PROCEEDINGS OF

OSPERT 2026

The 20th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 7th, 2026 in Lund, Sweden

in conjunction with

The 38th Euromicro Conference on Real-Time Systems
July 7–10, 2026, Lund, Sweden

Editors:
Marion Sudvarg
Antonio Paolillo

Contents

Message from the Chairs

Welcome to OSPERT'26, the 20th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. This year, 7 contributions — technical papers, open problems, and experiment reports — 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'26 will open with a keynote by Elad Lahav, Lead Architect at QNX. He will discuss the tension between throughput and low latency in operating system design on modern hardware, drawing on his experience building the QNX microkernel for automotive and robotics use cases. This will be followed by three sessions with presentations of technical papers, open problems, and experiment reports.

OSPERT'26 received 9 submissions. After a competitive review process, 7 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 13 experts, for volunteering their time and effort to provide useful feedback to the authors, and of course to all the authors for their contributions and hard work.

OSPERT'26 would not have been possible without the support of many people. The first thanks are due to Angeliki Kritikakou, Johan Eker, Sebastian Altmeyer, and the whole ECRTS organizing team for entrusting us with organizing OSPERT, and for their continued support of the workshop. To Tanya Amert and Joshua Bakita, thank you for stepping in to chair a session. 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,

Marion Sudvarg
Washington University in St. Louis
The United States

Antonio Paolillo
Vrije Universiteit Brussel
Belgium

Program Committee

An Zou *Shanghai Jiao Tong University*
Andrea Bastoni *Technical University of Munich*
Bohdan Trach *Huawei Dresden Research Center*
Bryan Ward *Vanderbilt University*
Denis Hoornaert *Axelera AI*
Jinwen Wang *University of Texas at Dallas*
Joshua Bakita *Mohamed bin Zayed University of Artificial Intelligence*
Michael Roitzsch *Barkhausen Institut*
Peter Wägemann *FAU Erlangen-Nürnberg*
Phani Kishore Gadepalli *VMware*
Tam Chantem *Virginia Tech*
Takuya Azumi *Saitama University*
Tanmaya Mishra *Analog Devices*

Keynote Talk

Throughput vs Low Latency on Modern Hardware: an OS Developer Perspective

Elad Lahav
Lead Architect, QNX

Current use cases for the QNX operating system, especially in automotive and robotics, require both high throughput and low latency. Attempting to achieve both goals often leads to conflicts and compromises in the design of the operating system. Matters are further complicated by the characteristics of modern hardware, which favors parallel high-throughput on average to single-core performance and to latency. Some of these problems are unique to the design of QNX as a micro-kernel, low-latency, general-purpose operating system, but for the most part are of interest to anyone trying to build real-time systems on current and future hardware.

Elad Lahav has been with QNX since 2012, spending the majority of this time with the kernel team. Prior to that, he led the kernel team for the BlackBerry OS. In 2019, he initiated the design of a new QNX microkernel, which was released as part of QNX 8 in December 2023. Since 2024, he has served as Lead Architect at QNX. He holds a Master of Mathematics degree from the University of Waterloo.

Dolia: an MPU-backed Framework for Secure Multitenancy on Embedded MCUs

Julien Gourgue
UCLouvain
Belgium
julien.gourgue@uclouvain.be

Cristel Pelsser
UCLouvain
Belgium
cristel.pelsser@uclouvain.be

Ramin Sadre
UCLouvain
Belgium
ramin.sadre@uclouvain.be

***Abstract*—Modern microcontroller units (MCUs) are increasingly powerful, yet often dedicated to a single lightweight application that underutilises their capabilities. Sharing a device among multiple independent stakeholders improves utilisation, but requires strong isolation between tenants to prevent interference and unauthorized access. Additionally, the system must support post-deployment reconfigurability and preserve soft real-time guarantees. Existing solutions rely on virtualization through language-level sandboxing, which provides isolation and heterogeneity but introduces runtime overhead that is incompatible with the real-time constraints of resource-constrained devices. This paper presents a framework built on top of Zephyr RTOS that enables secure multitenancy on MCUs. The framework leverages PMP-based hardware memory protection for tenant isolation, dynamically loadable applications via Zephyr's LLEXT subsystem with mTLS-authenticated connection for post-deployment reconfigurability, and a condition-based publisher-subscriber model that minimises kernel-to-user crossings to preserve real-time behaviour. Evaluated on an ESP32-C6 RISC-V SoC, results show a $2.5\times$ overhead for kernel-to-user transitions compared to kernel-to-kernel switches, and approximately $4\times$ faster execution than WebAssembly AOT. A connected vehicle use case demonstrates that soft real-time deadlines are preserved even under adversarial CPU load.**

***Index Terms*—IoT, real-time, security, isolation, multitenancy**

I. Introduction

The growing proliferation of Microcontroller Units (MCUs) across diverse sectors, including smart cities, industrial automation, and connected vehicles, is driving demand for increasingly capable embedded platforms. Modern MCUs are becoming significantly more powerful, featuring multicore architectures, hardware memory protection, and larger flash memories, among other advances. These improvements enable them to host workloads that previously required multiple dedicated devices [1].

However, in systems such as automotive embedded platforms, where functionality is provided by different software suppliers, consolidating multiple functions onto shared hardware introduces several challenges. First, applications originating from different stakeholders must be strictly isolated from one another and from the operating system to prevent interference and unauthorized access. Second, the system must support post-deployment reconfiguration, whether for bug fixes, feature updates, or the addition of new applications, without requiring manual reflashing. Third, applications with real-time constraints must execute directly on the device, since offloading computation to remote servers introduces latency and reliability concerns.

Existing work on implementing secure multitenancy for embedded hardware has largely relied on language-based sandboxing, recently particularly through the use of WebAssembly combined with interpretation, Just-In-Time (JIT) compilation, or Ahead-Of-Time (AOT) compilation [2]–[6]. However, such sandboxing introduces performance overhead and restricts applications from directly accessing hardware capabilities that are essential for real-time use cases [7].

This work addresses multitenancy by presenting a framework built on top of Zephyr RTOS [8] that enables the dynamic deployment of isolated, untrusted native applications on resource-constrained MCUs while preserving timing guarantees for soft real-time workloads. The framework leverages Zephyr's user mode and the Linkable Loadable Extension (LLEXT) subsystem, which supports loading native object files at runtime, to provide a secure and flexible environment for concurrently executing multiple applications. The framework targets MCUs equipped with a hardware Memory Protection Unit (MPU) and operating with a single shared kernel. In contrast to existing approaches, the framework relies on native hardware-enforced memory protection rather than language-based sandboxing, and operates as a layer on top of Zephyr.

The main contributions of this paper are as follows:

- We design and implement a framework that enables native, memory-protected multitenancy on MCUs, allowing multiple untrusted native applications to co-exist on the same device without interference.
- We develop a secure loading mechanism that ensures only authenticated applications are deployed and executed on the device.
- We implement a publish-subscribe model for sensor data distribution, enabling applications to receive sensor updates based on specified conditions while preserving isolation and minimizing kernel-user communication overhead.
- We evaluate the framework on an ESP32-C6 MCU, measuring overhead and scheduling latency, and demonstrate its real-time properties through a vehicle use case.

The source code of our implementation is publicly available at https://github.com/juliengourgue/Dolia.

The rest of the paper is organized as follows. Section II reviews related work. Section III provides background information on Zephyr RTOS. Section IV describes the design and implementation of our framework. Section V presents evaluation results. Finally, Section VI concludes the paper and outlines future work directions.

II. Related Work

In recent years, isolated applications executing directly on resource-constrained devices have gained significant attention. However, existing approaches for enabling multitenancy on MCUs mainly rely on virtualization through language-level sandboxing.

WebAssembly has emerged as the dominant technology in this area. Ramesh et al. [2] propose Silverline, a lightweight virtualization framework for distributed real-time systems based on WebAssembly deployment across the edge-cloud continuum. Skrivankova et al. [4] introduce KaOS, a distributed control platform for smart building using WebAssembly containers on ESP32 devices. No experimental evaluation is provided, but the authors report a sub-millisecond execution latency overhead as a preliminary result. Zandberg et al. [3] propose femto-containers based on rBPF and report execution overheads of 37× for WASM and 77× for rBPF compared to native execution. Similarly, Has et al. [5] show that native execution remains superior in performance and energy efficiency on ESP32-C6 devices. Coppolino et al. [6] propose WASMBOX, a WASM-based sandboxing solution combining a patched WASI interface with a Trusted Execution Environment to prevent sandbox escapes.

Other approaches include Polyglot CerberOS [9], [10], a multilingual resource-secure operating system based on a lightweight JVM, and MicroEJ [11], which provides a virtual execution environment for hardware-independent embedded applications.

While these approaches provide portability and isolation, they introduce runtime overhead and restrict direct interaction with hardware peripherals such as sensors and actuators [7]. An alternative approach is native isolation through hardware memory protection. The Tock operating system [12] leverages the MPU to isolate unprivileged processes with lower overhead than VM-based approaches.

In contrast, our framework relies on native MPU enforcement while operating as a layer on top of the popular Zephyr RTOS, enabling multitenancy on existing Zephyr-based systems without requiring migration to a dedicated operating system.

III. Background

a) Zephyr RTOS: Zephyr [8] is a lightweight, scalable, real-time operating system (RTOS) designed for resource-constrained devices across multiple architectures. It was selected for this work due to the following characteristics:

- *Open source:* Zephyr is released under the Apache 2.0 license, enabling unrestricted academic and industrial use.
- *Wide hardware support:* Zephyr supports more than 1000 boards, covering a broad range of hardware platforms and architectures.
- *Active community:* Zephyr benefits from an active community, with regular updates and contributions.
- *Rich ecosystem:* Zephyr provides a large set of libraries, drivers, and tools, including Memory Protection Unit (MPU) support, user/kernel space isolation, the LLEXT subsystem for dynamic loading, and a priority-based preemptive scheduler with Earliest Deadline First (EDF) support, suitable for real-time applications.

b) Memory Protection Units and Zephyr User Mode: On traditional computers, memory protection is primarily achieved through a Memory Management Unit (MMU), which translates virtual addresses into physical ones. MMUs are however generally absent from the MCU ecosystem due to cost, power consumption, and latency constraints. Memory isolation therefore relies on Memory Protection Units (MPUs), which provide a simpler mechanism by defining fixed memory regions with configurable access permissions. MPUs are found in both the ARM and RISC-V ecosystems, where the Armv8-M architecture [13] may implement an MPU, while RISC-V provides a similar but distinct mechanism through its Physical Memory Protection (PMP) [14]. In both cases, the hardware exposes a set of registers that define memory regions and their associated permissions (read, write, execute). The operating system may reconfigure these registers during a context switch so that each task can only access the memory regions assigned to it.

Zephyr provides within its kernel the possibility to enable the *User Mode* [15] functionality, which allows threads to be executed at a reduced privilege level. When such a thread issues a system call, the processor switches to kernel mode, verifies that the thread is permitted to perform the requested operation, executes it, and returns control to the caller. At each context switch, the PMP registers are reconfigured to enforce the memory access rights of the incoming thread, ensuring that no thread can access memory outside its assigned regions. Because Zephyr treats user-mode threads as untrusted, they are isolated both from one another and from the kernel. Our framework builds on this mechanism to isolate untrusted applications.

c) Linkable Loadable Extension (LLEXT): To support dynamic deployment of applications at runtime, our framework employs the Linkable Loadable Extension (LLEXT) [16] subsystem, which provides an API for extending application functionality without re-flashing or rebooting the device. Extensions are precompiled into the ELF binary format.

The LLEXT loader is responsible for allocating memory for extensions, applying relocations, resolving imported symbols, and linking extensions against exported Zephyr kernel APIs. Because symbol resolution depends on kernel-exported interfaces, extensions must be compiled against the exact kernel headers and configuration of the running Zephyr image.

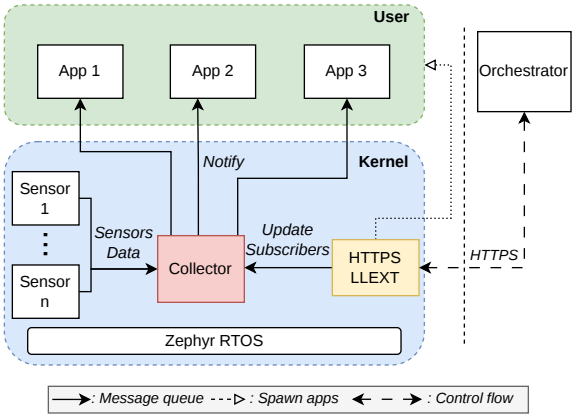


Fig. 1: Overview of the framework architecture. The sensor threads, collector, HTTPS/LLEXT worker, and user applications run on the MCU. The orchestrator runs on a remote server.

IV. Framework Design

Our framework is built on top of the Zephyr RTOS [8], leveraging its user mode and the Linkable Loadable Extension (LLEXT) subsystem to provide memory isolation on devices equipped with MPU hardware, authenticated deployment, and condition-based sensor sharing. The threat model assumes a network-level adversary capable of intercepting or replaying communications, as well as a malicious third-party application attempting to access unauthorized memory regions or sensor data. The device hardware and the orchestrator (see below) are considered trusted.

The framework consists of several components. On the device side, an HTTPS/LLEXT worker retrieves user applications from the remote server and deploys them as user threads. In parallel, a set of dedicated sensor threads continuously acquires data and forwards it to the collector, which evaluates predefined conditions and subsequently notifies the corresponding user applications. Finally, the orchestrator, implemented as a remote server, is responsible for managing both devices and deployed applications. The overall architecture is illustrated in Fig. 1 and is described in detail in the following sections.

A. HTTPS/LLEXT Worker

The HTTPS/LLEXT worker is responsible for managing the secure deployment of applications on the device. It establishes a mutually authenticated TLS (mTLS) [17] connection with the orchestrator, ensuring both the legitimacy of the device and the authenticity of the server before any application is transferred. Since both the device and the orchestrator are owned by the same entity, a private Certificate Authority (CA) is used to issue and manage certificates for both parties, avoiding the overhead and complexity of a public CA.

The worker follows a pull-based model, where the device periodically contacts the orchestrator to check for updates. This design avoids the need for the device to expose an open incoming port and eliminates the overhead of maintaining a persistent connection on a resource-constrained device.

The device periodically issues HTTPS GET requests over mTLS to the orchestrator to obtain application manifests. A manifest is a JSON file describing the application that should be deployed on the device, along with its sensor conditions (see Section IV-B), and the URL from which the corresponding ELF file can be downloaded. An example manifest is shown in Fig. 2. Our implementation makes use of Zephyr's JSON parsing model, which uses statically defined structures to reduce memory usage and avoid dynamic allocation, making variable-length application lists difficult to transfer efficiently. Instead, the orchestrator is responsible for presenting one manifest per request. At each request, the orchestrator selects which application manifest to serve, allowing different applications to be deployed across successive pull cycles.

Upon reception of a manifest, the device parses it and, if the application is not already deployed, downloads the corresponding ELF file. The worker then deploys the application using the LLEXT API, loading it into memory and initializing it as a user thread within Zephyr. Verification of the ELF file integrity and the granting of sensor and actuator access rights to an application prior to loading is delegated to the orchestrator. Since application deployment is an infrequent operation, the overhead of this process is considered acceptable. Full dynamic application lifecycle management, including hot replacement and removal, is left as future work.

B. Sensor and actuator access

MCUs are commonly used with sensors that collect data from their environment. Since user applications are untrusted, direct hardware access must be prevented: a malicious application could corrupt sensor state or skew sampling data, affecting other tenants. Two categories of sensors are considered: polling sensors, which monitor physical quantities such as temperature or humidity at a fixed sampling rate, and interrupt-driven sensors, which notify the system when a specific event occurs, such as a button press or motion detection.

To handle both categories while minimizing kernel-user context switch overhead, a publisher-subscriber model is adopted. Since transitions between kernel and user threads require reconfiguring PMP registers and performing privilege level checks, notifying user threads only when their condition is satisfied reduces unnecessary context switches compared to a direct polling approach. Polling sensors are managed by a dedicated per-sensor thread that forwards data to the central collector thread, while interrupt-driven sensors forward data directly from their interrupt handler to the collector.

The collector is responsible for maintaining a list of subscribers, which are the user threads that have expressed interest in receiving data from specific sensors if a condition is met. Conditions are specified at deployment time: the HTTPS/LLEXT worker extracts them from the manifest. There are four types of conditions: less than (LT), greater than (GT),

```
{
  "name": "Ext1",
  "path": "/ext1/1.0.0/",
  "conditions": [
    {
      "sensorid": "6",
      "op": "GT",            "threshold_low": "0",
      "has_deadline": false, "deadline_ns": 0
    },
    {
      "sensorid": "1",
      "op": "GT",            "threshold_low": "1",
      "has_deadline": true,  "deadline_ns": 1500000
    }
  ]
}
```

Fig. 2: Example manifest with two sensor conditions: the first triggers on button press, the second notifies when the distance sensor exceeds 1 m, with a 1.5 ms deadline.

equal to (EQ), or within a range (RANGE). An example of such conditions is shown in Fig. 2.

To send data to the collector, the sensors use a shared message queue. Upon reception of a new sensor value, the collector consults the subscriber list to determine which user threads should be notified based on their specified conditions. If a user thread's condition is satisfied, the collector forwards the corresponding data to that thread through a per-thread message queue, allowing it to react accordingly. Multiple applications can subscribe to the same sensor with different or identical conditions, and each will receive its own notification when its condition is satisfied. This architecture ensures hardware abstraction for untrusted applications while reducing kernel-user context switches, as user threads are notified only when their condition is satisfied.

Access to actuators, typically through write operations to MCU output registers, is performed via system calls that enforce authorization checks for the calling user thread. As with sensor access, the authorization decision is made offline by the orchestrator by allowing the application to be deployed.

C. Scheduling

Scheduling determines which thread runs, when, and for how long. In Zephyr's execution model, each thread is assigned a priority. Sensor threads and the collector run as high-priority kernel threads, ensuring timely data acquisition and notification delivery. The HTTPS/LLEXT worker runs at the lowest priority to avoid interfering with time-critical operations. User threads run at an intermediate priority level.

All user threads share the same base priority. When no deadline is specified, time-sliced preemptive scheduling with a 1 ms slice ensures a fair distribution of CPU time. When a deadline is specified as part of the sensor condition in the manifest (see Fig. 2), the collector sets the deadline on the corresponding user thread upon notification, and the scheduler applies an EDF policy, ensuring that the most time-critical thread executes first. User threads without a deadline are assigned an infinite deadline, ensuring they are always deprioritized relative to deadline-bearing threads.

Since arbitrary deadlines could cause starvation, the orchestrator is responsible for validating that all assigned deadlines are feasible. It also ensures that the combined workload, based on deadlines and sensor sampling rates, remains schedulable before deploying applications on a device.

D. Orchestrator

The orchestrator is a remote server responsible for managing one or multiple devices. It can be deployed at the edge of the device network or in the cloud, and communicates with the HTTPS/LLEXT worker running on the devices. As described in Section IV-A, one of its duties is to offer the manifests of new or updated applications to the respective devices.

The orchestrator can perform various pre-deployment validation tasks and deployment decisions (hence its name), although those are optional and depend on the device owner's device usage and management policies. Given a set of application manifests containing the sensor conditions and corresponding deadlines, the required actuators, and additional deployment constraints, the orchestrator verifies that the target device is equipped with the required sensors and actuators, that the respective application suppliers are authorized to access the requested resources, and that the combined deadlines of all applications deployed on a device are schedulable. The orchestrator could also check the integrity of the ELF files. Since these tasks are not directly linked to Zephyr and the embedded device, we consider them as outside the scope of this paper and do not discuss them further.

V. Evaluation

A. Methodology

All benchmarks are conducted on an ESP32-C6 [18], a RISC-V based MCU from Espressif Systems equipped with a PMP unit, running Zephyr 4.3.99 with Zephyr SDK 0.17.4. The first two benchmarks characterize the fundamental overhead introduced by the framework: the cost of syscalls from user threads, which require a kernel-to-user context switch, and the end-to-end latency of the interrupt-driven pipeline. The third benchmark compares the execution time of a user and kernel thread against a WebAssembly-based alternative. Finally, a vehicle use case provides a realistic scenario to validate the soft real-time properties of the framework.

Two timing measurement approaches are used depending on the context. For kernel thread measurements, Zephyr's `k_cycle_get_32()` function provides high-resolution cycle-accurate measurements. For measurements inside user threads, where this function is not accessible from unprivileged mode, an external logic analyzer based on a Raspberry Pi Pico 2 running *logicanalyser* [19] is used instead, measuring GPIO toggling latency directly on the hardware.

B. System call overhead (User vs Kernel)

Since user threads operate at a reduced privilege level, certain instructions are not directly available to them and must be performed through system calls. When a user thread issues a system call, it is preempted and execution transfers to the kernel, which identifies the requested system call, verifies that the operation is permitted for the calling user thread, and then

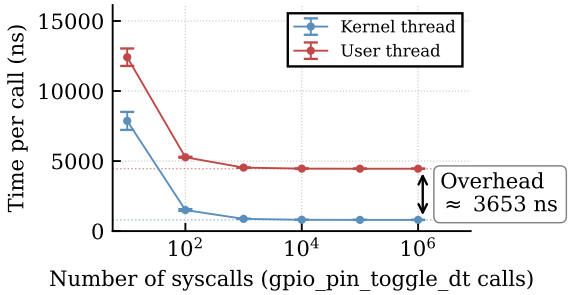


Fig. 3: Overhead of a GPIO syscall issued from a user thread compared to a kernel thread.

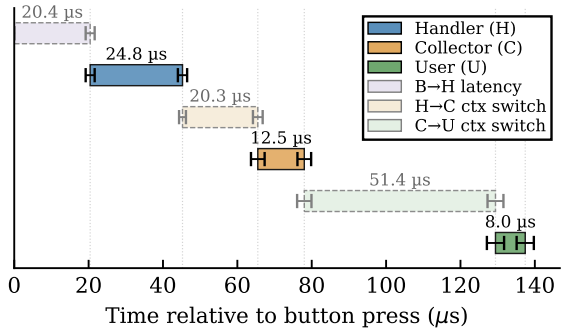


Fig. 4: End-to-end latency decomposition of the interrupt-driven pipeline, from button press to user thread execution. B: button press, H: interrupt handler, C: collector thread, U: user thread, ctx: context switch.

invokes the corresponding implementation before returning to the caller. In contrast, a kernel thread issuing the same system call bypasses the permission verification and directly invokes the implementation. These additional verification steps introduce an overhead specific to user threads.

To measure the system call overhead, a simple benchmark program is implemented in which a thread toggles a GPIO pin N times using the `gpio_pin_toggle_dt()` system call. As direct access to the CPU hardware cycle counter is not available from a user thread, the measurement is performed by wrapping the entire thread execution between two `k_cycle_get_32()` calls to compute the elapsed time. This approach however introduces a fixed cost associated with the start of the thread. To amortize this bias, we use very large N such that the thread creation overhead becomes negligible compared to the system call overhead. The results are presented in Fig. 3 for different N. Points represent the mean time per call across repeated measurements, and error bars denote ± 1 standard deviation. The measurements stabilize for $N > 1000$, revealing that the user thread incurs a system call cost approximately $5.5\times$ higher than that of the kernel thread.

C. Interrupt latency decomposition

This benchmark measures the end-to-end latency of the interrupt-driven pipeline, from physical button press to user thread execution. The goal is to quantify the cost introduced by our framework, in particular by the collector thread, for interrupt-driven data processing applications. The measurement is decomposed into three phases, each delimited by GPIO pins raised and lowered at phase boundaries using the logic analyzer: (i) the latency from button press to interrupt handler execution, (ii) the collector processing time from message reception to user notification, and (iii) the application-side handling time. Results are presented in Fig. 4, where the time origin corresponds to the physical button press. Each bar spans from the mean segment start time to the mean segment end time across repeated measurements. Error bars indicate ± 1 standard deviation of the segment start and end times.

We observe that the context switch between the interrupt handler and the collector thread is noticeably shorter than the one between the collector and the user thread. This difference can be explained by the additional operations required when switching from kernel mode to user mode. In particular, the kernel must verify thread permissions and reconfigure the PMP registers to enforce the memory access rights associated with the incoming user thread. These operations are not required during the kernel-to-kernel context switch between the interrupt handler and the collector. Concretely, the kernel-to-kernel context switch takes approximately 20 µs, while the kernel-to-user switch takes approximately 51 µs.

Overall, the measurements show that the additional latency introduced by the framework mainly originates from the transition to user space rather than from the collector itself. These results justify the use of a publisher/subscriber model. Without such a mechanism, all sensor events would need to be forwarded to the user threads regardless of whether the applications are interested in them; the large number of kernel-to-user context switches and preemptions would introduce a significantly higher overhead compared to our solution. Of course, our solution incurs additional costs compared to an implementation in which *all* threads execute in kernel mode. However, this overhead is the trade-off required to provide thread isolation and memory protection between applications.

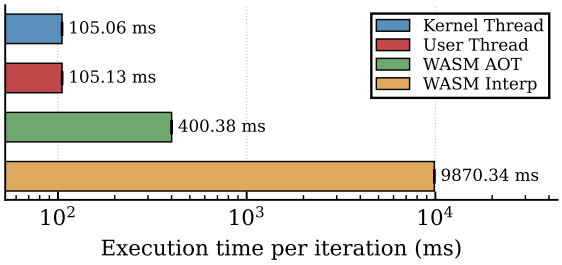


Fig. 5: Per-iteration execution time (bubble sort, 2048 elements, 10 runs). Standard deviation below 0.02 ms for all configurations.

D. WebAssembly Micro Runtime comparison

As already discussed, WebAssembly has been widely explored as an isolation mechanism for embedded systems [2]–[6]. To quantify the overhead of WASM-based isolation, we

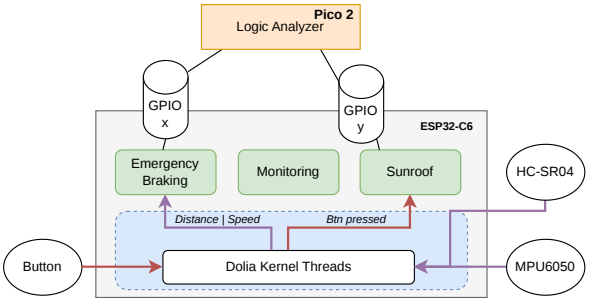


Fig. 6: Vehicle use case setup. The Automated Emergency Braking System (AEBS) and sunroof applications toggle a GPIO pin upon task completion, measured via a logic analyzer.

use the WebAssembly Micro Runtime (WAMR) [20] as a representative implementation and execute a bubble sort on an array of 2048 elements repeated 10 times, under four configurations: a kernel thread, a user thread, an interpreted WebAssembly module, and an AOT WebAssembly module. Bubble sort was chosen as it involves frequent memory accesses and conditional branches, which stress both WASM bounds checking and interpreter dispatch and is representative of a heavy data processing and analysis workload. Execution time was measured excluding runtime initialization and module loading for the WebAssembly configurations.

The results are presented in Fig. 5. Bars represent the mean execution time per iteration across repeated measurements, while error bars indicate ± 1 standard deviation. The observed variability is very small, with standard deviations below $0.02, \mathrm{ms}$ for all configurations, making the error bars barely distinguishable at the scale of the plot. The kernel and user thread configurations complete an iteration in 105 ms on average, with similar performance since no system calls are involved. The WebAssembly module running in interpreter mode requires 9870 ms, representing a $94\times$ overhead, while WASM AOT reduces this to 400 ms, still representing a $3.8\times$ overhead over native execution. While AOT significantly narrows the performance gap, WebAssembly's isolation model depends on a software sandboxing layer that relies on the correctness of the WASM runtime for security guarantees. As shown by recent work [6], WASM sandboxing alone is insufficient against malicious tenants attempting to escape the sandbox via arbitrary host calls. Furthermore, WASM AOT bounds checking and syscall proxying introduce additional overhead for sensor-driven workloads [7]. Our approach relies solely on hardware MPU enforcement, avoiding any dependency on software runtime correctness, at the cost of requiring hardware with MPU support.

E. Real-world use case

To evaluate the framework under realistic conditions, we consider a vehicle scenario in which two independent third-party applications from different suppliers are deployed on the same MCU. This consolidation reduces both hardware cost and the number of physical components in the vehicle. The

Application	Sensors	Deadline
Emergency braking	HC-SR04, MPU-6050	1.5 ms
Sunroof control	Button	150 ms
Monitoring	–	None

TABLE I: Connected vehicle use case application parameters.

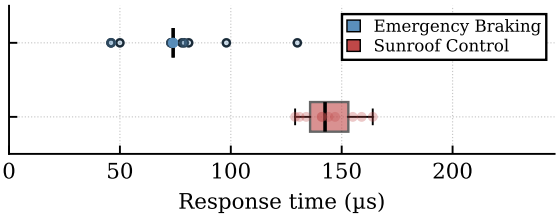


Fig. 7: Response time of both applications under adversarial CPU load. Deadlines are 1.5 ms (braking) and 150 ms (sunroof).

first application implements an Automated Emergency Braking System (AEBS) that computes a Time-To-Collision (TTC) from distance and accelerometer data, triggering an alert when the TTC falls below a threshold. The second application handles sunroof control via an interrupt-driven button press. A third monitoring application runs opportunistically when the CPU is idle. The setup is shown in Fig. 6. The application parameters are summarized in Table I. The emergency braking and sunroof control applications toggle a GPIO pin upon task completion, simulating actuator interaction while simultaneously allowing deadline compliance to be measured via the logic analyzer. While a production AEBS would rely on higher-frequency sensors, this scenario serves as a representative workload to demonstrate the real-time scheduling properties of the framework.

All three user threads share the same base priority. When the collector receives sensor data that triggers a user notification, it assigns a deadline to the corresponding user task. Pending user tasks are then scheduled according to the EDF policy. A time slice of 1 ms is configured to ensure that the monitoring application, which continuously executes busy-wait loops to emulate a worst-case CPU load, cannot monopolize the processor for more than 1 ms at a time. The goal of this experiment is to verify that the emergency braking and sunroof applications always meet their respective deadlines under this adversarial condition.

Two measurement campaigns are conducted. In the first, the emergency braking application runs alongside the monitoring application, and the deadline compliance of the TTC computation is measured. In the second, the sunroof application is triggered by a button press while both the emergency braking and monitoring applications run concurrently, and the deadline compliance of the sunroof handler is measured. In both cases, measurements are performed using the logic analyzer, starting when the event is received by the collector and ending when the user thread has completed its processing.

Results are presented in Fig. 7. Boxplots summarize the distribution of the measured response times. The box spans the 25th to 75th percentiles, with the central line indicating

the median. Whiskers extend to the farthest observations lying within 1.5 times the interquartile range (IQR) from the box, while individual measurements are shown as scattered points. For the emergency braking application, the box is nearly collapsed to a single line, indicating that most measurements are concentrated around the same response time, with only a few isolated outliers. This highlights the very deterministic behavior of the system.

The emergency braking application exhibits a mean response time of 68 µs, well below its 1.5 ms deadline, while the sunroof application responds in 146 µs on average, far under its 150 ms deadline. Both applications consistently meet their deadlines despite the monitoring application continuously consuming CPU resources. Since both applications complete within the 1 ms time slice, real-time guarantees are enforced solely through the collector's higher priority and EDF scheduling, with the time slice acting as a safety net for longer-running tasks.

VI. Conclusion

This paper presents a framework for deploying isolated, untrusted applications on resource-constrained MCUs while preserving soft real-time guarantees. Built on top of Zephyr RTOS, the framework leverages hardware memory protection through the PMP mechanism, authenticated deployment via mTLS, and a condition-based publisher-subscriber model for sensor data sharing. Evaluation on an ESP32-C6 shows that the kernel-to-user context switch introduces a $2.5\times$ overhead compared to a kernel-to-kernel switch, and that native execution remains approximately $4\times$ faster than WebAssembly AOT.

Dynamic application lifecycle management, including hot replacement and removal, is left as future work, as is ELF file integrity verification prior to loading. Future work will also investigate the design of an orchestration algorithm capable of intelligently placing applications across a network of heterogeneous devices, optimizing resource utilization while respecting real-time constraints. Finally, while our evaluation targets a single-core RISC-V MCU, extending and evaluating the framework on multicore hardware and on other architectures such as ARM are left as future work.

References

[1] I. Mandel and U. Gupta, "Silicon Foraging: Harvesting Excess Compute for Sustainable Edge Computing," in *Proceedings of the 2025 ACM Designing Interactive Systems Conference*, ser. DIS '25. New York, NY, USA: Association for Computing Machinery, Jul. 2025, pp. 2574–2584.

[2] A. Ramesh, T. Huang, E. Ruppel, D. Dasari, B. Pourmohseni, F. Smirnov, M. Giani, P. Pazzaglia, C. Shelton, N. Pereira, A. Hamann, D. Ziegenbein, and A. Rowe, "Silverline: Lightweight Virtualization and Orchestration of Distributed Systems," in *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Irvine, CA, USA: IEEE, May 2025, pp. 375–388.

[3] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, and J.-P. Talpin, "Femto-Containers: Lightweight Virtualization and Fault Isolation For Small Software Functions on Low-Power IoT Microcontrollers," Oct. 2022.

[4] K. Skrivankova, M. Handley, and S. Hailes, "Why Are Smart Buildings Still Dumb: The Road Ahead," in *Proceedings of the ACM SIGCOMM 2025 Posters and Demos*. Coimbra Portugal: ACM, Sep. 2025, pp. 91–93.

[5] M. Has, T. Xiong, F. B. Abdesslem, and M. Kušek, "WebAssembly on Resource-Constrained IoT Devices: Performance, Efficiency, and Portability," Nov. 2025.

[6] L. Coppolino, S. D'Antonio, G. Mazzeo, R. Nardone, L. Romano, and M. Schmitt, "Wasmbox: A lightweight wasm-based runtime for trustworthy multi-tenant embedded systems," *IEEE Transactions on Emerging Topics in Computing*, vol. 13, no. 2, pp. 467–480, 2025.

[7] M. Seidler, A. Krause, and P. Ulbrich, "Wasm-io: Enabling low-level device interaction in webassembly for industry automation," *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 5s, pp. 1–26, 2025.

[8] Zephyr Project Contributors, "Zephyr Real-Time Operating System," https://www.zephyrproject.org, 2015, accessed: 2026-05-05.

[9] S. Akkermans, W. Daniels, G. Sankar R., B. Crispo, and D. Hughes, "CerberOS: A Resource-Secure OS for Sharing IoT Devices," in *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '17. Uppsala, Sweden: International Conference on Embedded Wireless Systems and Networks (EWSN), Feb. 2017, pp. 96–107.

[10] S. Akkermans, B. Crispo, W. Joosen, and D. Hughes, "Polyglot CerberOS: Resource Security, Interoperability and Multi-Tenancy for IoT Services on a Multilingual Platform," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ser. MobiQuitous '18. New York, NY, USA: Association for Computing Machinery, Nov. 2018, pp. 59–68.

[11] microEJ, "microej: Embedded java platform," https://www.microej.com.

[12] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kB Computer Safely and Efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 234–251.

[13] Arm Limited, "M-Profile architectures," https://www.arm.com/architecture/cpu/m-profile, accessed: 2026-05-06.

[14] A. Waterman, K. Asanović *et al.*, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture," RISC-V International, Tech. Rep., 2021, version 20211203. Available at https://github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12.

[15] "Zephyr user mode," https://docs.zephyrproject.org/latest/kernel/usermode/index.html, 2026, accessed: 2026-04-16.

[16] "Zephyr linkable loadable extensions (llext)," https://docs.zephyrproject.org/latest/services/llext/index.html, 2026, accessed: 2026-04-16.

[17] B. Campbell, J. Bradley, N. Sakimura, and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens," Internet Engineering Task Force, RFC 8705, Feb. 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8705

[18] "Esp32-c6," https://www.espressif.com/en/products/socs/esp32-c6 [Accessed: 2026-05-06].

[19] gusmanb and al., "LogicAnalyzer," Tech. Rep., 2022, available at https://github.com/gusmanb/logicanalyzer.

[20] bytecodealliance, "WebAssembly Micro Runtime," Tech. Rep., 2021, available at https://github.com/bytecodealliance/wasm-micro-runtime.

Interference-Free Channels: OS Support for Real-Time NoC Communication

Viktor Reusch
Barkhausen Institut
Dresden, Germany
viktor.reusch@barkhauseninstitut.org

Michael Roitzsch
Barkhausen Institut
Dresden, Germany
michael.roitzsch@barkhauseninstitut.org

The growing interest in Internet-of-Things (IoT) devices —many with stringent real-time requirements—has fueled demand for edge computing [1], [2], [3] to offload computation [4] and enable low-latency coordination [5] among distributed sensors and actuators. However, this convergence creates a unique challenge for edge clouds: They must host mutually distrusting tenants on shared compute resources [6], [7] while simultaneously satisfying real-time demands of varying criticality [5].

Tenant isolation lies at the heart of these mixed-criticality edge clouds, where predictable latencies and low jitter are mandatory. Proper isolation also strengthens the resilience of security-critical applications against timing side channels [8]. On heterogeneous manycores, the network on chip (NoC) is a critical shared resource: Without isolation, unrelated applications contend for routers and links, threatening real-time guarantees. While prior work has developed effective hardware primitives for NoC isolation [8], [9]—such as time-division multiplexing (TDM) and dynamic route control —these mechanisms remain low-level. They are not yet elevated to a secure operating system (OS) abstraction that can dynamically configure routes. With OS-managed route control, permitted applications can selectively secure routes as desired—e.g, by separating them from best-effort traffic using TDM, guaranteeing isolation among tenants. OS control also ensures fair access to the NoC—and especially the route isolation mechanism—as a limited, shared hardware resource.

In the accompanying presentation, we outline our research agenda to close this gap by making NoC route control a first-class, securely managed resource in the *M³* microkernel-based operating system [10]. The M³ hardware platform is a heterogeneous manycore with custom, per-tile *trusted communication units* (TCUs), which isolate tiles and mediate cross-tile communication. M³ keeps application tiles out of the trusted computing base by relying on only a small, capability-based microkernel as the trusted, central permission manager. We plan to extend the M³ hardware platform with a dynamically controllable NoC—supporting source- or table-based routing together with TDM—so that the OS can isolate critical inter-process communication channels from competing traffic.

We present early design considerations for a route-control service that computes optimal routes while also ensuring fair and isolated allocation of NoC resources among applications. We propose securing this abstraction by extending M³'s capability-based permission system to cover network routes, with the TCU hardware enforcing kernel-managed permissions at the tile boundary and preventing unauthorized router reconfigurations. This co-design of OS-level route management and hardware-based capability enforcement aims to transform the NoC from a globally shared bottleneck into a securely partitioned communication substrate. For mixed-criticality edge clouds, this directly translates into strengthened real-time guarantees for latency-sensitive IoT workloads and the elimination of one of the few remaining potential side channels in the tiled M³ architecture—ultimately enabling trustworthy coexistence of distrusting tenants on a single chip.

References

[1] M. Satyanarayanan, "The Emergence of Edge Computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017, doi: 10.1109/MC.2017.9.

[2] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, "Cloudlets: at the leading edge of mobile-cloud convergence," in *6th International Conference on Mobile Computing, Applications and Services*, 2014, pp. 1–9. doi: 10.4108/icst.mobicase.2014.257757.

[3] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019, doi: 10.1109/JPROC.2019.2918951.

[4] B. Li, W. Dong, and Y. Gao, "WiProg: A WebAssembly-based Approach to Integrated IoT Programming," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10. doi: 10.1109/INFOCOM42981.2021.9488424.

[5] W. Shao, B. Ye, H. Wang, G. Parmer, and Y. Ren, "Edge-RT: OS Support for Controlled Latency in the Multi-Tenant, Real-Time Edge," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 1–13. doi: 10.1109/RTSS55097.2022.00011.

[6] Y. Ren *et al.*, "Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, July 2020, pp. 927–942. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/ren

[7] X. Han, Y. Gao, G. Parmer, and T. Wood, "Byways: High-Performance, Isolated Network Functions for Multi-Tenant Cloud Servers," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024, pp. 811–829. doi: 10.1145/3698038.3698547.

[8] U. Ali, S. A. R. Sahni, and O. Khan, "Characterization of Timing-based Software Side-channel Attacks and Mitigations on Network-on-Chip Hardware," *J. Emerg. Technol. Comput. Syst.*, vol. 19, no. 3, June 2023, doi: 10.1145/3585519.

[9] H. Omar and O. Khan, "IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 111–122. doi: 10.1109/HPCA47549.2020.00019.

[10] 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*, 2016, pp. 189–203. doi: 10.1145/2872362.2872371.

Partition-Sensitive Interference on NVIDIA GPUs: 500,000 Ways to Exceed WCETs

Sarah Crowder*, Syed W. Ali*, and James H. Anderson*
*Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC
crowdies@unc.edu, swali@cs.unc.edu, anderson@cs.unc.edu

***Abstract*—As safety-critical systems increasingly utilize AI workloads, providing robust real-time guarantees becomes a pivotal issue for certifiably safe operation. Spatial GPU partitioning techniques have been proposed to satisfy the computational requirements of such systems. With over 500,000 ways to select partitions on modern GPUs, it becomes infeasible to test every configuration while deriving worst-case execution times (WCETs) in measurement-based timing analysis; thus, understanding how and to what extent partitions can affect execution times is critical. This work explores how NVIDIA GPU partition selection affects WCETs, demonstrating an up to 70.4% increase solely due to partition selection.**

***Index Terms*—real-time systems, GPU partitioning, measurement-based timing analysis**

I. Introduction

The field of artificial intelligence (AI) has become an increasingly important area of interest for safety-critical systems. These systems can be massively sped up with the use of graphical processing units (GPUs), highly parallel processors with massive raw compute potential. One breakthrough to address the substantial computational requirements of AI workloads is "spatial partitioning" of the GPU [1]. This enables increased throughput [2] by allowing practitioners to specify *which* compute units of the GPU to run workloads on.

Lacking public information on the internal scheduling and implementation details of NVIDIA GPUs, many researchers and practitioners have turned to measurement-based timing analysis to derive worst-case execution times (WCETs) for these systems. These techniques typically involve running a given program many times to observe its behavior [3]. Practitioners simulate the worst-case deployment conditions when deriving these bounds, often including co-running interfering kernels.

However, GPU spatial partitioning introduces an underexplored variable that could affect WCETs: *which* compute units are being used. Using an NVIDIA 3060 Ti, for example, gives practitioners 2^{19} (524,288) possible configurations to experiment with. Newer GPUs can give upwards of 2^{40} configurations, or over one trillion. Running on each configuration directly is infeasible, especially as modern GPUs become more complex.

In this work, we observe that spatial partitions of the same size can exhibit WCET variance of up to 70.4%. We argue that this variable must be further explored and safely accounted for to ensure reliable WCET measurement.

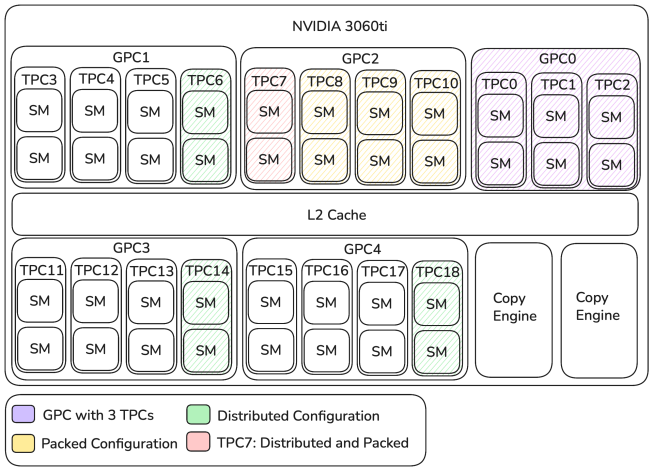


Fig. 1. A simplified model of NVIDIA 3060 Ti architecture.

This paper makes the following contributions. First, we profile arbitrary partition configurations on an NVIDIA 3060 Ti and demonstrate significant but manageable inherent differences in execution times. Second, we demonstrate that with co-running workloads, partition selection can introduce much larger variance.

II. Background

NVIDIA GPU Architecture. Although an NVIDIA GPU has many components, the compute engine does the main processing. It has a nested structure: Each GPU contains Graphical Processing Clusters (GPCs), GPCs contain Texture Processing Clusters (TPCs), and TPCs contain Streaming Multiprocessors (SMs). Each SM has its own L1 cache, whereas the L2 cache and larger memory is shared between all SMs on the system. A visual is presented in Figure 1.

GPU Programming Fundamentals. To program these architectures, NVIDIA developed CUDA: an API to run custom general processing workloads on the GPU. A CUDA program is split into a CPU-side section and a GPU-side section. The GPU-side section is called a *kernel.* Normally, one kernel can execute at a time on a GPU. However, one can run multiple kernels at once using a CUDA *stream.* A *stream* is a FIFO queue of instructions to run one after the other. This work uses separate CUDA streams to run multiple kernels at the same time on the GPU.

Spatial Partitioning Methods. There are multiple ways to spatially partition a GPU, but each comes with tradeoffs. NVIDIA's own Multi-Instance GPU server, for example, is one popular solution. However, it comes with the drawback of having high overhead, which is not ideal for a safety-critical system. For this work, we utilize `libsmctrl`: a lightweight user-space library that allows users to specify which TPCs to run kernels on an NVIDIA GPU [4], and is a popular method in the real-time community. While our findings may extend to other partitioning methods, we consider them to be out of scope.

The `libsmctrl` library works by allowing users to specify a *TPC mask*: a 128 bit integer. The least significant n bits of the mask, where n is the number of TPCs in the system, are used to specify which TPCs are disabled. This API allows for 2^n possible masks.

Although libsmctrl provides spatial compute partitioning, it does not guarantee mitigation of interference from other shared resources such as the L2 cache. For this work, we define memory interference as competing with a co-running kernel that contends for memory resources.

III. Related work

There has been research on quantifying and understanding inter-SM interference [5], [6]. However, most of these works do not consider partition *selection* as a variable. To our knowledge, there are two studies that do consider it, but they each have their own limitations.

Otterness et al. [7] explores scheduling details and the effects of mask selection in AMD GPUs. Instead of testing across the decision space of all possible masks, they limit their investigation to a few mask selection techniques. The authors introduce the terminology of "SE-packed" and "SE-distributed" partitions. Structurally, an "SE" has the same nested structure as a GPC on an NVIDIA GPU. An "SE-packed" partition assigns TPCs to a workload in the same GPC first, whereas an "SE-distributed" partition assigns TPCs on different partitions if possible. Figure 1 shows an example of these masking strategies on an NVIDIA GPU. The authors found that this distinction does modestly impact execution times. However, many of their findings and reasoning are AMD-specific and not directly applicable to NVIDIA GPUs [4].

Bakita et al. not only developed the `libsmctrl` library, they also investigated whether "SE-packed" and "SE-distributed" [4], [7] made any difference in the execution times. The authors found that there was a negligible difference between the masking strategies on NVIDIA GPUs. However, this experimental setup had no interference, limiting the scope of their experiments. This discrepancy is noted as a limitation.

None of the prior works characterize TPC mask selection as a variable for worst case execution time with memory interference on NVIDIA hardware, a critical oversight for safely deploying a real-time system with spatially partitioned GPUs as a component.

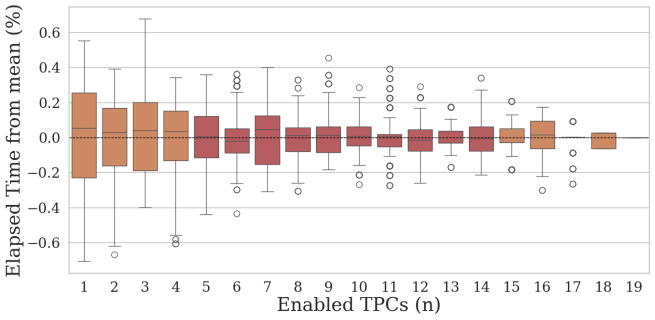


Fig. 2. Variance between 90th percentile runs of different masks. Exhaustive searches are put in orange and non-exhaustive searches (sample size = 500) are put in red.

IV. Experiments

Experimental Setup. We perform our experiments on a system equipped with an NVIDIA 3060 Ti. The NVIDIA 3060 Ti has a total of 19 TPCs and five GPCs. Each GPC has four TPCs, except for GPC0, which has three. The GPU topology was found using `libsmctrl` tooling and is presented in Figure 1.

We use a tiled matrix multiplication kernel with 32×32 thread blocks and shared memory caching. In our experiments, the kernel multiplies a 640×640 matrix by a 640×1280 matrix. The kernel is launched with 800 thread blocks across a 40×20 grid. We follow the example of [7] using matrix multiplication as a benchmark kernel, as it is a common operation in AI workloads and is well-understood. Execution times are measured using CUDA events for GPU-side execution only after a warm-up execution.

TPC Selection Under No Interference. We perform the following experiment to separate any inherent differences from the effects with interference. To establish a baseline, we characterize TPC mask selection on the 3060 Ti as follows. We define $k = TPC_{enabled}$. The number of possible masks that assign k total TPCs is given by $\binom{19}{k}$. For this experiment, we exhaustively test every $\{k \mid \binom{19}{k} \leq 10000\}$, corresponding to $k \leq 4$ and $k \geq 15$. For remaining values of k, we use a random sampling of 500 possible TPC masks, manually ensuring inclusion of "SE-distributed" and "SE-packed" configurations. For each mask, we repeat 1,000 times and present 90th percentile results in Figure 2.

Observation 1. TPC mask selection without interference introduces up to 1.27% variance in execution times across all tested masks.

A Kruskal-Wallis test confirms that differences between TPC masks are significant $p < 10^{-20}$ between max and min for all TPC configurations. However, while the difference is present and statistically significant, it is small enough to be safely accounted for in most safety critical applications that already add pessimism to their WCETs. However, we still recommend profiling any differences in partition configurations before running a critical system.

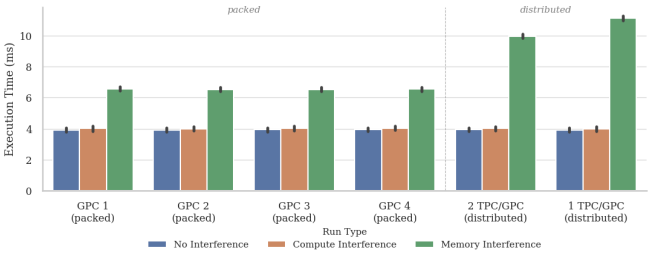


Fig. 3. Matrix multiply execution time (ms) by TPC mask and contention mode.

TPC Selection Under Memory Interference. In a real spatially partitioned system, there are likely *interfering* workloads running concurrently. In accordance with [5], we create an "enemy" interfering workload. This is done to simulate a kernel that is performing memory interference more than any realistic co-running kernel. Our interference kernel allocates a 512 MB buffer and performs pseudo-random read-modify-write operations using a Knuth multiplicative hash. The kernel is launched with 64 thread blocks of 256 threads each. We also perform a test where we instead run a second matrix multiply with larger dimensions in place of the memory kernel.

Our CUDA programs are split into three sections: a setup section, an execution section, and a tear-down section. We utilize a C++ program to set up two CUDA streams. We release the execution segments at the same time using a `pthread_barrier`.

For this experiment, we run matrix multiply with four enabled TPCs. We test six different configurations of TPCs each time: one on each full GPC (GPC1 - GPC4), one on a "distributed" configuration of TPCs, and one with two TPCs per GPC enabled. We choose four enabled TPCs to most clearly compare an "SE-packed" vs. "SE-distributed" strategy, as it allows us to fully "pack" TPCs in a GPC as well as nearly maximally "distribute." We perform 5,000 iterations for each mask configuration in a rotating order.

Observation 2. We observe memory interference has a great effect on matrix multiply execution time across all tested partition configurations, increasing it by up to 183%.

This result is alarming, but consistent with prior work [5]. While the impact that memory interference has on execution times is very large (and we do not argue our enemy is optimal), it can be accounted for. It is known that while deriving WCETs, practitioners must run their work alongside the most interfering task [5].

Observation 3. In the presence of memory interference, we observe a 70.4% difference in execution times from the best partition selection to the worst.

Unlike prior work [4], this observation suggests that partition selection alone can be a significant variable for execution times on NVIDIA GPUs. Hence, partition selection represents a significant area of inquiry that warrants further investigation.

We continue from this experiment by profiling a representative repetition of the experiment using Nsight Compute [8]. Unfortunately, we were unable to study the mechanisms of interference using this tool. Since Nsight Compute serializes execution of kernels when profiling, we were unable to analyze statistics such as L2 hit rate differences. This concession is mirrored in work by Elvinger et al. [6]. We invite discussion on the possible mechanisms of differing interference.

We repeat the experiment on an NVIDIA 4080 Super, which has 40 TPCs distributed across 7 GPCs. When replicating the experiment, we perform matrix multiplication across the six (rather than four) TPCs, either in a packed or spread configuration. We observe up to a 521% (compared to 183%) total memory interference effect and an 89% (compared to 70.4%) partition selection effect. We hypothesize that the difference is due to more efficient interference production.

V. Conclusion

Implications for Real-Time systems. Our results, while preliminary, could be a critical warning for GPU-accelerated spatially partitioned safety-critical applications. If partitions are not taken into account, it could be possible to underestimate WCETs, posing critical safety risks. We suggest that when deriving measurement-based timing analysis for spatially partitioned systems, WCETs for both "SE-packed" and "SE-distributed" configurations should be measured alongside co-running workloads. While more research is needed, our results suggest this extra step could be critical for ensuring safe systems. However, it might be beneficial to avoid memory-intensive GPU workloads altogether while utilizing spatial partitioning until a more complete picture of interference mechanisms is achieved for maximum safety.

Future Work. We propose further research and experiments for TPC mask selection in both the synthetic worst case and realistic workloads in order to further quantify and understand the extent that partition selection can have on interference as well as the methods. In order to facilitate this and any research into methods of interference, tooling to deeply profile workloads under interference, similar to or extending Nsight Compute, would be greatly beneficial. In the case where partition-aware interference variability is shown to be significant in realistic workloads, there would be opportunity to develop new software tooling to reliably provide safe upper bounds or runtime allocation policies to avoid detrimental partitions.

Final remarks. In the advent of spatially partitioned GPUs, the effect of selecting partitions is underexplored. In this work, we have shown that the configuration of TPCs selected to run can be critical: increasing execution times by up to 70.4% with memory interference. Further investigation is needed to establish safe spatial partitioning for memory-intensive workloads, and we welcome contributions from the research community towards this end.

References

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[2] S. W. Ali, J. Goh, J. Bakita, S. Chakraborty, and J. H. Anderson, "Concurrent fft execution on gpus in real-time," in *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2025, pp. 154–161.

[3] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, "Wcet measurement-based and extreme value theory characterisation of cuda kernels," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 279–288. [Online]. Available: https://doi.org/10.1145/2659787.2659827

[4] J. Bakita and J. H. Anderson, "Hardware compute partitioning on NVIDIA GPUs," in *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 54–66.

[5] T. Yandrofski, J. Chen, N. Otterness, J. H. Anderson, and F. D. Smith, "Making powerful enemies on NVIDIA GPUs," in *Proceedings of the 43rd IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 383–395.

[6] P. Elvinger, F. Strati, N. Enright Jerger, and A. Klimovic, "Understanding GPU resource interference one level deeper," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2025.

[7] N. Otterness and J. H. Anderson, "Exploring AMD GPU scheduling details by experimenting with "worst practices"," in *Proceedings of the 29th International Conference on Real-Time Networks and Systems (RTNS)*, 2021, pp. 24–34.

[8] NVIDIA Corporation, *Nsight Compute Profiling Guide*, 2025, accessed: 2026-05-12. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

Scheduling Behaviors of the CUDA Graph API

Theodore L. Demetriades, Rebecca Moran, Tanya Amert*
Department of Computer Science
Carleton College
Northfield, Minnesota, USA

***Abstract*—Despite their near-ubiquity in computer-vision and deep-learning applications, scheduling behaviors of NVIDIA GPUs are still not clearly documented. This is particularly true when considering the interaction between various configurable properties of GPU operations, such as the use of high-priority streams used to organize and synchronize GPU operations.**

This is exacerbated when considering the CUDA Graph API, which enables GPU operations to be organized into dependency graphs that can be launched on the GPU as a unit, and how its use interacts with the default stream, prioritized streams, and with launches of non-graph kernels. This paper details explorations of GPU scheduling when using NVIDIA's CUDA Graph API, including the presence of implicit synchronization (including blocking of the CPU itself) when mixing graph and non-graph kernels.

***Index Terms*—graphics processing units, hardware accelerators, scheduling**

I. Introduction

From early uses in accelerating computer-vision applications to the ever-changing landscape of large language models and other deep-learning-based applications, Graphics Processing Units (GPUs) have remained a prominent hardware accelerator, with NVIDIA persisting as a leader in GPU design. Furthermore, the mechanisms by which we can submit work to GPUs continue to evolve, including the ability to organize GPU operations into graphs with complex inter-node dependencies using NVIDIA's CUDA Graph API [9].

Scheduling policies of NVIDIA GPUs were previously explored by Amert et al. [2]; however, they focused on the basic scheduling rules, considering only a few complexities in isolation. Yang et al. [13] extended this work to describe the hidden pitfalls that can arise with NVIDIA GPUs, in particular noting implicit synchronization that can occur (causing unexpected blocking *on other CPU cores*) when interleaving certain GPU operations. Others have sought to modify the GPU scheduling behavior, either by modifying the GPU driver [4], [5] or by configuring user-specified affinity masks to limit the mapping of GPU operations to streaming multiprocessors [3].

However, prior work has neglected the scheduling behavior using the CUDA Graph API, despite the presence of CUDA Graphs in GPU-using frameworks like TensorFlow and PyTorch [6]. As a result, it is unclear whether any of the pitfalls explored by Yang et al. [13] may arise in the presence of graphs, as well as how graph computations are scheduled relative to existing kernels.

*Corresponding author: tamert@carleton.edu

Contributions. In this paper, we explore the scheduling of GPU operations using the NVIDIA CUDA Graph API. We reveal surprising behaviors relating to the order that operations are submitted to the GPU, resulting in implicit synchronization on both the GPU and the host CPU. We also provide insight into the impact of mixing the default and user-defined streams, prioritized streams, and graph and non-graph operations.

Organization. We first provide background on NVIDIA GPUs and the CUDA API and CUDA Graphs (Sec. II). Then, we describe some simple experiments working with the CUDA Graph API in the presence of user-defined streams and prioritized streams (Sec. III). Next, we explore the interactions of these scenarios with each other and with non-graph kernels, in which we reveal a new unexpected source of implicit synchronization (Sec. IV) before concluding (Sec. V).

II. Background

We now provide background on NVIDIA GPUs and the use of the CUDA API to interact with them, as well as prior efforts to elucidate details of the internal scheduling policies employed by the CUDA runtime.

NVIDIA GPUs and CUDA programming. From a hardware perspective, an NVIDIA GPU features a few to tens or even hundreds of *streaming multiprocessors* (SMs) that comprise its Execution Engine and execute computations structured into functions called *kernels*. One or more Copy Engines serve to copy data between the CPU ("host") and GPU ("device").

Communication with an NVIDIA GPU is performed using the CUDA API [8], which extends C/C++ to provide additional functionality for submitting kernels and memory operations to the GPU. A typical GPU-using program has the following pattern: (1) memory is allocated on the GPU, (2) input data are copied from the CPU to the GPU, (3) one or more kernels are executed on the data, (4) results are copied back to the CPU, and (5) the allocated memory on the GPU is freed.

GPU operations may be organized within a *stream* to ensure FIFO ordering, or within multiple *user-defined streams* if concurrent execution is desired. If no user-defined stream is specified, the *default* (or *NULL*) stream is used; any two operations must execute serially if an operation is submitted to the default stream between their submissions.[1]

[1]Starting with CUDA 7, the compiler option `--default-stream per-thread` changes the behavior of the default stream to only explicitly synchronize operations belonging to an individual CPU thread; we do not use this option for our experiments.

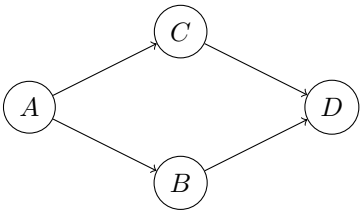


Fig. 1: A four-node graph in which nodes represent GPU operations (e.g., kernels or copies) and edges represent dependencies.

Examining GPU scheduling behavior. Despite extensive documentation, there are still many details of the scheduling policies utilized by NVIDIA GPUs that are left unclear. Amert et al. [2] detailed several rules governing the behavior of individual kernel and copy operations submitted to the GPU, specifically focused on the details pertaining to operations submitted to user-defined streams. Their work was based on a series of experiments performed using the CUDA Scheduling Examiner tool [10]. (See Fig. 2a in Sec. III for an example plot resulting from experiments using this tool.)

Potential pitfalls. Yang et al. [13] built on this work to describe several pitfalls that GPU programmers may encounter. Note that from reading the NVIDIA documentation, one may assume that kernel launches are asynchronous with respect to the CPU (i.e., that the CPU continues to execute instructions immediately). However, Yang et al. found that implicit synchronization on NVIDIA GPUs can result in GPU operations unexpectedly blocking the *CPU thread*.

CUDA Graphs. CUDA Graphs, to our knowledge yet unexplored in terms of scheduling behavior, were added to the CUDA API in version 10.0, released in September 2018. In a CUDA Graph, nodes represent operations and edges represent dependencies between the operations. Nodes can be one of many types, including kernels, memory operations, host functions executed on the CPU, or even child graphs. A simple example graph that we utilize for our experiments (as explained in Sec. III and IV) is depicted in Fig. 1.

The purpose of CUDA Graphs is to reduce CPU-side overhead in repeated launches of the same workflow. To achieve this, a CUDA Graph is defined and instantiated once, and launched as many times as needed. One way[2] to define a CUDA Graph is with what is called a "stream capture", which monitors a stream without actually launching any operations in order to infer the graph's structure. Alternatively, one can explicitly invoke API calls to build a graph by directly specifying nodes and edges.

III. CUDA Graph Scheduling

We now describe our experiments using the CUDA Graph API. Our benchmark code and scripts are available online.[3]

[2]More information regarding CUDA Graphs is available in the CUDA documentation [7].

[3]The code for our experiments is available online at https://github.com/tanya-amert/cuda_scheduling_examiner_mirror.

Experiment setup. For our explorations of CUDA Graph scheduling, we implemented a simple benchmark within the CUDA Scheduling Examiner framework.[4] Our benchmark corresponds to the graph in Fig. 1 with each node representing a CUDA kernel; this simple graph enables observation of both concurrent execution and inter-node dependencies (e.g., kernels B and C may execute together, and kernel D cannot begin until both kernels B and C complete). We create the graph using explicit API calls rather than stream captures, as discussed in Sec. II. To focus on a predictable order of kernel execution, we configure each kernel to execute for a fixed duration (e.g., 0.5 or 1.0 seconds), as in the original GPU scheduling exploration by Otterness et al. [11].

We performed each experiment on two machines: (1) a laptop running Windows 11 with Ubuntu 24.04 via Windows Subsystem for Linux, equipped with a discrete NVIDIA GeForce RTX 4050 (with CUDA 13.1) and (2) an NVIDIA Jetson Orin NX 8GB (with CUDA 12.2). For the sake of visibility (due to fewer SMs), we present the results from the Jetson Orin NX in this paper, but observed the same trends on both machines.

Motivating questions. Our investigation is guided by the following questions:

- **Q1** How does CUDA assign graph nodes to streams?
- **Q2** What restrictions apply when a graph is put in the default stream?
- **Q3** How does CUDA prioritize graphs launched into streams with different priorities?
- **Q4** How does CUDA schedule graphs and non-graph kernels together?

In the remainder of this section, we present the results of experiments addressing the first three of these questions in isolation. In Sec. IV we address Q4 and consider the other questions in combination.

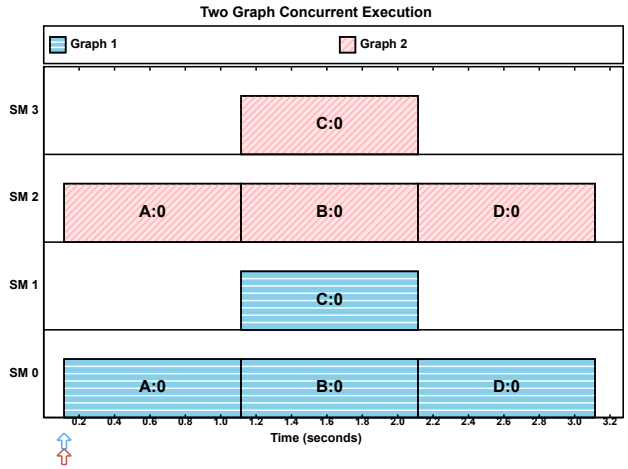
A. Assigning Graph Nodes to Streams

We begin with an exploration of Q1. In our example graph in Fig. 1, kernel nodes B and C could, in theory, execute concurrently given that they have no dependence on one another. However, when we explicitly create and launch a graph, we do so in a single CUDA stream. Because streams correspond to FIFO queues of operations, if B and C execute concurrently, then at least one of them must do so in a different stream than the one into which we launch the graph.

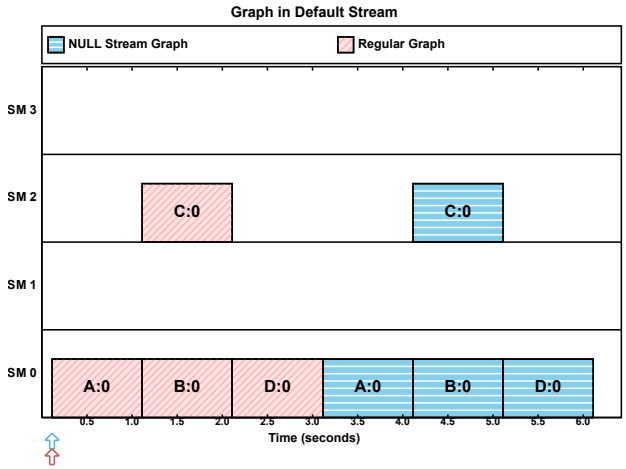
We observe this behavior as part of a larger experiment in which we launch two copies of our graph in two separate user-defined streams. The results are depicted in Fig. 2a.

In this figure, each rectangle corresponds to a block (the schedulable unit of CUDA kernels, although in these experiments we configure each node to be a kernel with one just block) with its height indicating the number of GPU threads it uses to execute. The horizontal position of a rectangle indicates

[4]Available online at https://github.com/yalue/cuda_scheduling_examiner_mirror.



(a) Two graphs launched in separate user-defined streams may execute concurrently.



(b) A graph launched in the default stream (blue) cannot execute concurrently with another graph in a user-defined stream.

Fig. 2: Changes in CUDA Graph scheduling behavior due to stream choice.

the interval during which that block executes, and the vertical position indicates the SM on which it executes. Each block is labeled with the kernel/node name and the block number (one kernel may comprise several blocks). Arrows below the plot indicate the time each kernel is launched onto the GPU; for CUDA graphs, the arrow indicates the time of the call to `cudaGraphLaunch` for the entire graph.

For this experiment, we see that all blocks of nodes B and C between the two graphs execute concurrently on the GPU.

Obs. 1: CUDA can create separate streams to enable concurrent execution of graph nodes.

It is important to note that the graph colors used in plots by the CUDA Scheduling Examiner correspond to the stream in which a graph is launched, not the actual stream in which a given node (or kernel) executes.[5] Therefore, there could be more streams used than those indicated by the two colors in Fig. 2a. We separately ran this experiment using NVIDIA's Nsight Systems performance analyzer, which can report stream IDs, and verified that node C executes in a separate stream.

B. CUDA Graphs and the NULL Stream

As mentioned in Sec. II, non-graph kernels can execute concurrently if they are launched into different user-defined streams. We see the same behavior for CUDA Graphs, as illustrated in Fig. 2a. To explore Q2, we repeat the experiment depicted in Fig. 2a but instead launch one of the two graphs in the NULL stream; the result is shown in Fig. 2b. In this case, the two graphs' executions are serialized.

Obs. 2: A CUDA Graph launched in the default stream executes serially with respect to graphs launched in user-defined streams.

[5]This is because the programmer can only specify the stream in which a graph is launched, and cannot programmatically query the stream in which a node (or any kernel) executes. On the contrary, the SM on which a block executes is available by reading the `smid` register.

This same behavior was discussed by Yang et al. [13] as a source of potential pitfalls due to the implicit synchronization of the GPU when a kernel (or in this case, a graph) is launched in the NULL stream; we therefore propose the following recommendation when using CUDA Graphs:

Recommendation #1: Avoid launching CUDA Graphs in the default stream.

There is one oddity with Fig. 2b that should be addressed: the kernels corresponding to nodes B and C of the graph in the default stream execute concurrently.

Obs. 3: Nodes of a CUDA Graph can execute concurrently even when the graph is launched in the default stream.

We again used Nsight Systems to verify that node C executes in a separate stream (further evidence of Obs. 1). However, it is somewhat unexpected that node C can execute concurrently with a node (B) in the NULL stream.

C. Starvation of Low-Priority Graphs

Amert et al. [2] showed that kernels launched in low-priority CUDA streams can experience starvation when the GPU is filled with work from high-priority streams. To investigate Q3, we consider an analogous situation in which a graph launched in a low-priority stream is blocked by graphs launched in higher-priority streams, as shown in Fig. 3.

Obs. 4: A graph launched in a low-priority stream may experience starvation due to graphs in higher-priority streams.

As shown in Fig. 3, node A of the low-priority graph (yellow) is launched and begins executing first, but node B is delayed as higher-priority graphs utilize the majority of the GPU resources. Furthermore, node D of the low-priority graph is blocked for even longer, demonstrating potentially unbounded blocking due to launches of higher-priority graphs.

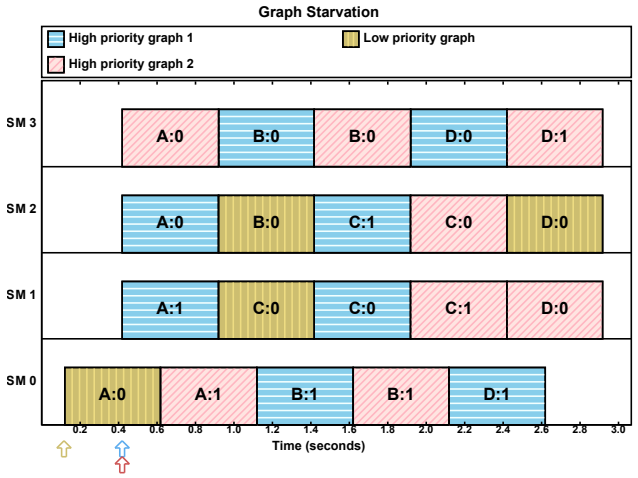


Fig. 3: Starvation of a graph in a low-priority stream.

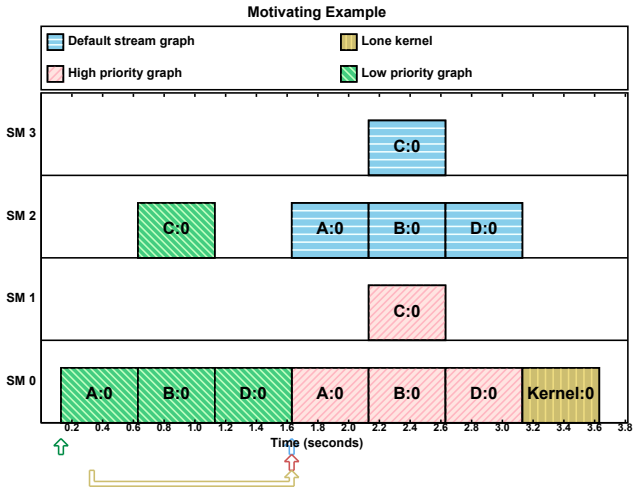


Fig. 4: Execution of three graphs and a lone kernel.

IV. Complex Scheduling Scenarios

In Sec. III we explored a few basic scenarios. The experiments presented in this section allow us to tease apart complex interactions that occur in our motivating experiment, the results of which are shown in Fig. 4. In this experiment, we launch three graphs into separate streams (one in the default stream and one in each of a high- and low-priority streams) as well as a single non-graph kernel (which we call a *lone kernel*).

Looking at Fig. 4, we observe two unexpected behaviors:

1) The graph launched in the default stream (blue) executes concurrently with the graph in the high-priority stream (red) for the entire duration of both graphs.
2) The lone kernel launch experiences, or perhaps causes, implicit synchronization: the launch of the lone kernel at time 0.33 not only blocks its CPU thread, but also delays the launch (on the CPU) of the two remaining graphs, from time 0.43 to time 1.63.

To attempt to make sense of the interactions occurring in Fig. 4, we first consider interactions corresponding to Q2 and Q3 together and then address Q4.

A. Mixing the Default and Prioritized Streams

To address the first unexpected behavior listed above, we performed an experiment comprising two CUDA Graphs, one launched into a high-priority user-defined stream, and another into the default stream. The results of this experiment are shown in Fig. 5; this leads us to a modification of Obs. 2:

Obs. 2 (modified): A CUDA Graph launched in the default stream executes serially with respect to graphs launched in *normal-priority* user-defined streams.

Obs. 5: A CUDA Graph launched in the default stream may execute concurrently with a graph launched in a *high-priority* user-defined stream.

This experiment is extremely similar to the one presented in Fig. 2b, except that the user-defined stream (corresponding to the blocks shown in red) is of normal priority in Fig. 2b and high priority in Fig. 5. The use of high-priority streams seems to enable the default-stream graph to begin executing concurrently with the prior one, contradictory to the typical semantics of the default stream.

B. Mixing CUDA Graphs and Lone Kernels

To address Q4, we performed the same experiment as in Fig. 5, except we replaced the default-stream *graph* with a default-stream *lone kernel*. The result is shown in Fig. 6a.

To our surprise, we found that CPU blocking occurs in this case; the elongated arrow at the bottom of the plot indicates a delay between when the launch call began and when it returned, i.e., that it was *synchronous* with respect to the CPU. Note that although lone kernels by themselves execute serially when the default stream is used, the launches do not block the CPU. Furthermore, we see this same behavior in Fig. 4 with a graph in a low-priority stream instead of the high-priority one in Fig. 6a.

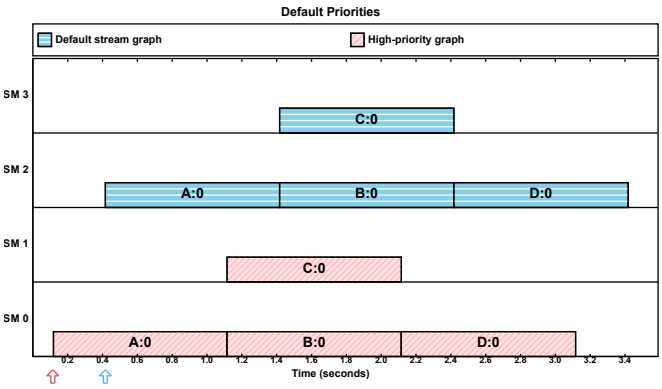
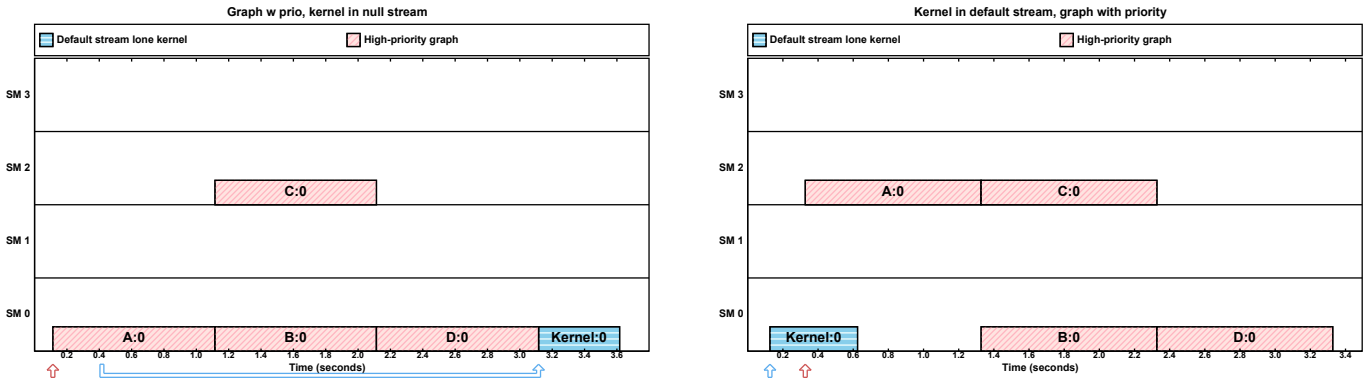


Fig. 5: A graph launched in the default stream (blue) *can* execute concurrently with another graph in a *high-priority* user-defined stream.



(a) Implicit synchronization of a lone kernel and graph.

(b) Concurrent execution of a lone kernel and graph.

Fig. 6: Interactions when mixing graph and lone-kernel executions.

However, we see very different behavior if the default-stream lone kernel executes first and the graph is launched later, as shown in Fig. 6b. In this case, both the kernel and the first node of the graph execute concurrently.

Obs. 6: Kernels launches while a graph is executing are blocked on the CPU until prior work completes, whereas graphs launched after a kernel has begun execution are not.

It is worth noting that we see this same behavior when the graph and the lone kernels are launched in regular (non-prioritized) user-defined streams as well. Given this behavior, we propose another recommendation for using CUDA Graphs:

Recommendation #2: Avoid using both CUDA Graphs and non-graph kernels in the same application.

V. Conclusion

In this paper, we present the results of our explorations of CUDA scheduling policies when using the CUDA Graph API. Our experiments utilize a simple graph topology to elucidate several unexpected behaviors, such as the concurrent execution of graphs launched in user-defined streams and the default stream as well as implicit synchronization and CPU blocking resulting from kernel launches while a graph is executing.

In the future, we plan to explore whether these behaviors differ if the CUDA Graph is set up using the stream capture approach, and to explore more complex graph topologies. We additionally plan to utilize the `libsmctrl` library [3] with our experiments to study whether partitioning the SMs between CUDA Graphs and lone kernels impacts the observations we have presented here. Finally, we plan to survey frameworks like PyTorch [12] and TensorFlow [1] to both determine the types of graph topologies they use and identify other functionality provided by the CUDA API that has yet to be explored from a predictable-scheduling point of view.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), http://tensorflow.org/, software available from tensorflow.org

[2] Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D.: GPU scheduling on the NVIDIA TX2: Hidden details revealed. In: Proceedings of the 38th IEEE Real-Time Systems Symposium. pp. 104–115 (2017)

[3] Bakita, J., Anderson, J.H.: Hardware compute partitioning on NVIDIA GPUs. In: Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 54–66 (2023)

[4] Capodieci, N., Cavicchioli, R., Bertogna, M., Paramakuru, A.: Deadline-based scheduling for GPU with preemption support. In: Proceedings of the 39th IEEE Real-Time Systems Symposium. pp. 119–130 (2018)

[5] Kato, S., Lakshmanan, K., Rajkumar, R., Ishikawa, Y.: TimeGraph: GPU scheduling for real-time multi-tasking environments. In: Proceedings of the USENIX Annual Technical Conference. pp. 17–30 (2011)

[6] Nguyen, V., Carilli, M., Burc Eryilmaz, S., Singh, V., Lin, M., Gimelshein, N., Desmaison, A., Yang, E.: Accelerating PyTorch with CUDA Graphs (2021), https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs

[7] NVIDIA: CUDA C++ programming guide (2025), https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[8] NVIDIA: CUDA runtime API :: CUDA toolkit documentation (2025), https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[9] NVIDIA: CUDA runtime API :: CUDA toolkit documentation, graph management (2025), https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

[10] Otterness, N.: Developing Real-Time GPU-Sharing Platforms for Artificial-Intelligence Applications. Ph.D. thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA (2022)

[11] Otterness, N., Yang, M., Amert, T., Anderson, J.H., Smith, F.D.: Inferring the scheduling policies of an embedded CUDA GPU. In: Proceedings of the 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications. pp. 47–52 (2017)

[12] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[13] Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J.H., Smith, F.D.: Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In: Proceedings of the 30th Euromicro Conference on Real-Time Systems. pp. 20:1–20:21 (2018)

Architecting Safety with Microkernels : An experience report to ASIL compliance on the example of L4Re

Kai Lampka (kai.lampka@nxp.com), NXP Semiconductors, Germany

Developing software in compliance with ISO 26262 entails strict adherence to rigorous engineering best practices across the entire V-model lifecycle. The standard mandates specific methods tailored to the software's assigned safety level. For instance, ISO 26262 highly recommends documenting detailed designs using semi-formal methods and standardized technical languages at both the system and software-unit levels. During the implementation phase, the standard emphasizes exhaustive verification techniques, including comprehensive statement and branch coverage testing, coupled with rigorous static code analysis. It is easy to see that developing safety-related software and achieving formal safety qualification can, therefore, quickly become insurmountably expensive.

This cost trajectory conflicts with two industry trends: the exponential growth of automotive software complexity and tightening regulatory mandates for functional safety and cybersecurity. To maintain economic feasibility, engineering teams must reduce the software codebase that undergoes safety qualification to a bare minimum.

This reduction is systematically driven by decomposing the software system according to its assigned safety integrity levels. Software decomposition aims to partition a monolithic software stack into independent, modular components. Under this paradigm, each individual component must comply only with the highest Automotive Safety Integrity Level (ASIL) found among the specific safety features it directly or indirectly implements. For this architectural approach to comply with the core principles of ISO 26262, the system must guarantee that faults cannot transparently migrate from lower-priority components to higher-priority ones. This fundamental property is known as Freedom-From-Interference (FFI). This safety requirement aligns perfectly with the compositional structure of modern microkernels. Microkernels inherently minimize risk by placing only essential core functionality—such as thread scheduling, memory management, and Inter-Process Communication (IPC)—inside the privileged kernel space. All other traditional operating system services, including device drivers, file systems, and networking stacks, are isolated within separate user-space components. Because this architecture naturally enforces strict spatial and temporal isolation, it restricts the functional blast radius of each component. Consequently, engineers can clearly define and limit the safety scope of both the microkernel and its dependent components. Tailoring and optimizing this Trusted Computing Base (TCB) is of paramount importance to ensure the overall certification project remains financially viable.

The presentation highlights an approach to certifying an open-source, microkernel-based operating system. Using the L4Re system as a concrete example, the talk demonstrates how to achieve ISO 26262 certification under Automotive Safety Integrity Level B (ASIL-B). The primary challenge addressed is how to certify a complex, comprehensive software stack that incorporates substantial pre-existing open-source components—all without resorting to a costly, impractical full reimplementation of every single line of legacy code. By leveraging the decomposition strategy and the hierarchical, compositional architecture of the L4Re system as a guiding design principle, the engineering team successfully minimized certification efforts, keeping the entire project highly cost-effective and scalable.

Resurrecting seL4's WCET Analysis

Simon Sillitoe[1], Abdul Basit[2], Massimiliano Giacometti[2], Gernot Heiser[1]
[1]UNSW Sydney, Australia
[2]PlanV Tech, Munich, Germany
{s.sillitoe,gernot}@unsw.edu.au, {abdul.basit,massimiliano.giacometti}@planv.tech

***Abstract*—We report on our ongoing work to re-establish the WCET analysis of the latest version of the verified seL4 microkernel on the 64-bit RISC-V architecture. By modifying the Heptane analysis tool to include missing features and remove scalability problems, we can now perform the analysis on an unmodified kernel, currently using place-holder latencies and loop bounds. Establishing high-assurance loop bounds and infeasible-path refutations is in progress, as is determining sound instruction latencies.**

1. Original WCET Analysis of seL4

Fifteen years ago, Blackham et al. performed the first sound and complete worst-case execution time (WCET) analysis of a protected-mode operating system (OS) kernel [2]. They performed the analysis on the Armv7 (32-bit) version of the seL4 microkernel, whose C implementation had been proved functionally correct [6]. That verification made seL4 a high-assurance foundation for safety- and security-critical systems, and thus an apt target for WCET analysis. Later, Sewell et al. strengthened the analysis by proving loop bounds and infeasible-path refutations on the source-code level, and then using a translation-validation toolchain to pass these into the WCET analysis [16].

This earlier work was based on the Chronos tool [7], albeit heavily modified to deal with the complexities of the kernel's 10 k source lines of code (SLOC) code base. However, Chronos became unmaintained even before that line of work was finished. Furthermore, with the introduction of out-of-order (OoO) cores, Arm discontinued publishing instruction latencies even for their in-order (IO) cores,[1] making it infeasible to maintain seL4's WCET analysis. Consequently, seL4 lost any hard real-time guarantees.

2. seL4 Developments

seL4 has undergone significant development since this early work, which not only means that the original WCET analysis has become completely meaningless, but also that re-establishing it has become non-trivial.

One major development is the move to 64-bit processors, especially Armv8 and RV64 (the 64-bit variant of RISC-V), including re-verification [15]. While the larger word size carries the risk of introducing scaling issues in static-analysis tools, support for the open RISC-V architecture with readily available open-source processor implementations [11], opens new avenues for obtaining latency information.

The second major change was the introduction of the *mixed-criticality systems* (MCS) variant, which (among other things) elevates time to a first-class resource with capability-authorised budgets [8]; the MCS variant has just been verified [14].

3. Resurrecting the WCET Analysis

We recently embarked on an effort to re-establish seL4's WCET analysis. Our past experience made it clear that we needed an open-source analysis tool to be able to deal with the challenges posed by a real-world OS kernel. WCET analysis is generally performed on relatively simple control code, as well as benchmark suites that are also fairly simple [4].

We selected Heptane [5] as the most promising candidate. Alternatives include OTAWA [1], which seems unmaintained,[2] and Platin [9], whose integration with LLVM/clang is a disincentive for us.

As expected, Heptane required some effort to get it to the point where it was able to process the complete seL4 kernel. The specific areas requiring work were:

- Heptane's support for the RISC-V architecture was incomplete. While sufficient for analysing application code, it lacked support for the instructions accessing the control and status registers (CSRs) as specified in the "Zicsr" extension to the base architecture [13], certain other supervisor-mode instructions, and a number of arithmetic and shift variants emitted by the compiler for kernel code. We added support for the missing instructions, the supervisor CSRs they reference, and the corresponding instruction formats. We also replaced the MIPS-derived RISC-V data-address simulation with a native implementation.
- The tool lacked support for control-flow patterns that are required by the kernel. For example, while usermode execution completes by returning from the

1. Arm later resumed publishing instruction latencies for their IO cores.

2. We received no response to our attempts to contact the OTAWA maintainers.

`main()` function, the kernel returns from a system call by executing a specific instruction sequence ending in `sret` which performs the mode switch back to the user. This must be treated as a terminal node in the control-flow graph (CFG) rather than a normal return. Separately, the kernel relies on non-local jumps that bypass the standard call/return structure. Its call graph also contains bounded recursion, requiring configurable recursion bounds in the context-tree expansion.

- Important compiler optimisations were not supported, such as tail and sibling calls, where function returns bypass the original caller.
- Heptane's data-cache analysis originally only tracked accesses on the stack and through global pointers, while seL4 commonly dereferences arbitrary pointers. We extended the underlying data-address analysis with a pointer-keyed memory map, which the cache analysis then consumes unchanged.
- Parts of the implementation were too simplistic to scale to the size and complexity of the seL4 code base. This includes the fixed-point iteration algorithm for the cache analyses, which, while theoretically sound, had high computational complexity. We achieved significant speed-up by re-engineering this algorithm to leverage an ancillary data structure of the fully inlined control-flow graph and to process nodes in a more efficient order, avoiding redundant recomputation at CFG merge points. Additionally, we removed redundant ILP variable declarations, fixed memory leaks by adopting RAII idioms, and avoided copying large amounts of analysis state.

In the end, while substantial changes were required (102 files changed, with 5,182 SLOC added and 4,918 SLOC removed), the required effort was less than we feared. The first author, a recent graduate with no prior experience with WCET analysis nor formal methods in general, spent about five to six months of full-time work on the project, to the point where the tool analyses the complete RV64 code base of seL4.

For just over 11,000 SLOC, the total run time is approximately 3 h (around 1 h of analysis and a further 2 h for ILP solving). This is consistent with the experience from our earlier work [2], which took >4h (on older hardware) for a somewhat simpler kernel of 8,700 SLOC.

4. Present State

We analyse the 64-bit RISC-V kernel built from the verified MCS configuration (`RISCV64_MCS_verified.cmake`) at seL4 commit `deec818`, with the compressed-instruction extension (RVC) disabled and the compiler directed not to emit jump tables (`-fno-jump-tables`), so that the binary is amenable to the analysis.

While we can now analyse the complete kernel, the numeric WCET results are currently fairly meaningless, as we have set it up with placeholder latencies and loop bounds. Specifically we assume 3-cycle latency for all non-memory instructions, 1-cycle L1-cache hit, 10-cycle L2 hit, 100-cycle L2 miss, and all loop bounds being 10. This results in a kernel WCET (i.e. worst-case latency from kernel entry for any reason until kernel exit) of about 21 M cycles. But, given the arbitrary latency assumptions, and (so far) no elimination of infeasible paths, this is no more than an order-of-magnitude estimate.

TABLE 1. PER-ENTRY WORST-CASE LATENCY AND LONGEST-PATH INSTRUCTION COUNT, UNDER THE PLACEHOLDER LATENCIES AND LOOP BOUNDS DESCRIBED IN THE TEXT.

Entry	WCET (k cycles)	Instructions (k)
System call	21,000	730
Interrupt	470	16
Exception	430	15
`Call` (fastpath)	19	0.66
`ReplyRecv` (fastpath)	27	1.05

Table 1 breaks this down by kernel entry, giving the worst-case latency and the number of instructions on the longest path for each. The overall kernel WCET is dominated by the (slowpath) system-call entry.

These figures should be read against the size of the problem. The static control-flow graph is modest: 154 functions, around 3.2 k basic blocks joined by 4.4 k edges, and some 13 k instructions, with 50 loops. The context-sensitive analysis, however, distinguishes roughly 110 k distinct calling contexts, giving 1.5 M contextual basic blocks, and the resulting integer linear program has about 3.1 M variables and 5.9 M constraints. This growth is what motivates the algorithmic changes needed to scale the analysis to the full kernel.

5. Ongoing Work

We are working towards establishing sound parameters to make the analysis result meaningful, as well as extracting more detailed WCET values.

5.1. Verified loop bounds and path refutations

We are working on re-establishing the transfer of loop bounds (and infeasible-path refutations) proved at the source-code level into the binary analysis, by resurrecting the work of Sewell et al. [16]. The toolchain now runs on the RISC-V kernel. A handful of loops walk runtime-sized kernel structures, such as capability lookup tables, message cancellation queues, and the priority-sorted scheduling-context queues introduced by the seL4 MCS variant. These have no static bound; they are instead constrained by system design, such as the kernel's preemption mechanism or the static system configuration, and will require a parameterised WCET or additional preemption points in the kernel.

5.2. Sound instruction-latency bounds

CVA6 [10] is a 64-bit RISC-V core we use in a number of projects. Its open-source nature allows us to establish sound instruction latencies using a two-step process.

We first obtain an empirical latency bound for each functional unit (FU) implemented in the execution stage of the CPU. We isolate the unit by manual inspection of the SystemVerilog source code and cross-checking against the CVA6 user manual and microarchitectural documentation. We then exercise the FU in isolation under constrained-random stimulus, using randomised assembler code targeting the individual FUs, with randomised operand patterns. We record as a *candidate bound* the maximum observed cycle distance between the input and the output valid signal, by instrumenting the SystemVerilog code. While we use Siemens Questa as simulator, any SystemVerilog-capable one would be suitable.

Obviously these empirical latencies are unsound, as simulation can only sample a subset of operand patterns and control-flow paths, and documentation can contain errors. However, simulation is an effective way to find a candidate value while avoiding pessimism.

We then encode each candidate latency as a bounded-delay SystemVerilog Assertion (SVA) and use open-source model checker SymbiYosys [17] to prove the assertion. If the proof succeeds, the candidate is confirmed as a sound upper bound to be used in Heptane. Else SymbiYosys provides a counterexample trace which refutes the claim. In this case we either tighten the harness assumptions or raise the bound to a value that can actually be proved.

This work is expected to be completed in Q1'27. Meanwhile we plan to extract approximate (unsound) latencies from simulations as an interim solution.

We also expect to perform the WCET analysis on a suitable IO Armv8 processor.

5.3. More detailed analysis and optimisation

Heptane supports context-sensitive WCET analysis, which allows us to obtain different latencies for different types of kernel entries. We have used this to distinguish between the high-level entry types, i.e. interrupts, exceptions and system calls, obtaining a separate WCET bound for each, as well as for the `Call` and `ReplyRecv` fastpaths (Table 1). Distinguishing between the individual slowpath system calls is ongoing work, and will enable even less pessimistic schedulability analysis.

Once we have actual latencies, we expect to spend some time on optimising kernel code to reduce the WCET, similar to our earlier work [3].

6. Conclusions

In summary, we have succeeded in re-establishing the WCET analysis of the latest (more complex) version of seL4 for 64-bit RISC-V. We found the Heptane tool a suitable starting point and could extend/improve it to the point where it could process the complete kernel. Our changes have been upstreamed to the public Heptane code repository [12].

Acknowledgements

This work is being performed under the PISTIs-V project, funded by the German agency *Agentur für Innovation in der Cybersicherheit GmbH* (Cyberagentur) under the *Ecosystem Formally Verifiable IT – Provable Cybersecurity* (EVIT) Program. We thank Isabelle Puaut for her help with using Heptane.

References

[1] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, Lecture Notes in Computer Science, pages 35–46. Springer, 2010.

[2] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 2011. IEEE Computer Society.

[3] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *EuroSys Conference*, pages 323–336, Bern, Switzerland, April 2012. USENIX.

[4] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In *Workshop on Worst-Case Execution-Time Analysis*, pages 137–147, Brussels, BE, July 2010. OCG.

[5] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane static worst-case execution time estimation tool. In *Workshop on Worst-Case Execution-Time Analysis*, pages 1–8. Schloss Dagstuhl - Leibniz-Zentrum Informatik, 2017.

[6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[7] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming, Special issue on Experimental Software and Toolkit*, 69(1–3): 56–67, December 2007.

[8] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM.

[9] Emad Jacob Maroun, Eva Dengler, Christian Dietrich, Stefan Hepp, Henriette Herzog, Benedikt Huber, Jens Knoop, Daniel Wiltsche-Prokesch, Peter Puschner,

Phillip Raffeck, Martin Schoeberl, Simon Schuster, and Peter Wägemann. The Platin multi-target worst-case analysis tool. In *Workshop on Worst-Case Execution-Time Analysis*, page 2:1–2:14, Lille, France, July 2024. Schloss Dagstuhl - Leibniz-Zentrum Informatik.

[10] Open Hardware Foundation. CVA6: An application class RISC-V CPU core, 2020. URL https://docs.openhwgroup.org/projects/cva6-user-manual/index.html. Accessed 2026-06-26.

[11] Open Hardware Foundation. Projects, 2026. URL https://openhwfoundation.org/projects/. Accessed 2026-06-26.

[12] Isabelle Puaut. Heptane. URL https://gitlab.inria.fr/puaut/heptane. Accessed 2026-06-26.

[13] RISC-V International. "Zics" extension for control and status register (CSR) instructions, 2026. URL https://docs.riscv.org/reference/isa/unpriv/zicsr.html. Accessed 2026-06-26.

[14] seL4 Foundation. Functional correctness proof for MCS seL4 completed on RISC-V, June 2026. URL https://sel4.systems/news/2026.html#mcs26.

[15] seL4 Foundation. Verified configurations, 2026. URL https://docs.sel4.systems/projects/sel4/verified-configurations.html. Accessed 2026-06-26.

[16] Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Systems*, 53:812–853, September 2017.

[17] YosysHQ GmbH. SymbiYosys (sby) documentation, 2023. URL https://symbiyosys.readthedocs.io/en/latest/. Accessed 2026-06-26.

Generalized Job-Level Dependencies for End-to-End Latency Guarantees and Low Runtime Overhead

Shriraj Hegde, Matthias Becker
Department of Electronics and Embedded Systems
KTH Royal Institute of Technology, Stockholm, Sweden
{shriraj, mabecker}@kth.se

***Abstract*—Modern real-time systems implement multi-rate cause-effect chains to perform a diverse array of functions while subject to tight end-to-end latency constraints. Guaranteeing end-to-end latency constraints is challenging due to over- and undersampling between tasks of the chain, and the impact the scheduler has on execution. Job-level dependencies have been proposed as a schedule-agnostic mechanism to bound the end-to-end latency of cause-effect chains. At runtime, end-to-end latency constraints are guaranteed to be met, as long as the job-level dependencies are met. However, existing approaches introduce job-level dependencies per producer-consumer pair hyperperiod, which may result in an overstrained system, as any added dependencies between job pairs repeat, often unnecessarily, every hyperperiod of the pair. This paper revisits the definition of job-level dependencies and explores how a larger repetition interval can minimize runtime overhead while still meeting end-to-end latency constraints.**

We evaluate the improvement through the generalized job-level dependencies using a heuristic from the literature and an extensive dataset of 70,000 synthetically generated automotive cause-effect chains. Our results demonstrate that expanding the analysis scope to the hyperperiod of the chain yields a reduction of over 90% in absolute synchronization constraints. This substantial reduction in RTOS overhead is achieved without significantly sacrificing latency minimization capabilities.

***Index Terms*—Real-Time Systems, Cause-Effect Chains, Job-Level Dependencies**

I. Introduction

Real-time systems must react to environmental stimuli within strict temporal bounds. This is achieved through tasks, which are isolated units of computation subject to rigorous timing requirements. Failure to meet deadlines in critical systems can lead to catastrophic consequences. Complex functionalities are realized through cause-effect chains, which are sequences of communicating tasks. While individual tasks must meet their respective deadlines, the cause-effect chain as a whole must also satisfy end-to-end latency constraints dictated by the physics of the system. Guaranteeing these end-to-end constraints is non-trivial, as tasks may be shared by different chains and operate at different execution periods, forming *multi-rate* task chains [1], [2].

The increasing complexity of automotive control algorithms necessitates these multi-rate topologies. However, multi-rate chains introduce challenges such as oversampling (data is read multiple times before being updated), undersampling (data is overwritten before consumption), and timing anomalies, where localized execution improvements counter-intuitively increase global latency [3]. Consequently, a cause-effect chain may violate its maximum data age constraint even if all constituent tasks meet their individual deadlines.

While alternative approaches like the Logical Execution Time (LET) paradigm achieve temporal determinism by enforcing rigid read/write delays at period boundaries at the cost of latency, the implicit communication paradigm reads and writes data dynamically at execution time to minimize delay. Consequently, under implicit communication, cause-effect chains inherently lack chronological ordering between producer and consumer jobs. A consumer job scheduled immediately after a producer completes will read significantly fresher data than a consumer scheduled just before the producer executes. As a result, the job-level data flow is unpredictable under dynamic scheduling and implicit communication.

A schedule-agnostic analysis for the maximum data age has been proposed by identifying all possible data flow paths in multi-rate chains [4]. To prevent latency violations, they propose applying Job-Level Dependencies (JLDs), which are precedence constraints that require specific producer jobs to finish before corresponding consumer jobs begin, thereby pruning data paths that could otherwise violate the data age constraint. Due to the approach's schedule-agnostic nature, this can be done early in the design phase without extensive system information, and it effectively translates the problem of guaranteeing end-to-end latency into the synthesis of job-level dependencies, while leaving implementation choices entirely unrestricted.

However, while the enforcement of JLDs at runtime has been explored, little work has focused on generating JLDs. The heuristic in [4] enforces these dependencies relative to the producer-consumer pair hyperperiod (H_{pair}). Crucially, not all resulting precedence constraints are necessary at runtime to enforce the end-to-end latency constraint. Consequently, this baseline heuristic over-constrains the system, introducing redundant RTOS synchronization overheads that severely restrict the scheduler's freedom and affect the WCET analysis [5].

The paper presents the following contributions:

- A generalized definition of job-level dependencies is proposed, which allows the specification of the repetition interval.
- We demonstrate the benefits of expanding the enforcement scope of the JLD synthesis from the local pair hyperperiod to the global chain hyperperiod (H_{chain}).

- We evaluate this expanded scope using synthetic automotive cause-effect chains and demonstrate that the heuristic's success rate remains unchanged, while significantly fewer precedence constraints need to be realized at runtime, thus reducing the number of synchronization calls.

II. Related Work

For many embedded systems, correct functionality requires bounded end-to-end latency for data propagating through a chain of semantically related tasks [1], [6]. As a consequence, cause-effect chains have been extensively studied in the literature, and a large body of research is dedicated to the analysis of their end-to-end latency [3], [4], [7]–[10].

Other approaches focus on the configuration of cause-effect chains to influence the execution of their tasks, thereby minimizing or reducing end-to-end latency. Zhu et al. [11] utilize job-level precedence constraints between tasks to enforce deterministic data flow and eliminate timing anomalies under the implicit communication model. Günzel et al. [12] show how to configure the phasing between tasks in a cause-effect chain with semi-harmonic periods, to minimize the end-to-end latency under the Logical Execution Time (LET) paradigm. Paladino et al. [13] propose to use publisher tasks to remove the jitter of cause-effect chains under LET. The approach also enables efficient latency calculation, which is used to assign optimal task priorities. Maia and Fohler [14] optimize the end-to-end latency of cause-effect chains under LET by leveraging knowledge from later design phases, such as schedule information, to adjust LET intervals.

While the above approaches observe characteristics of the scheduling algorithm or communication semantics to optimize latency, Job-Level Dependencies (JLDs) have been proposed as a schedule-agnostic alternative [4]. By enforcing precedence constraints between specific task instances, the heuristic bounds the maximum end-to-end latency without altering the underlying scheduling policy [4], [15]. The heuristic mainly addresses single cause-effect chains, which negatively impact task sets where the same task is shared across multiple cause-effect chains, as noted by Verucchi et al. [16].

As JLDs are scheduling-agnostic, several works address the realization of job-level dependencies under different scheduling algorithms and platforms. Klaus et al. [5] investigate the realization of JLDs under partitioned EDF. Their work highlights the increased runtime overhead when JLDs are realized by OS synchronization primitives. As a motivating example, they report a 2.27% increase in utilization after adding two JLDs (using semaphores) to a FreeRTOS-based application running on a CORTEX-M4 @ 80MHz. Subsequently, they transform the parameters of individual jobs such that the required job ordering is maintained at runtime, while semaphore-based synchronization is used for JLDs between tasks on different cores. Forget et al. [17] demonstrated that, also under fixed-priority scheduling, precedence constraints can be realized entirely without synchronization mechanisms by encoding them directly into task attributes. An alternative approach statically realizes JLDs by merging dependent tasks into multi-frame tasks [18]. Denzinger et al. [2] use a similar approach to additionally consider safety criticality levels as part of their holistic approach for the early design phase of automotive applications. Beyond explicit runtime synchronization, Ruh and Craciunas utilize JLDs to enforce end-to-end timing guarantees when generating offline schedules for virtualized distributed systems [19]. In this paradigm, JLDs are applied purely as offline scheduling constraints, avoiding dynamic synchronization primitives entirely.

While several approaches address the realization of JLDs in online scheduling by modifying task parameters, they are generally more challenging for practitioners to realize than semaphore-based synchronization. Thus, we revisit the definition of JLDs and their synthesis to demonstrate that the number of synchronization points required at runtime can be dramatically decreased by enlarging the repetition interval of JLDs, thereby reducing the runtime overheads of semaphore-based synchronization methods.

III. System Model

The real-time system is modeled as a set of periodic, time-triggered, and preemptive tasks. All tasks are synchronously released at $t = 0$. A task τ_i is represented by a tuple (T_i, C_i^+). Here, T_i denotes the fixed activation interval or the period of the task and C_i^+ denotes the *Worst-Case Execution Time* (WCET). While both the WCET and the *Best-Case Execution Time* (BCET) are formally required to calculate the exact data propagation intervals between communicating jobs, we conservatively assume a BCET of zero ($C_i^- = 0$) throughout the paper. The j^{th} job of task τ_i is denoted by $\tau_{i,j}$ ($\tau_{i,1}$ is released at $t = 0$).

The set of all tasks in an application is denoted by Γ and the hyperperiod H is the *Least Common Multiple* (LCM) of the time periods of the tasks in Γ, after which the pattern of job releases (schedule) repeats. The tasks communicate using the implicit communication model via shared registers. Each job reads all input data at the beginning of the execution and writes the output when it finishes execution (*read-execute-write semantics*) [4]. The time span during which a job $\tau_{i,j}$ can read input data is bounded by its *Read Interval* $[R_{min}(\tau_{i,j}), R_{max}(\tau_{i,j})]$, where R_{min} is the job's release time and R_{max} is its absolute deadline minus C_i^+. Conversely, the time span during which its produced output data is available for consumption is bounded by its *Data Interval* $[D_{min}(\tau_{i,j}), D_{max}(\tau_{i,j})]$.

A *Cause-Effect Chain* E is a sequence of communicating tasks, represented as an ordered sequence of n tasks (where n denotes the chain length) that describe the data flow in an application, i.e., $E = (\tau_1 \rightarrow \cdots \rightarrow \tau_n)$. While a task can be shared by multiple cause-effect chains in the broader system [1], each chain is analyzed here as an independent linear sequence. To simplify notation, the remainder of the paper is described for a generic chain E. The hyperperiod of a chain E is defined as $H_{chain} = \text{LCM}(\forall \tau \in E)$. Cause-effect chains may contain several tasks with different

execution rates. The total delay from input data to output availability is broadly referred to as the *End-to-End Latency*. This work focuses on the Maximum Data Age (MDA), though results apply to other latency metrics. Conceptually, data age quantifies the maximum duration a sampled input can influence the system's output. Let $paths$ denote the set of all valid job-level data propagation paths, originating at a source job $\tau_{1,x}$ and terminating at a sink job $\tau_{n,y}$. An end-to-end data age constraint restricts the age of the input data at the exact moment the final output is produced. Analytically, the MDA is bounded by the longest data propagation path: $\text{MDA} = \max_{\forall(x,y)\in\text{paths}} \Big((R_{max}(\tau_{n,y}) + C_n^+) - R_{min}(\tau_{1,x}) \Big)$. This measures the elapsed time from the initial release of any source job (earliest sampling) to the worst-case completion time of the corresponding sink job.

A *Job-Level Dependency* (JLD) as described by [4], is a repeating precedence constraint between two specific jobs of successive tasks in a cause-effect chain. A JLD is represented by the notation $\tau_i \xrightarrow{(k,l)} \tau_{i+1}$, which indicates that the k^{th} job of τ_i must finish execution before the l^{th} job of τ_{i+1} can begin execution. The independent job indices k and l are bounded by the number of task activations within the chosen repetition scope. This precedence constraint ensures that $\tau_{i+1,l}$ consumes data no older than that produced by $\tau_{i,k}$. In multi-rate systems, data can flow through many paths between various jobs of the tasks. Some of these data paths may violate the maximum allowed latency. Hence, the goal is to augment the system by adding constraints (in the form of JLDs) to eliminate paths that do not meet the deadline [4].

A practical way to realize JLDs is to use semaphore-based synchronization primitives. However, enforcing JLDs at run-time with synchronization mechanisms inherently introduces synchronization and scheduling overheads proportional to the number of required precedence constraints [5]. Another critical goal of this work is to minimize the absolute number of precedence constraints that result from the synthesized JLDs. Therefore, it is imperative to prune only the specific data paths that violate the deadline, actively avoiding unnecessary dependencies to prevent the degradation of overall system performance.

IV. Method

This section first presents a brief recap of the JLD heuristic introduced in [4], before presenting the proposed generalized JLD.

A. Recap of the Baseline JLD Synthesis Heuristic

To contextualize our modifications, we briefly outline the baseline JLD synthesis heuristic [4]. A Data Propagation Tree (DPT) is constructed for each root job of a task chain (i.e., all jobs of the first task in the chain released within H_{chain}). A node represents a task job, while an edge represents possible communication between two jobs. Within this tree, each node may branch to multiple successor jobs, but retains exactly one incoming edge. To eliminate paths exceeding the deadline, the heuristic iteratively prunes the invalid branches using a bottom-up approach.

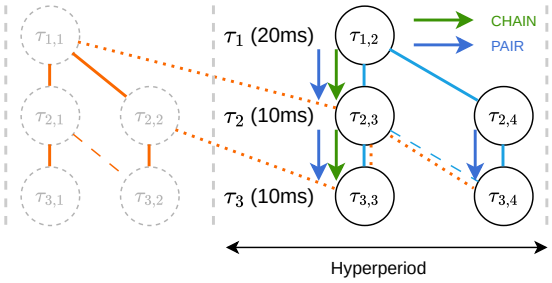


Fig. 1. Data propagation paths and synthesized Job-Level Dependencies for an example cause-effect chain. The figure overlays the PAIR and CHAIN scopes to highlight their divergence. Solid lines represent valid paths. Dotted lines indicate violating paths pruned by both approaches. Dashed lines highlight a path unnecessarily pruned by a redundant constraint generated under the PAIR scope. The solid arrows represent the precedence constraints generated by the heuristic.

First, it traverses the DPT to identify a violating leaf node ($\tau_{n,y}$). Second, it traces backwards from this leaf node to find its first parent ($\tau_{i,x}$) that acts as a structural branching point (i.e., a parent with multiple outgoing data paths). Finally, a Job-Level Dependency is enforced between the parent's subsequent job ($\tau_{i,x+1}$) and the direct successor job ($\tau_{a,b}$) on the violating path. This precedence constraint forces $\tau_{a,b}$ to wait for fresher data from $\tau_{i,x+1}$, effectively severing the invalid data paths. This is repeated until all DPT paths satisfy the latency bound or failure.

In [4], the scope of a synthesized JLD between a producer task τ_p and a consumer task τ_c is implicitly defined relative to their pair hyperperiod, i.e., $H_{pair} = \text{LCM}(T_p, T_c)$. Hence, the added JLD repeats every H_{pair}.

However, in multi-rate automotive cause-effect chains, communicating task pairs may involve large period transitions, such as from 1000 ms to 1 ms [1]. By definition, H_{pair} divides H_{chain}. Consequently, placing a dependency at the H_{pair} scope forces the constraint to instantiate H_{chain}/H_{pair} times within a single global execution cycle of the chain, whereas possibly only one dependency is required to satisfy the end-to-end latency constraint. This leads to unnecessarily constrained systems and synchronization overheads.

B. Generalized Job-Level Dependencies

To reduce redundant synchronizations without modifying the core pruning logic, we propose expanding the repetition interval of a JLD beyond the hyperperiod of the local task pair. We therefore introduce a generalized JLD:

$$\tau_p \xrightarrow[\alpha]{(k,l)} \tau_c \tag{1}$$

As for the JLDs in [4], the JLD describes a precedence constraint between the k^{th} job of τ_p and the l^{th} job of τ_c. In addition, α denotes the interval at which the JLD repeats, where $\alpha/H_{pair} \in \mathbb{N}$. By setting $\alpha = H_{pair}$, we obtain the

baseline scope proposed in our earlier work [4], which we denote as PAIR. By setting $\alpha = H_{chain}$, we get a JLD that instead repeats with the hyperperiod of the chain, which we denote as CHAIN. While larger values for α are possible, such as the hyperperiod of the task set, $\alpha = H_{chain}$ seems particularly suitable for constraining the data age of a cause-effect chain. In this case, an added JLD repeats by design only with the chain hyperperiod, which is also the interval over which all root jobs considered by the heuristic are released.

We retain the baseline synthesis heuristic [4] for generalized JLDs, including its pruning point selection and backtracking strategy, and change **only** the scope over which JLD patterns are defined. When the heuristic traces a deadline violation backward to the deepest branching point, it identifies a specific producer job $\tau_{p,x}$ and consumer job $\tau_{c,y}$ whose data path must be pruned. To enforce this, a generalized JLD is instantiated. The indices k and l of the generalized JLD can be calculated based on α. The relative indices k and l project the age violation (absolute job indices x and y) into the future periodic schedule.

$$k = \left((x-1) \bmod \frac{\alpha}{T_p} \right) + 1 \text{ and}$$
$$l = \left((y-1) \bmod \frac{\alpha}{T_c} \right) + 1 \tag{2}$$

Here, x and y represent the absolute physical indices of the two jobs involved since system startup ($t = 0$), and T_p and T_c are the periods of the producer and consumer tasks involved.

While the generalized JLD with $\alpha = H_{chain}$ significantly reduces the number of required runtime synchronization calls, it incurs a potential cost to heuristic runtime. The H_{chain} approach must explicitly identify and add individual dependencies wherever constraints are actually necessary across the schedule, rather than relying on aggressive duplication over the pair hyperperiod.

Fig. 1 illustrates a cause-effect chain τ_1 (20 ms) $\rightarrow$ τ_2 (10 ms) $\rightarrow$ τ_3 (10 ms) and the heuristic evaluated against a 25 ms deadline. Without constraints, delayed data paths (dotted lines) process stale data from older task instances (faded circles), resulting in deadline violations. To prune these paths, precedence constraints are synthesized to enforce the faster $\tau_{1,2} \rightarrow \tau_{2,3} \rightarrow \tau_{3,3}$ propagation path. While both scopes successfully resolve the deadline violation, the PAIR scope blindly repeats constraints every $H_{pair} = 10$ ms. As shown, this generates a superfluous constraint ($\tau_{2,4} \rightarrow \tau_{3,4}$) that unnecessarily prunes the path ($\tau_{2,3} \rightarrow \tau_{3,4}$) which naturally finishes by 20 ms and satisfies the deadline on its own. Conversely, the CHAIN scope synthesizes only the strictly necessary constraints, successfully eliminating the redundant synchronizations.

V. Evaluation

This section presents the evaluation of the proposed CHAIN approach against the baseline PAIR approach. All experiments applied a target latency deadline set to 50% of the unconstrained maximum data age. This threshold forces the heuristic to actively prune paths and halve latency while remaining schedulable for almost all configurations. This ensures a large sample size, isolating the runtime overhead differences between PAIR and CHAIN for comparison.

A. Experimental Setup

To assess the efficacy of the proposed global scope heuristic, we conducted an evaluation on a comprehensive dataset of 70,000 synthetic cause-effect chains generated according to established automotive benchmarks [1]. To ensure balanced representation across different chain lengths, the dataset contains exactly 5,000 chains for each chain length from 2 to 15 tasks. The initial activation period of each chain is sampled uniformly from the standard automotive set $\mathcal{P} = \{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ ms. For multi-rate chains, subsequent periods and inter-task period transitions are sampled according to the communication frequency class distributions reported by Kramer et al., with the geometric means of the logarithmic signal count bins (classes I-VI) used as transition weights. Task Worst-Case Execution Times (WCETs) are derived from exponentiated Weibull distributions as discussed in [1]. Enforcing an equal distribution of chain lengths from two to 15 tasks alters the original Kramer distribution of activation patterns (originally 70% single-rate, 20% two-period, 10% three-period). Specifically, to construct an equal number of chains for each length, the tasks per period distribution (originally from Table VII of [1]) is relaxed to a uniform distribution, and the period count is selected uniformly or forced to three periods for lengths ≥ 11. As a result, the generated dataset contains 17,368 single-rate (1-period) chains (24.8%), 18,205 two-period chains (26.0%), and 34,427 three-period chains (49.2%). Length constraints from Kramer et al. [1] (Sec. VII-A) are strictly satisfied: two-period chains contain 4-10 tasks, and three-period chains contain 6-15 tasks, with identical periods kept in contiguous blocks. This ensures chains with fewer than four tasks are strictly single-rate, validating the baseline behavior ($H_{pair} = H_{chain}$), as shown in Fig. 2.

B. Reduction in Absolute Synchronization Overhead

We evaluate the proposed extension primarily by quantifying its practical implementation overhead: the concrete count of OS synchronization primitives (e.g., semaphores) that must fire during the hyperperiod of the chain H_{chain}.

First, we report on the synthesis success rate (schedulability) of the two heuristics under the target deadline. Out of the 70,000 generated cause-effect chains, both heuristics successfully synthesized valid JLD constraints for 69,652 chains (99.50%), while neither could find a valid solution in 299 cases (0.43%). In a minimal fraction of edge cases (49 chains in total, or 0.07%), the baseline PAIR approach succeeded alone in 44 cases (0.06%), whereas the global CHAIN scope succeeded alone in 5 cases (0.01%). This confirms that expanding the synchronization enforcement scope does not practically compromise the schedulability or synthesis capability of the underlying heuristic search. Importantly, neither approach

strictly dominates the other. Analyzing failures by chain length reveals that cases where at least one approach failed to find a valid solution are concentrated entirely in intermediate chain lengths (4 to 12 tasks), peaking at 1.54% (77 cases) for 10-task chains, whereas chains of lengths 2-3 and 13-15 exhibit a 100% success rate. Because comparing constraint overheads and runtimes requires that both approaches yield a valid solution, all subsequent comparative plots and metrics are computed over the 69,652 chains in which both heuristics succeeded.

Fig. 2 reports the absolute synchronization points needed under varying chain length, separated by the number of activation patterns of the chain. The over-constraining nature of the baseline PAIR approach scales directly with the period diversity of the cause-effect chain. For single-rate chains (1-period), the CHAIN approach safely degenerates to the PAIR approach, yielding identical constraints. However, as the number of unique periods increases to three, the CHAIN scope reduces the absolute number of instantiated JLDs by a large magnitude.

Fig. 3 illustrates the absolute number of generated constraints across all evaluated chains, sorted by the baseline PAIR constraint count. For simple topologies, the CHAIN heuristic operates closely to the baseline, yielding similar synchronization points (visible as the overlapping lower region). However, as topological complexity increases, the baseline constraint count explodes rapidly. In contrast, the CHAIN heuristic actively suppresses this bloat; the diverging area between the two curves represents the direct reduction in required synchronizations.

The severity of this problem can be illustrated by the following example. Consider five tasks τ_1 to τ_5 with periods $(1000, 1000, 1, 1, 1)$ ms in a chain with deadline $1501\,\mathrm{ms}$. Both approaches synthesize four JLDs. Under PAIR, the two JLDs between $1\,\mathrm{ms}$ tasks have $H_{pair} = 1\,\mathrm{ms}$, instantiating 1,000 constraints each per H_{chain}, totalling 2,002 synchronization calls. Under CHAIN, all four JLDs are anchored to H_{chain} and fire exactly once, yielding only four calls: a 500x reduction in RTOS runtime overheads while meeting the same deadline.

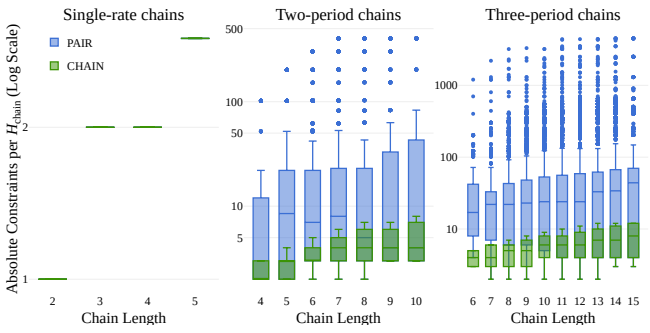


Fig. 2. Distribution of Absolute Synchronization Points by Task Count and Period Diversity. The CHAIN approach reduces absolute synchronization constraints across all multi-rate configurations.

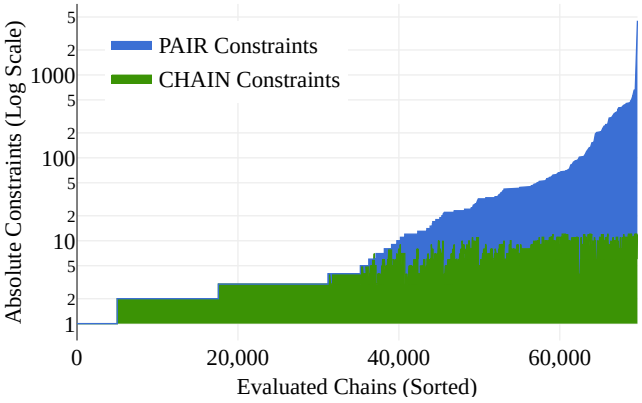


Fig. 3. Absolute constraint generation across all evaluated chains (log scale), sorted by the baseline PAIR constraint count.

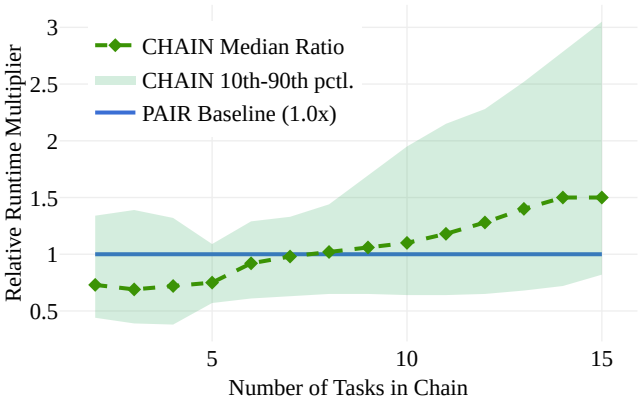


Fig. 4. Normalized synthesis runtime versus chain length. The baseline PAIR approach serves as the 1.0x reference.

C. Synthesis Runtime

Both the PAIR and CHAIN heuristics rely on recursive backtracking to prune the data propagation tree, meaning they fundamentally exhibit worst-case exponential asymptotic complexity with respect to the topological complexity of the cause-effect chain. To provide context for the synthesis time, the absolute synthesis runtimes (observed during the evaluation on a modern laptop processor) for PAIR were highly right-skewed, with a median of $32\,\mathrm{ms}$, a mean of $2.6\,\mathrm{s}$, and a 99th-percentile worst-case of $48.6\,\mathrm{s}$. The CHAIN approach exhibited a comparable median of $32\,\mathrm{ms}$, with a mean of $3.9\,\mathrm{s}$ and a 99th-percentile of $68.5\,\mathrm{s}$. This variance highlights that while deep chain topologies occasionally demand substantial computational effort, the typical chain is synthesized in milliseconds. Note that these absolute time values are heavily dependent on the specific implementation, algorithm optimizations and the random seed chosen during chain generation. The values demonstrate that marginal increase in offline analysis time for worst-case topologies is an entirely acceptable trade-off to achieve a significant reduction in online RTOS synchronization overhead.

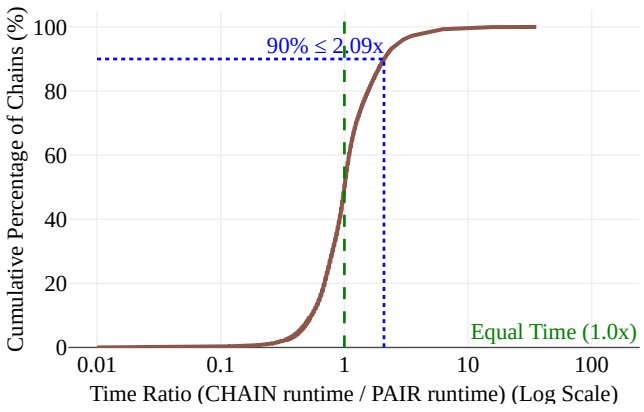


Fig. 5. Cumulative Distribution Function (CDF) of the synthesis time ratio.

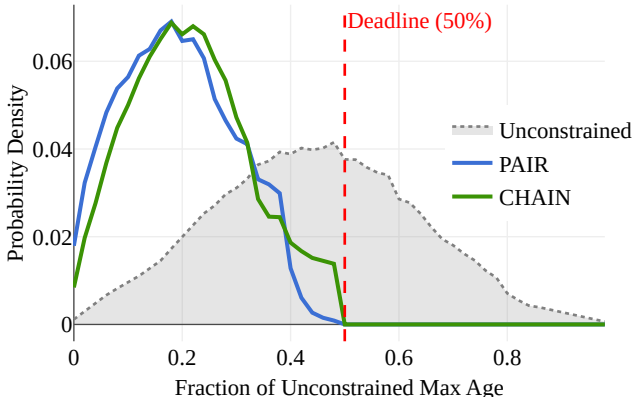


Fig. 6. Aggregated path age distribution across all evaluated chains. The vertical red dashed line indicates the target deadline (50% of the unconstrained maximum age).

To directly compare their algorithmic scaling behavior, Fig. 4 plots the execution times of the CHAIN heuristic normalized to the baseline PAIR approach. While both algorithms exhibit exponential absolute runtime growth with increasing chain length, Fig. 4 illustrates their relative divergence. Interestingly, the CHAIN approach exhibits non-linear relative scaling behavior. For short chains, CHAIN outperforms the baseline by avoiding localized constraint bloat. As chain depth increases, the overhead of repeatedly tracing paths across the chain hyperperiod introduces a computational penalty, slowing synthesis relative to the baseline.

Fig. 5 demonstrates the computational overhead incurred by CHAIN across the entire dataset. For 90% of all evaluated chains, the CHAIN heuristic incurred a computational multiplier of less than 2.1x compared to the baseline PAIR approach, with a median multiplier of 1.0x (indicating no additional overhead for at least half of the dataset), proving that the overhead of global path traversal is largely negligible in practice.

D. Timing Safety and Path Age Distribution

To confirm that the constraint reduction of the CHAIN scope does not compromise timing safety, Fig. 6 shows the probability density of path ages across all 70,000 cause-effect chains. Path ages are normalized by the unconstrained maximum data age of each respective chain to allow aggregation.

In the unconstrained case, a significant portion of path ages lies beyond the 50% deadline threshold. Applying JLD constraints under either the PAIR or CHAIN scope successfully prunes these violating high-latency paths, shifting the probability density entirely to the left of the deadline. Crucially, the resulting path age distributions for both approaches are nearly identical. This demonstrates that the CHAIN scope achieves equivalent timing safety and data-age containment as the baseline approach while requiring a fraction of the synchronization points.

E. Limitation: Minimum Achievable Latency

While the CHAIN scope drastically reduces the absolute constraints required to guarantee timing safety, both PAIR and CHAIN use the same underlying data propagation tree to find the worst-case path. In a separate experiment on 5,000 chains, a binary search revealed that the tightest achievable deadline (the latency floor) is virtually identical between PAIR and CHAIN (differing in small edge cases where PAIR's aggressive constraints provided a marginal benefit). This confirms that the contribution of the CHAIN scope is strictly RTOS overhead reduction rather than latency minimization.

VI. Conclusion and Future Work

This paper addresses and attempts to solve the issue of synchronization overhead caused by job-level dependencies by minimizing the absolute number of synchronizations required at runtime. We propose a generalized JLD and evaluate enforcement scope from the local pair-hyperperiod (H_{pair}) to the global chain-hyperperiod (H_{chain}). We demonstrate that the significant constraint bloat in the baseline heuristic can be safely removed without significant drawbacks. Our evaluation of 70,000 synthetic automotive chains shows that this simple scope expansion yields a 90.7% reduction in absolute synchronization primitives for highly diverse, realistic multi-rate chains.

Future work must address the greedy nature of the heuristic and its isolated and local optimization of linearized chains, a limitation identified by Verucchi et al. [16] that our current scope expansion does not resolve. Recent advances have successfully applied Deep Learning techniques to generate precedence constraints for DAG tasks [20]. Although applied in the different context of sub-task scheduling, utilizing similar learning-based techniques on Data Propagation Trees could theoretically bypass exhaustive backtracking search to and further optimize the generated JLD constraint sets.

Acknowledgement

This work was partially supported by the Swedish Research Council (VR) under project No. 2023-04773.

REFERENCES

[1] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *6th International workshop on analysis tools and methodologies for embedded and real-time systems (WATERS)*, 2015, p. 43.

[2] T. Denzinger, M. Becker, and P. Ulbrich, "From timing budgets to WCETs: Robust SIL- and BSW-aware clustering and allocation for iterative automotive software development," in *38th European Conference on Real-Time Systems (ECRTS)*, 2026.

[3] M. Günzel, K.-H. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J.-J. Chen, "Timing analysis of asynchronized distributed cause-effect chains," in *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 40–52.

[4] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2016, pp. 159–169.

[5] T. Klaus, M. Becker, W. Schröder-Preikschat, and P. Ulbrich, "Constrained data-age with job-level dependencies: How to reconcile tight bounds and overheads," in *27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 66–79.

[6] S. Mubeen and T. Nolte, "Applying end-to-end path delay analysis to multi-rate automotive systems developed using legacy tools," in *IEEE World Conference on Factory Communication Systems (WFCS)*, 2015, pp. 1–4.

[7] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2009.

[8] T. Kloda, A. Bertout, and Y. Sorel, "Latency upper bound for data chains of real-time periodic tasks," *Journal of Systems Architecture*, vol. 109, p. 101824, 2020.

[9] M. Günzel, H. Teper, K.-H. Chen, G. von der Brüggen, and J.-J. Chen, "On the equivalence of maximum reaction time and maximum data age for cause-effect chains," in *35th Euromicro Conference on Real-Time Systems (ECRTS)*, 2023, pp. 10–1.

[10] M. Günzel, H. Teper, G. v. d. Brüggen, and J.-J. Chen, "End-to-end latency of cause-effect chains: A tutorial," *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 1, pp. 1–18, 2024.

[11] Y. Zhu, B. Zhang, Y. Gao, H. Ren, C. Tang, C. Zhao, L. Gong, T. Wang, W. Lou, and X. Li, "Scheduling cause-effect chains without timing anomalies in end-to-end latency," *arXiv preprint arXiv:2604.09102*, 2026.

[12] M. Günzel and M. Becker, "Optimal task phasing for end-to-end latency in harmonic and semi-harmonic automotive systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2025, pp. 164–176.

[13] F. Paladino, A. Biondi, E. Bini, and P. Pazzaglia, "Optimizing Per-Core Priorities to Minimize End-To-End Latencies," in *36th Euromicro Conference on Real-Time Systems (ECRTS)*, 2024, pp. 6:1–6:25.

[14] L. Maia and G. Fohler, "Reducing end-to-end latencies of multi-rate cause-effect chains in safety critical embedded systems," in *12th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2024.

[15] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Mechaniser-a timing analysis and synthesis tool for multi-rate effect chains with job-level dependencies," in *Workshop on analysis tools and methodologies for embedded and real-time systems (WATERS)*, 2016.

[16] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna, "Latency-aware generation of single-rate dags from multi-rate task sets," in *IEEE 26th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 226–238.

[17] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010, pp. 301–310.

[18] M. Becker, "Meeting job-level dependencies by task merging," in *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 792–798.

[19] J. Ruh and S. S. Craciunas, "End-to-end schedulability of virtualized distributed time-triggered systems," in *32nd International Conference on Real-Time Networks and Systems (RTNS)*, 2024, pp. 242–254.

[20] B. Sun, M. Theile, Z. Qin, D. Bernardini, D. Roy, A. Bastoni, and M. Caccamo, "Edge generation scheduling for dag tasks using deep reinforcement learning," *IEEE Transactions on Computers*, vol. 73, no. 4, pp. 1034–1047, 2024.

OSPERT 2026 Program

	Tuesday, July 7, 2026
8:00 – 9:00	Registration & Welcome Desk
9:00 – 10:00	**Opening Remarks and Keynote.** Chair: Antonio Paolillo Keynote: Throughput vs Low Latency on Modern Hardware: an OS Developer Perspective *Elad Lahav, Lead Architect, QNX*
10:00 – 10:30	**Session 1: Securing & Sharing Embedded Platforms.** Chair: Antonio Paolillo Dolia: an MPU-backed Framework for Secure Multitenancy on Embedded MCUs *Julien Gourgue, Cristel Pelsser, Ramin Sadre*
10:30 – 11:00	Coffee Break
11:00 – 12:30	**Session 2: Interference on Shared Resources (NoC & GPU).** Chair: Joshua Bakita Interference-Free Channels: OS Support for Real-Time NoC Communication *Viktor Reusch, Michael Roitzsch* Partition-Sensitive Interference on NVIDIA GPUs: 500,000 Ways to Exceed WCETs *Sarah Crowder, Syed W. Ali, James H. Anderson* Scheduling Behaviors of the CUDA Graph API *Theodore L. Demetriades, Rebecca Moran, Tanya Amert*
12:30 – 14:00	Lunch
14:00 – 15:30	**Session 3: Timing, WCET & Safety Guarantees.** Chair: Tanya Amert Architecting Safety with Microkernels: An Experience Report to ASIL Compliance on the Example of L4Re *Kai Lampka* Resurrecting seL4's WCET Analysis *Simon Sillitoe, Abdul Basit, Massimiliano Giacometti, Gernot Heiser* Generalized Job-Level Dependencies for End-to-End Latency Guarantees and Low Runtime Overhead *Shriraj Hegde, Matthias Becker*
15:30	Fika
16:00	RoboLab Visit
17:00	ECRTS First-timer Reception
	Wednesday, July 8th – Friday, July 10th, 2026
	ECRTS main conference.