surviving Compton rings. Each model takes as input the energy and position estimates of interactions that gave rise to the ring, along with an estimate of the GRB's polar angle with respect to the detector z-axis.

Using the polar angle as an input has proved essential to model accuracy, but it is not known a priori. We therefore iterate between the basic localization computation — which produces an estimated source direction $\hat{\mathbf{s}}$ — and then discarding any input rings classified as background given $\hat{\mathbf{s}}$. Once the estimate $\hat{\mathbf{s}}$ converges, we re-estimate $d\eta$ for all surviving rings and obtain $\mathbf{s}$ with a final run of the core algorithm.

Validation. Fig. 6 summarizes experiments from [11] that measure the impact of machine learning. These results measure the accuracy with which ADAPT can localize simulated GRBs. Using ML consistently improved localization accuracy, both in the common case (68%ile error) and especially for outliers (95%ile error). We also measured the computational cost on ADAPT's flight computer. Reconstruction and localization for a representative bright, short burst required $\sim$220 ms.

Open Problem Area: ML in RT As evidenced by the recent ML-RT Agenda (ECRTS'24 and '25) and WMC (RTSS '24) workshops, there is growing interest in using machine learning safely and predictably in real-time systems. Our iterative approach to ML-based GRB localization allows us to trade off between accuracy and efficiency, but exploring this

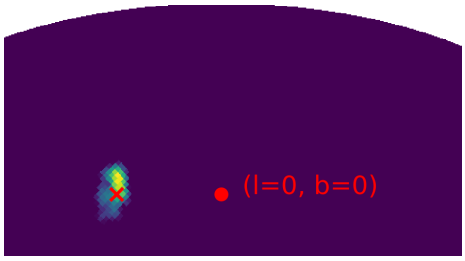


Fig. 7. Partial likelihood map for GRB 140329295 from Fermi GBM catalog, showing 99% containment region (from [7]). Lighter-colored pixels are more likely to contain the GRB source. The red cross denotes the actual source.

tradeoff space is an open problem, especially when the neural network models (which can exploit intra-task parallelism and vectorization) must run concurrently with each other and with safety-critical instrument- or satellite-control tasks. Several of these concerns have been outlined in a recent paper by Buttazzo [21]; we encourage members of the community with expertise in these areas to collaborate. Moreover, there may be opportunity to use IDK classifiers [6] for background rejection. Effective scheduling for sequences of multiple IDK classifiers (cascades) has garnered recent attention [22]–[25].

IV. Likelihood Mapping

The techniques of §III identify one likely direction for a GRB. To allow telescopes to search for an optical counterpart, we will also communicate a *likelihood map* over its possible location in the sky, as illustrated in Fig. 7.

Approach. Our mapping computation follows that of the `cosipy` library [26] released for the planned Compton Spectrometer and Imager (COSI) mission. Mapping, like point-source localization, begins with a set D of Compton rings, each with a center vector, radius, and measured energy E_m. These parameters define the *Compton data space* (CDS) of possible rings, which arise either from a source that appears for time Δt at location $\mathbf{s}$ or from background radiation.

The source and background are respectively characterized by an *instrument response* $R(\mathbf{s}, E_i)$ and a *background model* B. R and B each describe the expected number of rings in a given volume within the CDS observed during time interval Δt. R assumes that rings arise from photons of energy E_i arriving from a GRB of unit intensity in source direction $\mathbf{s}$, while B assumes that they arise from the background. R and B are derived empirically from extensive simulations.

To generate a likelihood map, we compare for each source direction $\mathbf{s}$ the hypothesis H_1^s that some portion of D arose from a GRB point source at $\mathbf{s}$, versus the null hypothesis H_0 that D arose from background alone. For H_1^s, the expected number of rings produced in a given volume of CDS is determined by $R(\mathbf{s}, E_i) \cdot \rho + B$, where ρ, the actual intensity of the GRB, is unknown *a priori* and so must be fit to maximize the likelihood. For both H_1^s and H_0, the observed number of rings within a given volume of CDS is assumed to be Poisson with that volume's expectation. The map score for direction $\mathbf{s}$ is the log-likelihood ratio of H_1^s versus H_0 given D.

Computational Cost. A key question is whether likelihood map generation can be done in real time for short-duration GRBs so that ADAPT and APT can coordinate with follow-up telescopes seeking optical counterparts that could fade within seconds. The computation is straightforward to parallelize across events, and the response R and background B are kept as arrays in DRAM for speed of access. We also implemented multiresolution mapping [27], [28], in which a map is produced at low resolution for the whole sky and then refined only in areas with likelihood scores high enough to plausibly contain the GRB. Selecting an appropriate granularity of subdivision to provide sufficient map precision to optical telescopes, while not introducing undue delays in the transmission of the map, remains an open problem.

We tested our implementation on 17 simulated GRBs from the 3rd COSI Data Challenge [29], using COSI's instrument response R and inferring the background B from three months of simulated observations in low-earth orbit. We generated likelihood maps with 12,288 HEALPix pixels ($\sim 2^\circ$ resolution), limiting output to the 90% containment region for the source. On an 8-core Arm Cortex-A78AE (Nvidia Jetson Orin NX), map generation consistently completed in under 200 ms. On 4 performance cores on ADAPT's Intel Core i7-13700TE CPU, it completes in under 100 ms. Further improvements may arise from porting our implementation to C++ and exploiting GPU or FPGA acceleration.

Challenges. Robust likelihood mapping for ADAPT and APT requires two further advances: real-time inference of GRB spectra, and efficient representation and inference of the response R. `Cosipy`'s spectral estimation fits a model to maximize the likelihood of the observed Compton rings, which requires nonlinear optimization. It also assumes the GRB's source location is known, resulting in a circular dependence with map generation. We will investigate simplified real-time fitting approaches that do not assume a known source location.

The size of the instrument response R — several gigabytes for even a low-resolution model — places large demands on memory and data bandwidth. Ongoing research includes compact machine-learning models that can approximate R. We will also investigate efficient active-learning approaches to infer R from fewer simulated photons.

V. Open Problems and Call for Collaboration

As we continue to develop the upcoming ADAPT instrument, work toward the proposed APT mission, and coordinate with ground-based optical telescopes for real-time follow-up observations of astrophysical transients, several open problems remain. These include (but are not limited to) development of efficient FPGA-based signal processing algorithms; real-time transmission of data from dozens of FPGAs to a CPU; computational offloading of ML models and likelihood mapping to GPU or FPGA accelerators; efficient representation and inference of the instrument response R; and runtime platforms to coordinate, schedule, and execute these with timing guarantees. The real-time systems community, with expertise in running latency-constrained applications atop SWaP-constrained hardware, is particularly suited to addressing these problems. We invite interested researchers to collaborate!

ACKNOWLEDGMENT

The authors would like to acknowledge the entire APT Collaboration (see https://adapt.physics.wustl.edu/). Support provided by NASA award 80NSSC21K1741, the McDonnell Center for the Space Sciences, the Peggy and Steve Fossett Foundation, and a Washington University OVCR seed grant.

REFERENCES

[1] P. Mészáros, D. B. Fox, C. Hanna, and K. Murase, "Multi-messenger astrophysics," *Nature Reviews Physics*, vol. 1, no. 10, pp. 585–599, 2019.

[2] K. Murase and I. Bartos, "High-energy multimessenger transient astrophysics," *Annu. Rev. Nucl. Sci.*, vol. 69, no. 1, pp. 477–506, 2019.

[3] R. Abbasi *et al.*, "All-sky search for transient astrophysical neutrino emission with 10 years of IceCube cascade events," *ApJ*, vol. 967, no. 1, p. 48, 2024.

[4] J. H. Buckley, J. Buhler, and R. D. Chamberlain, "The Advanced Particle-astrophysics Telescope (APT): Computation in space," in *Proc. of 21st Int'l Conference on Computing Frontiers Workshops and Special Sessions*, May 2024, pp. 122–127.

[5] Y. Htet *et al.*, "Prompt and Accurate GRB Source Localization Aboard the Advanced Particle Astrophysics Telescope (APT) and its Antarctic Demonstrator (ADAPT)," in *Proc. of 38th Int'l Cosmic Ray Conf.*, vol. 444, Jul. 2023, pp. 956:1–956:9.

[6] T. Trappenberg and A. Back, "A classification scheme for applications with ambiguous data," in *Proc. of IEEE-INNS-ENNS Int'l Joint Conference on Neural Networks*, vol. 6, Jul. 2000, pp. 296–301.

[7] D. Wang, Y. Htet, M. Sudvarg, R. Chamberlain, J. Buhler, and J. Buckley, "Coordinating instruments for multi-messenger astrophysics," in *Proc. of 22nd ACM Int'l Conference on Computing Frontiers Workshops and Special Sessions*, May 2025, pp. 213–218.

[8] W. Chen *et al.*, "The Advanced Particle-astrophysics Telescope: Simulation of the Instrument Performance for Gamma-Ray Detection," in *Proc. of 37th Int'l Cosmic Ray Conference*, vol. 395, 2021, pp. 590:1–590:9.

[9] ——, "Simulation of the Instrument Performance of the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope in the Presence of the MeV Background," in *Proc. of 38th Int'l Cosmic Ray Conf.*, vol. 444, Jul. 2023, pp. 841:1–841:9.

[10] M. Sudvarg *et al.*, "Front-End Computational Modeling and Design for the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope," in *Proc. of 38th Int'l Cosmic Ray Conf.*, vol. 444, Jul. 2023, pp. 764:1–764:9.

[11] Y. Htet *et al.*, "Machine learning aboard the ADAPT gamma-ray telescope," in *Proc. of Workshops of the Int'l Conf. on High Performance Computing, Network, Storage, and Analysis*, Nov. 2024, pp. 4–10.

[12] M. Kuwahara and G. S. Varner, "Design and Development of Advanced Low Power Hybrid Acquisition (ALPHA) ASIC for Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope (ADAPT)," in *Proc. of Nucl. Science Symp., Medical Imaging Conf. and Int'l Symp. on Room-Temp. Semicond. Detectors*, 2023.

[13] K. Bechtol *et al.*, "TARGET: A multi-channel digitizer chip for very-high-energy gamma-ray telescopes," *Astropart. Phys.*, vol. 36, no. 1, pp. 156–165, 2012.

[14] M. Sudvarg *et al.*, "HLS taking flight: Toward using high-level synthesis techniques in a space-borne instrument," in *Proc. of 21st ACM Int'l Conf. on Computing Frontiers*, 2024, pp. 115–125.

[15] S. Boggs and P. Jean, "Event reconstruction in high resolution Compton telescopes," *A&AS*, vol. 145, no. 2, pp. 311–321, 2000.

[16] M. Sudvarg *et al.*, "A Fast GRB Source Localization Pipeline for the Advanced Particle-astrophysics Telescope," in *Proc. of 37th Int'l Cosmic Ray Conf.*, vol. 395, Jul. 2021, pp. 588:1–588:9.

[17] Y. Htet *et al.*, "Localization of gamma-ray bursts in a balloon-borne telescope," in *Proc. of Workshops of the Int'l Conf. on High Performance Computing, Network, Storage, and Analysis*, Nov. 2023, pp. 395–398.

[18] M. Sudvarg *et al.*, "Parameterized workload adaptation for fork-join tasks with dynamic workloads and deadlines," in *Proc. of 29th Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, 2023, pp. 232–242.

[19] L. Huang, "New Signal Identification Algorithms for Enhanced Gamma-Ray Burst Detection in the Advanced Particle-Astrophysics Telescope," Master's thesis, Dept. of Electrical and Systems Engineering, Washington University in St. Louis, May 2025.

[20] S. Roozkhosh, D. Hoornaert, and R. Mancuso, "CAESAR: Coherence-aided elective and seamless alternative routing via on-chip FPGA," in *Proc. of Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 356–369.

[21] G. Buttazzo, "Toward predictable AI-based real-time systems," *Real-Time Systems*, pp. 1–16, 2025.

[22] T. Abdelzaher, K. Agrawal, S. Baruah, A. Burns, R. I. Davis, Z. Guo, and Y. Hu, "Scheduling IDK classifiers with arbitrary dependences to minimize the expected time to successful classification," *Real-Time Systems*, vol. 59, no. 3, pp. 348–407, 2023.

[23] S. Baruah, A. Burns, R. I. Davis, and Y. Wu, "Optimally ordering IDK classifiers subject to deadlines," *Real-Time Systems*, vol. 59, no. 1, pp. 1–34, 2023.

[24] S. Baruah, A. Burns, and R. I. Davis, "Optimal synthesis of robust IDK classifier cascades," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 5s, pp. 150:1–150:26, 2023.

[25] S. Baruah, I. Bate, A. Burns, and R. I. Davis, "Optimal synthesis of fault-tolerant IDK cascades for real-time classification," in *Proc. of 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2024, pp. 29–41.

[26] I. Martinez *et al.*, "The cosipy library: COSI's high-level analysis software," in *Proc. of 38th Int'l Cosmic Ray Conf.*, vol. 444, 2023, pp. 858:1–858:8.

[27] P. Fernique, M. Allen, T. Boch, A. Oberto, F. Pineau, D. Durand, C. Bot, L. Cambrésy, S. Derriere, F. Genova *et al.*, "Hierarchical progressive surveys: Multi-resolution HEALPix data structures for astronomical images, catalogues, and 3-dimensional data cubes," *Astronomy & Astrophysics*, vol. 578, p. A114, 2015.

[28] L. P. Singer and L. R. Price, "Rapid Bayesian position reconstruction for gravitational-wave transients," *Phys. Rev. D*, vol. 93, p. 024013, Jan 2016.

[29] The Compton Spectrometer and Imager (COSI) Collaboration, "COSI Data Challenges," Apr. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15126188

OSPERT 2025 Program

Tuesday, July 8, 2025

8:00 – 9:00	Registration
9:00 – 10:00	**Session 1: Opening Remarks and Keynote** Keynote: Challenges and Experiences Building a Software-Defined Vehicle Management System *Professor Rich West*
10:00 – 10:30	Coffee Break
10:30 – 12:00	**Session 2: Technical Papers I**
	UltraScale+ SpinalHDL Wrapper: Streamlining Ideas to Bitstream on UltraScale+ platforms *Denis Hoornaert, Giulio Corradi, Renato Mancuso, Marco Caccamo*
	Tintin: PMU Scheduling to Minimize Uncertainty *Marion Sudvarg, Ao Li, Sanjoy Baruah, Chris Gill, Ning Zhang*
	Towards a Linux-based Unikernel for Resource-Constrained Embedded Systems *Yoshifumi Shu, Yutaka Matsubara, Yixiao Li, Hiroaki Takada*
12:00 – 13:30	Lunch
13:30 – 14:30	**Session 3: Case Studies**
	Compute Kernels as Moldable Tasks: Towards Real-Time Gang Scheduling in GPUs *Attilio Discepoli, Mathias Louis Huygen, Antonio Paolillo*
	SentryRT-1: A Case Study in Evaluating Real-Time Linux for Safety-Critical Robotic Perception *Yuwen Shen, Jorrit Vander Mynsbrugge, Nima Roshandel, Robin Bouchez, Hamed FirouziPouyaei, Constantin Scholz, Hoang-Long Cao, Bram Vanderborght, Wouter Joosen, Antonio Paolillo*
14:30 – 15:30	**Session 4: Technical Papers II**
	IRx: RTOS-Aware Abstract Interpretation using an LLVM-based Interpreter *Andreas Kässens, Vitali Fendel, Daniel Lohmann*
	Bounded Resource Reclamation *Viktor Reusch*
15:30 – 16:00	Coffee Break
16:00 – 17:20	**Session 5: Demos, Tutorials, and Calls**
	Real-Time Virtualization on Heterogeneous MPSoCs: A Hands-On Tutorial with Jailhouse and Omnivisor *Daniele Ottaviano*
	Call for Collaboration: Contributing to Multi-Messenger Astrophysics *Marion Sudvarg, Ye Htet, Roger Chamberlain, Jeremy Buhler, James Buckley*
17:20 – 17:30	**Closing Remarks and Best Paper Award**
17:30 – 19:00	ECRTS First-timer Reception

Wednesday, July 9th – Friday, July 11th, 2025

ECRTS main conference.