Scheduling Constraints: A Universal OS Mechanism for Managing Shared Resources

Moritz Lumme
*ETH Zurich**
Zürich, Switzerland
moritz.lumme@inf.ethz.ch

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

Adam Lackorzyński
TU Dresden and Kernkonzept GmbH
Dresden, Germany
adam.lackorzynski@tu-dresden.de

***Abstract*—Modern cyber-physical systems consolidate subsystems onto shared multi-core hardware to meet requirements in size, weight, power and cost. This practice introduces contention for shared hardware resources, which can break performance isolation and complicate timing predictability. While many timing verification techniques for partitioning individual resources such as CPU time, caches, interconnects and power budgets have been proposed over the past decades, real-time operating systems currently lack a scalable and universal mechanism for attaching multiple of these constraints to individual system workloads and incorporating this knowledge into thread scheduling decisions. Motivated by this observation, we present a scheduling model that uses *scheduling constraints (SCs)* as a first-class operating system primitive. We show that our model allows for the flexible specification of execution environments, resource partitioning plans, and even high-level application policies which get respected by the system scheduler. We implement SCs in the L4Re real-time framework and show that they impose minimal overhead, help to bound or mitigate contention for shared resources, and add temporal isolation to the strong logical isolation guarantees of L4Re's microkernel-based design.**

***Index Terms*—Real-time Operating Systems, Shared Resource Contention, Multi-Core Interference, Timing Verification, Temporal Isolation**

I. Introduction

Modern cyber-physical systems (CPS) use multi-core processors because they offer high computational performance under constraints of size, weight and power (SWaP) [3]. However, the consolidation of diverse subsystems onto shared multi-core hardware introduces interference that originates from contention for shared hardware resources. Especially for real-time CPS, where timing predictability is key, this is problematic as it can lead to performance degradation of critical workloads [27]. The diversity of shared hardware resources such as caches, interconnects and power budgets, further complicates resource management in these systems as each resource must be managed individually.

A. Managing Resource Contention

Over the years, numerous timing verification techniques have been proposed to address the impact of resource contention on the performance of multi-core real-time systems [33]. In general, existing approaches fall into two main categories. In the first category, it is common to accept the existence of contention and incorporate this uncertainty into the schedulability analysis [10], [12], [17], [18], [23], [39], [47]. These approaches assume complete knowledge of the system to produce context-dependent correctness statements where the verification of a subsystem depends on a set of known co-runners. However, we have found the assumption of full system knowledge to be increasingly unsustainable with growing system complexity, leading to overly pessimistic resource allocation. Techniques of the second category, on the other hand, aim at context-independent verification by ensuring *temporal isolation* between subsystems [8], [14], [16], [34], [35], [42], [44], [48], [49]. The idea is to shape resource access in a way that upper bounds, fixes, or mitigates interference entirely. This helps to establish *performance isolation* where the runtime characteristics of individual subsystems can be maintained regardless of the resource usage of other co-runners in the system. A complex system can thus be represented through the composition of independently verifiable parts. This aligns well with the ideas of isolation and compartmentalization found in modern operating systems (OSes) [31]. Therefore, in this paper we focus only on techniques of the second category that can be implemented on commercial off-the-shelf multi-core hardware.

While available approaches can be found both in hardware and software, they share common limitations. We observe that existing solutions are often tailored to a specific resource, such as CPU time, caches, or memory bandwidth. Consequently, several techniques must be combined to build a system with comprehensive protection against interference. To this day, this remains to be a difficult endeavor, as illustrated by a recent industrial challenge posed by Arm [4]. In addition, we observe that existing timing verification techniques are strongly catered to a specific system architecture, platform or workload. As a result, it incurs significant design effort and runtime management overhead to combine several techniques, to apply a technique to only selected subsystems and applications, to migrate a workload between platforms that support different techniques, or to co-locate multiple workloads with different requirements. The root cause being that, from an OS perspective, there is no consistent interface for how timing verification techniques should specify their constraints on system workloads and how they interact with thread scheduling. We conclude that modern real-time OSes lack a central abstraction to overcome these limitations.

*This work was conducted while the author was at TU Dresden.

B. Contributions

Motivated by this observation, the goal of this paper is to develop a mechanism for the universal handling of shared resources in real-time OSes. As we aim to derive a general abstraction, this work omits a formal schedulability analysis, since it would be inherently tied to a particular combination of timing verification techniques. Instead, we present a portable abstraction that is compatible with a wide range of existing techniques, and we discuss its integration with real-time OS design. We make the following contributions:

1) After a brief introduction to terminology and fundamentals in Section II, we present a scheduling model for the universal management of shared resources in real-time OSes in Section III. As part of the model, we introduce *scheduling constraints* (SCs) as an efficient mechanism that allows resource availability conditions to be directly incorporated into thread scheduling decisions.
2) We demonstrate the feasibility of the model in Section IV by implementing SCs in the microkernel-based L4Re real-time framework [2]. In this section, we reflect on key design decisions and their impact on functionality and efficiency of our prototype.
3) In Section V, we evaluate the performance of our model by examining the overhead of SCs and their ability to establish temporal isolation for individual subsystems.

II. Background

A. Resource Management in Real-time Systems

In real-time systems, workloads must adhere to timing constraints. This property is essential for the design of real-time OSes, which must provide efficient and deterministic primitives to ensure minimal and predictable overhead on system workloads. To minimize interference through contention for shared resources, real-time OSes typically implement *resource partitioning* [30], where a shared resource is divided into well-defined reservations, either in space or in time, that can be allocated to consuming entities such as subsystems, applications, or threads. The control logic is encapsulated in a *resource server* [38], which monitors the resource state, accounts for consumption and mediates all accesses to the given resource. The resource server implements a concrete timing verification technique that dictates a plan for how a resource should be partitioned. It enforces this plan by controlling the runnable state of all threads involved and, for example, serializes all accesses by enabling them individually. In this paper, we do not advance real-time locking protocols, but rather, building on resource partitioning and resource servers, we present an OS-level scheduling primitive that embeds arbitrary resource constraints into scheduling decisions.

Traditionally, real-time systems were only concerned with the partitioning of CPU time as a resource. However, with increasing hardware complexity, freedom from interference now includes additional shared resources such as caches, interconnects, and power budgets, which must be partitioned as well [6]–[8], [36]. Partitioning resources beyond CPU time effectively adds more resource servers to the system [22], and thus additional constraints on a thread's runnable state. As a result, determining whether a thread is runnable at any given time becomes computationally expensive. According to its individual resource partitioning plan, each resource server has its own idea of whether a thread should be runnable or not. Only when they all agree that a thread is runnable should it be run. This computational overhead complicates timing predictability, as it varies depending on the number of techniques and type of partitioning applied to a thread's resources. In addition, changes in resource state, such as the depletion or replenishment of workload-specific reservations, are typically signaled via interrupts. Resource servers, monitoring resource state and handling these interrupts, aim to express the new state quickly and efficiently. However, when multiple resource servers are present, it requires additional global coordination for handling intricate dependencies such as blocking or unblocking and queuing or dequeuing the affected thread to and from a ready-queue. In systems where multiple threads share resource reservations, these complexities are further exacerbated and introduce non-deterministic overhead into interrupt response times that complicates accounting [50].

We raise the question whether it is possible to avoid this overhead and complexity, or at least distribute it in a way that preserves deterministic behavior. In this paper, we explore a new scheduling primitive that tries to address this challenge. Our mechanism installs an indirection between threads and resource servers, rearranging the widely used default scheduling regime. The new abstraction provides a universal interface to record resource state that can be updated and read in constant time. Thread states and thread scheduling decisions can be computed quickly and on-demand against this abstraction.

Resource partitioning and resource servers are particularly well adopted in real-time OSes with microkernel architectures. By isolating resource management in separate, dedicated components, these systems enhance both security and modularity of the overall system. For this reason, we implement our model in the L4Re framework [2], a modern real-time OS with real-world industrial deployments. In the following we provide an overview of the basic principles of L4Re.

B. L4Re Framework

The L4Re framework is a microkernel-based system architecture consisting of the L4Re microkernel, a descendant of the L4 family [29], and the L4 runtime environment (L4Re). The kernel follows the principle of minimality, so it provides only essential abstractions for isolation, execution, and communication. The kernel also implements access control through a capability system that requires explicit capability delegation for access to any kernel functionality or other system services, enforcing the principle of least authority [25]. The kernel implements an object-based design, where functionality is encapsulated in kernel objects, which must be accessed through capabilities. Kernel objects are created through *factories*, special kernel objects with this ability. The kernel supports spatial isolation through the use of protection domains called

tasks. Each task combines an address space and a capability space. The address space uses hardware mechanisms, such as a memory management unit or a memory protection unit, to control memory access permissions and handle virtual page mappings. The capability space records the set of capabilities available to the task, determining which kernel objects it can access. The kernel also provides *threads* as a fundamental unit of execution within the system. Each thread must be bound to a task. For communication between threads, the kernel implements inter-process communication (IPC) through a synchronous message-passing mechanism. In addition to IPC, the kernel supports asynchronous notifications through *interrupt requests (IRQs)*, which can be triggered by both hardware events and software signals. The runtime environment, on the other hand, is a modular software layer that runs in the unprivileged userspace and provides useful services like memory management and application loading. Following the microkernel design philosophy [29], L4Re pushes drivers and resource management into userspace components.

Thread Scheduling: L4Re performs local scheduling with a separate ready-queue on each CPU core. For multi-core settings, it provides a mechanism to migrate threads between different cores. The scheduling regime is implemented in the microkernel to achieve fast response times and deterministic performance, as competitive, real-time capable userspace implementation is hard [15], [19]. The kernel separates the state of a thread into scheduling state and execution state. The scheduling state consists of all parameters (priority, reservations, deadlines, etc.) necessary to make scheduling decisions about this thread, and is recorded in a special kernel-internal *scheduling context* object. Scheduling contexts only support CPU time partitioning, and their parameterization can be changed at compile time to allow modular scheduling strategies to be plugged into the scheduler. By default, L4Re implements a static priority round-robin strategy, but using scheduling contexts, an implementation of dynamic priority EDF can be easily added as well. In the past, scheduling contexts have been used to flatten scheduler hierarchies in virtualized settings by exporting relevant information directly to the host scheduler [26]. However, for the purpose of this paper, these objects are static in nature, limited to CPU time partitioning, and transparent to userspace control.

Timeslice Donation and Helping: To reduce the number of costly scheduler invocations, L4Re employs the concept of timeslice donation [28]. If requested during IPC, the kernel will not change the scheduling context in the context switch from sender to receiver thread. This facilitates highly optimized communication in which server threads can execute requests on the CPU time allocations of their clients.

In addition, the L4Re microkernel employs the concept of helping as a system-wide cross-core priority inheritance mechanism to mitigate priority inversion scenarios [20], [21]. Instead of blocking and waiting on a lock, threads can register themselves as helpers and donate their CPU time by helping the current lock owner in the critical section [40]. It is important to note, that for this mechanism to work correctly, the following invariant must hold: *A lock holder must always be ready to execute while it owns a helping lock*.

Exception Handling: When a thread raises an exception, the microkernel reflects it to userspace by sending exception IPC to a preassigned handler, accompanied by the execution state of the faulted thread. Different handlers can be registered per exception type such as page faults or hardware faults (e.g., division by zero). This keeps the kernel policy free and allows for tailored strategies such as demand paging in userspace.

Component Proxies: Another key concept of L4Re are component proxies. These objects implement an interface, such as that of a kernel object, in userspace and act as a proxy, interposing function calls and filtering parameters from invocations. They route accepted invocations to a lower component in a hierarchy all the way down to the real kernel interface, allowing for arbitrary chaining of proxies and filtering rules. This helps to build hierarchical subsystems effortlessly.

III. Design

A. Requirements

We now present a model for the universal management of shared resources in real-time OSes such as L4Re. The model should satisfy the following requirements:

- **Capability Control.** Capabilities are an established mechanism for fine-granular access control [13]. By utilizing capabilities for managing access to shared resources, the model should inherit the established semantics and flexibility of modern access control systems, like the one in L4Re. This integration also allows precise and secure control over resource access both from userspace as well as in the kernel.
- **Policy Freedom.** Given the diversity of shared resources, the model should not enforce a particular resource management strategy. Instead, it should provide the flexibility to implement and nest various management techniques, accommodating hierarchical and domain-specific policies.
- **Performance.** The model should not compromise the efficiency of existing resource management techniques. More importantly, it should add only minimal overhead to essential kernel functionality like IPC, context switching, and the creation, migration and deletion of threads.
- **Temporal Isolation.** The model must enable the construction of systems with temporal isolation between subsystems. Specifically, it should ensure that any interference caused by contention for shared resources is contained within subsystem boundaries and does not impact the performance or timing behavior of other subsystems.

B. Concepts

We introduce the *scheduling constraint (SC)* as the fundamental kernel object for expressing conditions that determine whether a thread may execute. An SC encodes a single boolean value, its *state*, and exposes the following set of operations:

- **`check()`:** Return (i.e., read) the current state of the SC.
- **`flip()`:** Toggle (i.e., write) the SC state. May be handled asynchronously or synchronously (see Section III-G).

- **Attachment / Detachment:** SCs may be attached to and detached from threads (many-to-many relation). Performed via kernel scheduler interfaces (`attach()`, `detach()`) and guarded by capabilities.
- **Scheduling Exceptions:** SCs may raise an exception when violated. Handled by a configurable handler thread.

To integrate SCs into a real-time kernel, we add two small phases to kernel operation: an *SC-switch phase* performed on context switches, and an *SC-check phase* performed during scheduling decisions. In the rest of this section, we describe these key concepts in greater detail and illustrate their interaction with conventional real-time OS design.

C. Formalization

SCs can be attached to threads. This is a many-to-many relation from the set of all constraints to the set of all threads, so it is possible to attach multiple SCs to one thread, and one SC to multiple threads. For a given thread, we refer to the set of attached SCs as the *scheduling context (SCX)* of that thread. The state of the SCX is the logical conjunction of the states of all member SCs. Semantically, a thread can run if its SCX state evaluates to true, i.e. if the conditions of all attached SCs are satisfied. Note that this retains the L4Re idea of a scheduling context object that accumulates the scheduling information for a given thread. We formalize these concepts in the following way:

- Set of scheduling constraints: $SC = \{sc_1, sc_2, sc_3, \dots\}$
- Set of threads: $T = \{t_1, t_2, t_3, \dots\}$
- SC state function: $s : SC \to \{\text{true}, \text{false}\}$
- Attachment relation: $A \subseteq SC \times T$
  where $(sc_i, t_j) \in A$ means scheduling constraint sc_i is attached to thread t_j.
- SCX definition: $k : T \to \mathcal{P}(SC)$
  where $\forall\ t \in T :\ k(t) = \{\ sc \mid (sc, t) \in A\}$.
  For readability, we denote $k(t)$ by scx_t.
- Runnable function: $r : T \to \{\text{true}, \text{false}\}$
  where
$$\forall\ t \in T :\ r(t) = \bigwedge_{sc\ \in\ scx_t} s(sc)$$

With the introduction of SCs, we install a flexible interface between threads and resource servers. Figure 1 gives an overview of how SCs integrate into the existing design of a real-time OS. Our model provides a clear separation of concerns, decoupling workload specification, resource partitioning and resource assignment into distinct functions that interface through SCs. Because of this distinction, each function remains independently auditable without requiring further knowledge of the rest of the system.

D. Resource Servers

The SC abstraction is very powerful as the state of an SC can describe arbitrary boolean conditions. For example, we can use it to express the availability of a resource reservation (slice, budget, etc.) of CPU time, cache capacity, memory bandwidth

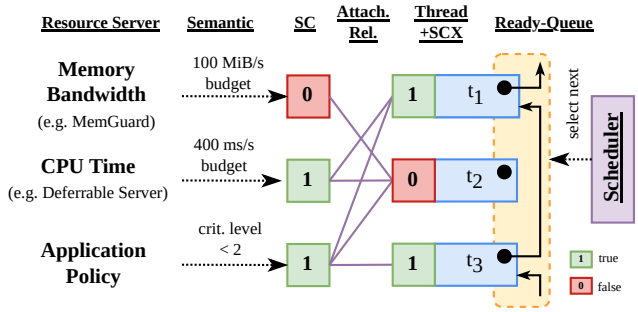

Fig. 1. System overview shows how scheduling constraints serve as an indirection between threads and resource servers.

or power. In addition, the SC state can also encode higher-level constraints such as mutual exclusion or the affiliation to a certain criticality level. We comment on mutual exclusion and the eventual consistency of SC-based scheduling decisions across multiple cores below in Subsection III-G. As long as a condition can be meaningfully expressed in a boolean variable, SCs can encode it. In general, SCs provide a policy free abstraction and it is up to the actor controlling the SC state to decide the semantics of the constraint being enforced.

Hence, SCs offer a universal interface to resource servers. In our model, resource servers do not need to know which threads are relying on the resources that they manage. Similarly, they also do not need to handle the intricate dependencies that arise from changes in resource state, e.g., modifying thread states or the ready-queue whenever a time slice is depleted or a budget replenished. Instead, they can use a set of SCs to express the state of the individual resource reservations that they hand out. While monitoring the underlying resource and accounting for its consumption, they flip the SC state accordingly and, by definition of the SCX, automatically affect the runnable state of all threads attached to the SC. This ensures quick and deterministic resource management independent of how many threads are affected. Since SCs are regular kernel objects and can be controlled through capabilities, we support resource servers both in userspace as well as in the kernel. In addition, it is important to note that SC capabilities, which embody the ability to control the access to a shared resource, can be delegated and revoked at any time, just like any other capability in L4Re.

Accounting for Consumption: Allowing SCs to represent resource reservations has some interesting implications. For example, a thread executing with SCs attached means that it is actively consuming the resources they describe. To account for this consumption, a resource server must be notified whenever an attached thread is activated, deactivated, migrated or terminated. The resource server can then use these hints to adapt its accounting infrastructure, e.g., by programming CPU timers, performance counters or other special-purpose extensions of the instruction set architecture (ISA) accordingly.

We enable this functionality by introducing an additional *SC-switch phase* to each context switch. In this phase, the kernel iterates the SCXs of the outgoing and incoming threads,

notifying the associated resource servers to *deactivate* the old SCs and *activate* the new ones. Note that the overhead induced by this phase scales linearly with SCX size, i.e. the amount of constraints attached to both threads. Therefore, we ensure an upper bound through a configuration parameter that globally limits the number of constraints that can be attached per thread. In addition, the overhead depends on the actual setup/teardown work performed by each resource server. In practice, we found that for hardware resources, this is usually very small and deterministic, e.g., just writing to one register to reprogram the CPU timer or some performance counter.

E. Resource Assignment

Another important consequence of SCs encoding resource reservations is how resources are assigned in the system. In our model, instead of maintaining fixed scheduling parameters per thread, the kernel only needs to keep track of which SCs are attached to each thread. The concrete implementation of the resource management technique and necessary parameters can remain hidden in the resource server, as shown in Figure 1. Therefore, controlling resource assignment necessitates the ability to attach or detach individual SCs.

We decide that the scheduler is responsible for offering mechanisms allowing to manipulate the attachment relation of SCs to threads. Therefore, we extend its interface with two appropriate methods: `attach()` and `detach()`. They take as arguments a capability to a thread and a capability to an SC, and if the capabilities are valid, both methods update the kernel-internal data structure that tracks the attachment relation. We opt for this way, because it has the advantage of interacting very well with L4Re's existing concept of component proxies. A component proxy that implements the new scheduler interface can now enforce application-specific resource assignment policies in userspace, i.e. limit what kind of constraints can be attached or cannot be detached, while the kernel-level mechanism remains simple and policy free. In addition, preventing nonsensical configurations is a domain-specific task that can already be done in userspace, further sparing the kernel from unnecessary policy and entry traps. It is important to note that L4Re's scheduler interface is also used to instruct the kernel-level scheduler to start, stop and migrate a thread. Therefore, a scheduler proxy can easily enforce that a certain set of SCs is attached to each thread that passes through it. This enables the *principle of least authority*, where by default, each thread only has minimal resource access.

This mode of resource assignment complements the design decision for conjunction-like semantics with SCs. No matter how many SCs are attached to a thread, the most restricting set of constraints always wins. Hence, we enable the creation of nested and hierarchical subsystems with a monotonically increasing refinement of available resources. We also see the usefulness of a model with disjunction-like semantics. However, early experiments with such a design revealed significant overheads for resource accounting and thread scheduling. Therefore, we define SCs as conjuncts for a simpler system design and less complexity in basic kernel functions. In general, since SCs can be attached to multiple threads simultaneously, our model supports a variable granularity of resource assignments, ranging from individual threads to CPU cores up to entire subsystems.

Failure Tolerance: We expose no mechanism to userspace that allows to query for all resource assignments of an individual thread. Instead, the necessary delegation of SC capabilities to the correct actors must be already considered when provisioning the system. However, we handle the lifetime of SC kernel objects in a special way. Normally, the L4Re microkernel would start destroying and garbage collecting a kernel object as soon as it is no longer referenced by a valid capability, e.g., when a capability holder fails and terminates. This is problematic for SCs as they can be still attached to live threads in the system. Therefore, we use reference counting to record the number of kernel-internal references left, i.e. how many threads are attached to the SC. If there are none, the destruction protocol proceeds regularly. If, on the other hand, references do exist, SCs enter a *quiescent state* in which they remain as kernel objects and still enforce their condition. They only transition from this state to destruction when all attached threads have been terminated or detached. In general, the handling of quiescent SCs is an application-specific policy that is closely related to the strategy for failure tolerance. In the current design, we simply preserve the execution environment, represented by the SCs, at the time of failure. For resilient systems that recover automatically, the control of quiescent SCs can be handed over again to restarted components.

Exception Handling: When threads violate the conditions dictated by attached SCs, e.g., through overrunning a budget or exceeding a bandwidth, this can be seen as a contract violation, and therefore as an illegal execution state. In this case, it might be required to not only prevent execution of the faulting thread, but also to propagate this fact in the system in order to enact application-specific policy. Hence, we make the kernel handle these synchronous, potentially recoverable faults as exceptions and reflect them to userspace. This is conceptually quite similar to *preemption IPC* in earlier works [43]. It should be possible to handle scheduling exceptions independently from regular ones, therefore, each thread must keep an additional reference to a scheduling exception handler. This handler is the envisaged mechanism to implement additional logic such as admission control, failure recovery or a criticality switch.

F. Thread Scheduling

Thread scheduling is an orthogonal concept to SCs and our model does not limit the range of available scheduling strategies. As can be seen in Figure 1, SCs are transparent to the thread selection logic and only help to determine the set of runnable threads in the system. From this set, which is recorded in the ready-queue, the scheduler then selects the next thread to run according to a specific strategy. Therefore, without loss of generality, we retain L4Re's default total ordering of threads using static priorities and a round-robin motion in each priority band.

We also considered subsuming thread priorities into the SC mechanism, but ultimately decided not to. We considered a system design, in which at every point in time there exists only one runnable thread whose SCX state evaluates to true. Alternatively, the thread priority could be derived from the set of attached SCs by summing up a priority share offered by each SC. Or, following a more traditional scheme, the priority could be derived from the internal state of attached SCs, e.g., the distance to a deadline. A detailed exploration of these designs remains out of scope for this paper. Instead, we chose a design, where SCs can add extra thread-blocking reasons to the system without altering the existing scheduling logic. If the system is dimensioned correctly, these blocking reasons can act as additional guardrails for isolation, safety, and security.

Distinguishing Different Blocking Reasons: Traditionally, the state of a thread is described by the well-known *ready, running, blocked* state machine. A thread transitions to the blocked state whenever it is hindered in execution such as waiting for a resource or an event. However, with the introduction of SCs, it is helpful to distinguish between these two different reasons for thread blocking.

On the one hand, a thread may block due to an internal reason, such as waiting for an IPC event. In this case, there is a logical barrier in the program flow that prevents the thread from continuing execution, independent of how many resources are available. In the following, we refer to this state as *internal blocking*. On the other hand, a thread may block due to an external reason such as a resource not being available indicated by an SC not being satisfied. In this case, the program flow is very much clear and intrinsically, the program is able to continue. However some constraint of the execution environment, e.g., insufficient memory bandwidth, can prevent execution, typically to satisfy performance guarantees for other applications. In the following, we refer to this state as *external blocking*. The crucial difference is that externally blocked threads can continue execution if the resource environment changes while internally blocked threads can't. With this knowledge, we can safely extend the helping and donation concepts in L4Re to a thread's entire SCX. In these cases, the kernel will retain the SCs of the caller/helper thread during the context switch. This ensures that the receiver not only profits from donated CPU time, but also from the entire, potentially less restrictive SCX of the caller/helper thread. In addition, the invariant of the helping lock remains intact, as a lock holder that is externally blocked can always continue execution with a different valid SCX. We depict a simplified version of L4Re's thread state machine in Figure 2. We observe that individual threads transition between the ready, running and externally blocked states quite often, virtually every time the state of an attached SC flips. Therefore, we need to find an efficient way to record all runnable threads in the system to speed up scheduling decisions.

Optimizing Thread Selection: The scheduler interfaces with the ready-queue to select one thread to run next. However, frequent changes in SC state might continuously update the set of runnable threads in the queue. Therefore, we must find

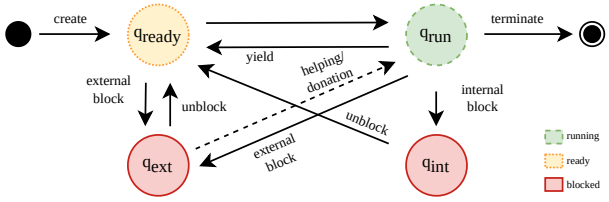

Fig. 2. Thread state diagram with distinction for internal/external blocking

a compromise between accuracy and update overhead of the ready-queue. To achieve reasonable performance, we propose a design with lazy evaluation of SCX state. Internally blocked threads are immediately added to and removed from the ready-queue as before. Externally blocked threads, on the other hand, are eagerly added but removed only on demand.

We realize this design by introducing an additional *SC-check phase* to each scheduling decision. After the scheduler identifies the highest-priority thread in the queue, the kernel examines the candidate's SCX and checks the state of each SC attached. If the SCX state evaluates to true, the context switch is performed. Otherwise, the context switch is aborted, and the scheduler must select another candidate. This check adds very little overhead, as it only reads one boolean value per SC and is also upper bounded by the maximum number of SCs per thread. Crucially, a thread that fails this check remains externally blocked until the offending SC becomes satisfied again. So instead of repeatedly removing and re-inserting such threads from the ready-queue, we remove the thread when it first fails this check and register it with the faulting SC. When that SC eventually flips to be satisfied, it wakes and requeues all registered threads. This optimization is based on our observation that, when SCs are shared, many externally blocked threads in the ready-queue are never actually attempted for execution before they become ready again. By deferring removal until a thread is actually about to run, we avoid the overhead of updating the ready-queue. In the worst case the scheduler may scan the entire queue and remove all externally blocked threads before finding a runnable one. However, this cost is usually smaller, or at worst, equal to the alternative of immediately removing them every time an attached SC changes.

G. Synchronization

As SCs can be attached to multiple threads simultaneously, modification of the SC state might influence threads across different CPU cores. This poses an interesting challenge to synchronization in multi-core settings.

The anticipated access pattern for SCs is characterized by frequent reads and infrequent writes. In this design, core-local schedulers act as multiple readers and the owning resource server acts as a writer. For example, the state of an SC is read on every context switch involving it, but the boolean variable is only modified if the underlying condition changes. Similarly, a thread's SCX is also iterated on every context switch, while it is only modified when an SC is attached or detached.

To optimize for this pattern, we keep reads fast and unsynchronized. There are three operations on the write side that modify SCs: flipping SC state, manipulating the attachment relation, and registering a handler thread for a faulting SC. These operations correctly serialize on a per-SC lock. To honor any such changes to the SCs on the read side, we must initiate a scheduling decision on each CPU core that hosts attached threads. If only the local CPU core is affected, inconsistencies are impossible, so we simply invoke the scheduler when leaving the kernel. If a remote CPU core is affected, however, we send an inter-processor interrupt (IPI), that triggers a scheduling decision on the remote core upon receipt. This IPI is sent asynchronously, so the local core continues execution immediately. As soon as the remote core receives the IPI, it performs the next scheduling decision, and at at this point becomes aware of any SC state changes.

A key observation here is that information propagation between multiple cores cannot occur in zero time, since it relies on messages being sent between the cores. For resources like memory bandwidth, which programs consume ambiently, a resource depletion that is detected on one core will take non-zero time until it can be acted upon on another core. Therefore, a small overshoot of resource consumption is inevitable in a multi-core setup. However, this overshoot is bounded by the amount of resource that can be consumed within the in-flight time of the IPI. Once the IPI is received by the target core, the scheduling decision will detect the flipped SC and deschedule the thread whose reservation is depleted.

For resources that are not consumed ambiently, but rather explicitly, for example buffer space in a software queue, mutual exclusion behavior may be desired. To express such resources with SCs, they must allow the writer of the SC boolean state to know, when the read-side is aware of the change. More precisely, when flipping such a resource from available to blocked, the writer of the SC bit must be able to learn, at what point the IPIs have been delivered to all other cores and a scheduling decision there has acted on this change. At that point, the writer can reason that all other threads that previously may have used the resource have now been descheduled. It can then continue to use the resource itself.

Such a flow requires one important change: The IPI after writing the SC bit cannot be sent asynchronously, but we must instead synchronously wait for its delivery and the subsequent scheduling decision. This is similar to how atomic CPU instructions are implemented, except that inter-core messaging and stalling of the calling thread happens internally in the CPU by way of cache coherence messages rather than IPIs. Our implementation supports this mutual-exclusion-compatible behavior. It can be configured individually per SC, whether writes to this SC bit should be handled asynchronously or synchronously.

IV. Implementation in L4Re

We base our implementation on the publicly available snapshot of L4Re (version `l4re-base-23.04.02` [2]). In addition, we publish our patches as open-source software with their adoption being under active discussion upstream.

First, we extend the L4Re microkernel to support the concept of SCs. This entails the implementation of a new kernel object class *Sched_constraint* and the associated changes to thread state, context switching and the ready-queue as described in Section III. In addition, we modify the scheduler to manipulate the kernel-internal representation of the attachment relation. Regarding this relation, we experiment with different data structures for the recording of attachment information. We recognise that it is sufficient to record attachment in one direction only, i.e. the SCs attached to a given thread, as this data is needed frequently, e.g., at each scheduling decision. The other direction, which indicates the threads attached to a given SC, is needed less often and is partially covered by the reference count in the SC. Furthermore, we also realize that the choice of data structure has far-reaching consequences for the expressiveness of the SC-based scheduling regime. For example, when using a linked list where a thread only stores the head pointer to a chain of SCs, these chains coalesce when SCs are shared by multiple threads. As a result, the model can only describe scenarios with transitive resource sharing. It remains an interesting incentive for future work to identify the affinity of individual data structures to the requirements of different deployment scenarios. For this work, we choose a static size array per thread, where the size is dictated by the maximum number of SCs per thread, because it strikes a good balance between expressiveness, memory demands and efficiency for enumerating and looking up elements. Next, we enable SC capabilities in the L4 runtime environment. We also enable scheduler proxies for the enforcement of resource assignment policies in userspace and we add the infrastructure to handle scheduling exceptions reflected by the kernel.

To complete the prototype, we implement a diverse collection of resource servers that utilize SCs for resource partitioning and performance isolation. We implement one server that encodes fixed-size timeslices of CPU time with SCs and another that encodes budgets of CPU time following the deferrable servers algorithm [44]. Both servers leverage the hardware timer of the CPU to program interrupts that signal the depletion of a timeslice or the replenishment of a budget. When handling these interrupts, both servers set the state of their SCs accordingly. In addition, we introduce a server inspired by MemGuard [49] that uses SCs to encode memory bandwidth partitioning. This server uses the performance counters of the CPU to count misses and write-backs in the last-level cache. The counter is periodically programmed to a value that overflows when a pre-configured bandwidth is exceeded, and the resulting interrupts help the server to set the SC state correctly.

We also add a server that takes advantage of Arm MPAM [5] for memory bandwidth and cache capacity partitioning in hardware. Unfortunately, at the time of writing, the commercial availability of MPAM-enabled hardware is extremely limited, as it requires a compatible CPU and memory subsystem. This also applies to the MPAM support in common

simulators and emulators (QEMU, Arm FVP, etc.). For the time being, we can only work with the technical specification and leave the functional evaluation of this server pending. On MPAM-capable hardware, the server would program the MPAM registers, instructing the hardware to enforce specific memory bandwidth or cache capacity budgets. The MPAM hardware would generate interrupts when a budget is depleted, prompting the server to update the SC state. On our prototype, the required register writes take place, but are directed to regular memory that is not backed by an actual MPAM device.

MPAM support for cache capacity partitioning is interesting and requires additional explanation. Applications are distinguished by so-called *PARTIDs*. These are identifiers that MPAM uses to configure resource partitioning and that resource clients (i.e., cores accessing a cache) use to tag their requests. MPAM allows static way partitioning of caches, which results in limits, which portions of the cache (i.e., which ways of each set) a PARTID is allowed to use. Such a static partitioning is entirely enforced by MPAM in hardware and does not require enforcement by way of descheduling applications. Thus, our SC mechanisms are not needed in such a configuration. However, MPAM also supports more dynamic resource partitions for caches. Thresholds can be set that allow deliberate over-allocation, where the sum of all partitions are larger than the cache. Whenever these configurable capacity thresholds are reached, the MPAM hardware triggers an interrupt. Similarly to DRAM bandwidth, it is now the job of the operating system to govern cache allocation by throttling heavy-use cache clients.

Any resource server using our mechanism to manage a resource can make such policy decision for its resource. It communicates this decision by flipping the corresponding SC bit to false. The kernel applies the flip locally and, if threads attached to that SC run on remote cores, issues IPIs so remote schedulers re-evaluate their SC-checks. Threads whose SCX now evaluates to false become externally blocked and are registered with the faulting SC until the server flips it true again. When the server re-enables the reservation, the kernel wakes and requeues registered threads, using IPIs as needed for remote wake-up.

To allow for a fair comparison to the baseline, we implement the servers for hardware resources directly in the kernel. This has two benefits. First, the resource servers themselves are not constrained by any SCs, simplifying system provisioning significantly. Second, they can manipulate SC state while interrupts are disabled, ruling out potential deadlocks caused by the server blocking while its own CPU time or memory bandwidth budgets are replenished. It is important to note that deadlocks are still possible for resource servers in userspace and misconfigurations of static thread priorities. In this sense, our model offers the same liveness insurances as L4Re's previous scheduling regime.

Beyond their conception in L4Re, SCs are designed as a portable mechanism. This means that no part of the SC abstraction fundamentally relies on microkernel-style primitives. Support for reference-counted kernel objects, context switch and scheduler callbacks, and custom IPI handlers is commonly available in many real-time systems. In addition, capability control is not strictly necessary. In our prototype, capabilities add a fine-granular and highly secure access control layer and open up the possibility for resource servers in userspace. However, SCs can also just be a fully transparent, internal mechanism for a real-time kernel to model the interaction of threads and shared resource reservations. Therefore, we expect the SC mechanism to integrate effortlessly with the architecture of other real-time OSes such as FreeRTOS or the real-time Linux project. The underlying principle of SCs, abstracting resource state into a set of boolean flags, provides a platform-agnostic interface that decouples resource management from scheduling details. Thanks to their expressiveness, SCs allow multiple, resource-specific techniques to be combined, nested, or applied selectively without incurring significant design complexity or runtime overhead. This not only streamlines the integration of various resource partitioning strategies but also lays a solid foundation to enable mixed-criticality co-location.

TABLE I
EVALUATION PLATFORM HARDWARE SPECIFICATION

Platform	Raspberry Pi 4 Model B (Broadcom BCM2711)
CPU	4x Cortex-A72 (ARMv8-A) 64-bit @ 1.5GHz
Cache	L1: (per-core) D: 32KB, I: 48KB; L2: (shared) 1MB
Memory	2GB LPDDR4 SDRAM

V. EVALUATION

We evaluate our implementation to quantify the overhead of SCs and to determine our model's ability to establish temporal isolation for individual subsystems. We conduct the evaluation on a Raspberry Pi 4 Model B. Detailed specification of the evaluation hardware can be found in Table I.

A. Performance Comparison

To measure SC overhead, we compare our model's performance against the implementation baseline (L4Re snapshot version `l4re-base-23.04.02`). We conduct two microbenchmarks to investigate the performance of individual IPC and context switch operations, which are the most critical ones for theoretical kernel performance. Afterwards, we perform a set of macrobenchmarks to evaluate the impact of our changes under realistic workload scenarios.

IPC: In this scenario, a client sends an IPC message to a server, and the server instantly replies back. It represents the fastest IPC roundtrip through the microkernel and enables all possible optimizations such as SCX donation. We stepwise increase the number of individual SCs attached to client and server thread to examine their effect on IPC performance. For both threads, the first SC represents a CPU timeslice, while all subsequent constraints are always satisfied and have no enable/disable logic. The benchmark is conducted across all possible sender-receiver placements, covering both local and cross address spaces and CPU cores (lAS-lCPU, xAS-lCPU, lAS-xCPU, xAS-xCPU). We measure the overall IPC

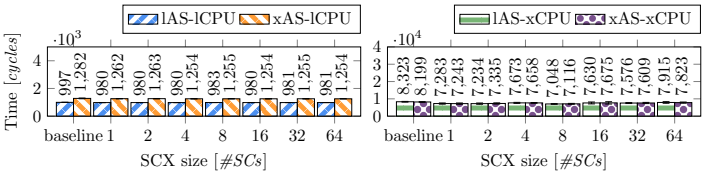

Fig. 3. Average IPC roundtrip time over number of attached SCs for various possible sender-receiver placements. IPC performance is agnostic to SCX size.

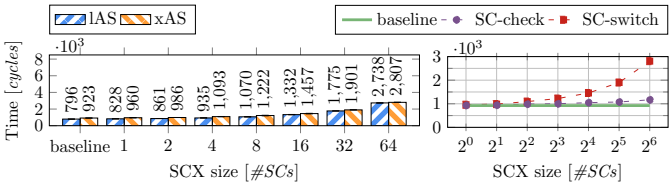

Fig. 4. Left: Average context switch time over number of attached SCs for various thread placements; Right: Detailed composition of context switch overhead for xAS setting. The context switch performance degrades linearly with SCX size. The SC-switch phase accounts for most of the overhead.

roundtrip time in CPU cycles. To account for cache and pipelining effects, results are averaged over 100k iterations.

Figure 3 shows the average results of 100 runs for each flavor. IPC performance remains constant at around 981 cycles for local address space, 1255 cycles for cross address space, and 7486 cycles for cross CPU core [1]. Fluctuations are irregular and well within the standard deviation. The diagram also depicts slightly faster IPC times in the modified kernel.

In our model, a thread's SCX only aggregates the scheduling information instead of storing it directly. Therefore, the runnable state of a thread must be computed in the SC-check phase, which entails additional overhead for thread scheduling. Our lazy implementation of the ready queue avoids this whenever possible, which saves a few cycles in the optimal case, earning the modified microkernel slightly better IPC performance in this scenario. Furthermore, this benchmark shows that the efforts to keep SC overhead out of the critical IPC path are successful. We conclude that the modified microkernel experiences no overhead in IPC performance compared to baseline and IPC performance is agnostic to SCX size. In general, communication is not affected by the amount of constraints that are attached to sender or receiver thread.

Context Switch: In the next scenario, we measure the time that it takes to switch from one thread to another. This depicts the fastest way to switch between subsystems, involving preemption, interrupt handling, a scheduling decision and the continuation of the selected thread. We stepwise increase the number of individual SCs attached to both threads to examine their effect on context switch performance. For both threads, the first SC represents a CPU timeslice, while all subsequent constraints are always satisfied and have no enable/disable logic. The benchmark is conducted in two flavors, where both threads reside either in the same address space (lAS) or in two different ones (xAS). We measure the overall context switch time in CPU cycles. To account for cache and pipelining effects, we use the fastest result over 1k iterations.

Figure 4 shows the average results of 100 runs for each flavor. We see that context switches with one SC attached are on average 32 cycles slower for lAS and 37 cycles slower for xAS compared to the baseline. Furthermore, the context switch time increases linearly with the number of SCs up to 2738 cycles for lAS and 2807 cycles for xAS at an SCX size of 64. Figure 4 also depicts the xAS overhead's detailed composition in SC-check and SC-switch phase. Notably, the latter is responsible for the majority of the overhead observed.

Even in the optimal case where both threads are ready, the scheduler must perform at least one SC-check and SC-switch phase during a scheduling decision. For the smallest configuration with one SC per thread, this emerges as a minimal overhead compared to the baseline. In addition, it can be seen that each additional SC adds a constant amount of overhead to the context switch time, because the runtime of both phases scales linearly with the number of SCs attached. However, Figure 4 highlights an important aspect of our design. The SC-check phase is highly optimized, only checking a single boolean value per SC in the SCX. This is significant because the scheduler checks the runnable state of a thread more often than it performs the actual switch to it. The SC-switch phase, which entails comparatively costly setup/teardown work, dominates the overhead observed.

Applications: To assess the system as a whole, we evaluate our implementation under four realistic application workloads. The goal is to answer the question whether round-robin scheduling with one CPU timeslice SC per thread causes any noticeable overhead compared to the timeslice implementation in the baseline. We measure the runtime of all workloads in CPU cycles. For the *pi* workload, a single thread executes a spigot algorithm to calculate the first 15k decimal digits of π as fast as possible. It represents a simple long-running computation that does not interact with the kernel, but is periodically interrupted by the timeslice logic. The *vm-linux* workload is more involved and starts L4Re's virtual machine monitor to boot into an unmodified `linux-v6.1-rc1` guest. VMs periodically interact with the microkernel for VM-related events. This workload is interesting as it represents a typical use case for L4Re. The *prodcons* workload embodies a producer consumer scenario that processes 1 million items of 64 bytes in a shared buffer. The buffer size is limited to one item and the signalling between producer and consumer is done through a semaphore. The *omp* workload uses OpenMP directives to spawn 512 threads and distributes them evenly across the four available CPU cores. The thread routine itself is insignificant and fast, putting a heavy load on the thread creation, migration and context switch mechanisms in L4Re.

Table II lists the average runtime and standard deviation over 100 runs of all workloads normalized to their execution time in the baseline framework. The *pi* workload is stable and

[1]We used a development version of the L4Re microkernel to evaluate the relative performance impact of our modifications. The latest performance numbers can be found on the official L4Re website.

TABLE II
RUNTIME OF REALISTIC WORKLOADS NORMALIZED TO BASELINE

App	Description (Stress Factor)	t	σ
pi	CPU Computation (Throughput)	1.000039	0
vm-linux	Hypervisor (VM entry/exit)	1.000800	0.001973
prodcons	Producer-Consumer (Semaphore)	1.030476	0.012969
omp	Parallel Program (Thread Spawn)	1.053397	0.007818

exhibits barely noticeable overhead. The *vm-linux* workload shows a small overhead of 0.08%, however the measurements demonstrate a high variance. For the *prodcons* workload, an average runtime overhead of 3.05% is observed. The *omp* workload shows a noticeable overhead of 5.34%.

The *pi* workload is single-threaded. So at the end of each time slice, the scheduler checks for higher-priority threads and verifies if the *pi* thread is still eligible through the SC-check phase, adding minimal overhead. In the *vm-linux* workload, the virtual machine monitor attaches SCs to each newly spawned vCPU thread. This imposes the small overhead that we observed. However, in this experiment, the standard deviation is bigger than the absolute difference in measurements between baseline and modified L4Re, which means that the overhead might not be statistically significant. For the *prodcons* workload, the semaphore used for signaling prevents SCX donation when unblocking a waiting producer or consumer thread, making this workload suffer from context switch overhead. Similarly, the *omp* workload uses POSIX threads, which rely on a scheduler proxy to automatically attach SCs to spawned threads. Together with context switch overhead, this explains the additional CPU cycles observed.

We conclude that regular application workloads that generate only modest interaction with the kernel exhibit nearly no runtime overhead (0-0.1%) under round-robin timeslice scheduling with SCs. In addition, even applications that interact with the kernel excessively only experience a marginal overhead of less than 6%. Most importantly, the *vm-linux* example demonstrates that SC-based resource assignment even extends to the vCPUs of virtual machines running on L4Re.

Multi-core Consistency: Finally, we examine the overhead of propagating SC changes from userspace across multiple threads on different CPU cores. For this experiment, we measure the time taken to flip the state of an SC via a system call. We distinguish between two modes based on the SC state's write behaviour. In the asynchronous case, the initiating CPU core continues immediately. In the synchronous case, however, the kernel blocks the initiating thread on this core until all the affected remote cores have received the IPI and made a scheduling decision. Only then is the initiating thread allowed to continue. Therefore, this experiment also quantifies the overhead incurred when using SCs to implement mutual exclusion-style access to a resource. We stepwise increase the number of cores affected by the flip and measure the execution time of the system call in CPU cycles. The results are averaged over 1k measurements.

Figure 5 shows the average results for each mode and CPU

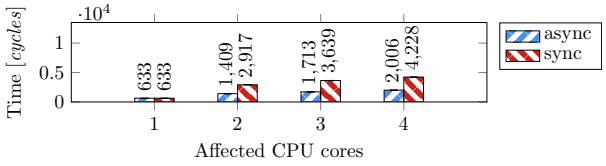

Fig. 5. Time taken to flip SC state over number of CPU cores being affected. The overhead increases linearly with the number of cores. For multi-core settings, mutex-style flips (sync) incur roughly twice the overhead compared to asynchronous ones.

set over 100 runs. There is a significant slowdown when switching from the single-core to the multi-core setting, due to the additional requirement of sending an IPI. In the multi-core setting, the overhead increases linearly with the number of affected CPU cores. We conclude that propagation of SC changes is performant, for example, it is generally faster than an IPC roundtrip. Additionally, the synchronous mode guarantees mutual exclusion-style access to the underlying resource, incurring roughly twice the overhead compared to the asynchronous case. In the asynchronous mode, the initiating thread continues without guarantee that the IPIs have been delivered to all other cores and a scheduling decision there has acted on the new SC state. Therefore, simultaneous resource consumption and interference are possible in this mode, but they are bounded by the in-flight time of the IPI.

B. Temporal Isolation

By using SCs to guard the access to CPU time and memory bandwidth, we conduct two benchmarks to see how well our model establishes temporal isolation for individual subsystems. For these experiments, a subsystem is one L4Re task.

CPU Time: We construct a popular scenario for a mixed-criticality system. A high criticality, low time-sensitivity task (*crit*) performs important work and is irregularly interrupted by a low criticality, high time-sensitivity task (*qual*). *Crit* executes the *pi* workload from earlier and must stay above a critical performance threshold of extracting at least 5k digits per second. These regular deadlines are jeopardized by *qual*, which has a higher priority that allows it to instantly access the CPU whenever it becomes ready. In the unconstrained case, *qual* might monopolize the CPU, making *crit* miss its deadlines. To establish temporal isolation, we attach to *qual* an SC that represents a CPU time budget. Configured to a period of 10ms, we stepwise increase this budget from 0ms to 10ms in 1ms increments. To quantify the interference experienced by *crit*, we measure its throughput in digits per second.

Figure 6 (left) depicts the average throughput of *crit* over 100 benchmark runs at each budget setting for *qual*. The standard deviation (<1%) is omitted. We see that the throughput of *crit* decreases linearly with increasing CPU utilization of *qual*. In the undisturbed case, *crit* achieves a throughput of at least 12950 digits per second. Notably, for up to 60% CPU utilization of *qual*, *crit* can still meet its deadlines.

In this scenario, *crit* runs in the slack time left over by *qual*. Since the slack time is inversely proportional to the CPU

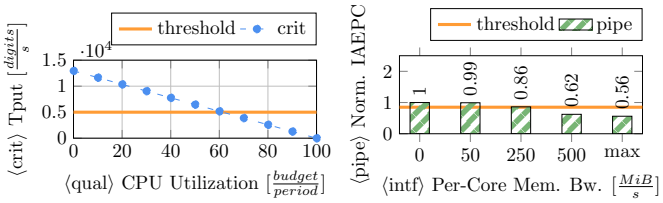

Fig. 6. Left: Throughput of the *crit* task decreases linearly as CPU utilization of the interfering *qual* task increases. Restricting *qual* with a CPU time SC achieves temporal isolation; Right: Throughput of the *pipe* task is impaired as memory bandwidth usage of the *intf* task increases. Restricting *intf* with a memory bandwidth SC achieves temporal isolation.

utilization of *qual*, and since the spigot algorithm executed by *crit* extracts digits at a constant rate, the throughput scales linearly with available CPU time. By calibrating the budget setting accordingly, the performance of *crit* can stay above the critical threshold. We conclude that the implementation of the deferrable server algorithm is effective in bounding CPU time interference of a low-criticality task on a high criticality task. Therefore, it is compatible with the SC interface.

Memory Bandwidth: This benchmark pairs a memory-sensitive task (*pipe*) with a memory-intensive task (*intf*). *Pipe* is a sequential video processing pipeline using `libav` libraries from the FFmpeg project [1]. It uses a single thread on a single CPU core to read raw image data from memory and apply a selection of filters. It must stay above a critical threshold of processing at least 30 frames per second. To ensure consistency, we attach to *pipe* an SC that represents a memory bandwidth budget of 400 MiB/s. *Intf* performs pointer chasing on the remaining three CPU cores, stressing the memory subsystem and generating contention at the DRAM controller. To establish temporal isolation, we constrain *intf*'s available memory bandwidth on each CPU core with another SC and conduct the benchmark with varying budgets. To quantify the interference experienced by the pipeline, we measure its instructions architecturally executed per cycle (IAEPC).

Figure 6 (right) shows the average IAEPC of the pipeline over 100 benchmark runs for each budget normalized to running in isolation. The standard deviation (<0.5%) is omitted. The throughput of *pipe* decreases with increasing memory bandwidth available to *intf*. At maximum interference, the pipeline is almost twice as slow, executing merely 56% of the instructions compared to isolation in the same time frame. For up to 250 MiB/s bandwidth per-core available to *intf*, the pipeline can still meet its deadline. Notably, for up to 50 MiB/s interference per-core, pipeline throughput is nearly unaffected.

Higher interference from *intf* leads to higher contention at the DRAM controller and also higher latency for instructions that access memory. Thus, the throughput of *pipe* degrades with increasing interference by co-runners in the system. However, when the DRAM controller's random access servicing rate is not exceeded, interleaving memory accesses from different subsystems do not interfere and they can achieve comparable performance to running in isolation, as shown with a budget of 50 MiB/s in Figure 6 (right). We conclude that the implementation of memory bandwidth partitioning is effective in mitigating interference on memory-sensitive tasks. Therefore, it is compatible with the SC interface.

C. Discussion

In summary, the evaluation demonstrates that the adoption of the SC primitive incurs minimal overheads for a state-of-the-art real-time OS. Furthermore, we also confirmed that the new scheduling regime conforms to the other design requirements, offering SCs as a universal, capability-controlled, and policy-free mechanism that can encode a great variety of resource partitioning and timing verification techniques. Our implementation adds about 3150 lines (0.5%) to the code base of the L4Re snapshot, augmenting both kernel and userspace runtime services.

VI. Related Work

The SC-based scheduling model presented in this paper explores an OS design for temporal isolation and timing verification on multi-core platforms. Existing work in this domain, that also accounts for interference through contention for shared resources, can be grouped into two main categories.

A. Combined Frameworks

On the one hand, there are frameworks that combine management techniques for CPU time, caches and interconnects to achieve higher degrees of freedom from interference. For example, MARACAS [48] extends the Quest real-time OS with multi-core scheduling and load-balancing capabilities. After primarily ensuring CPU timing constraints, the system throttles threads in excess CPU cycles to minimize memory latency and contention for shared caches and the memory bus. However, the admission policy is static, implemented in the kernel, and oblivious to userspace control, making it unsuitable for microkernel systems such as L4Re. Our model, by contrast, uses capability-controlled SCs to provide a uniform interface to resource servers both in userspace as well as in the kernel. In addition, SCs do not enforce any order of importance on execution constraints or underlying resources. The multi-resource server model [22] isolates tasks from CPU time and memory bandwidth interference by decrementing counters on each request. SCs are not limited to this very specific approach and can describe arbitrary system resources. The Single Core Equivalence technology package [34] combines kernel-internal techniques for cache management, DRAM bandwidth regulation and bank partitioning to derive isolated per-core schedulability results. While the system implements isolation at the granularity of CPU cores, our model pushes the boundary to subsystems and individual applications. Since SCs are attached to single threads and can be shared across different CPU cores, our model has the flexibility to depict a wider range of scheduling configurations. Recently, RT-Gang++ [8] has been proposed as a solution to an industrial challenge posed by Arm [4]. The system is tailored to a specific platform and workload, combining partitioned real-time gang scheduling with bandwidth regulation for the last-level

cache and integrated GPU. In contrast, the generic nature of our model allows SCs to be driven by the majority of available resource management techniques for timing verification [33], facilitating the ability to scale to other workloads and other platforms more easily.

Finally, Arm and Intel have also introduced support for resource partitioning in hardware [5], [11]. While these extensions are not a panacea for shared resource contention [41], [51], they can serve as an efficient foundation for isolation [37]. The scope of these specifications ends at the hardware boundary, leveraging interrupts to signal changes in resource state to software. As SCs target exactly this use case, our model provides the perfect OS counterpart to easily adopt these hardware-based partitioning approaches.

B. Universal Interfaces

On the other hand, there are efforts to design interfaces that standardise access to shared system resources. MC-IPC [9] is a synchronous IPC protocol that encapsulates a resource in a single-threaded resource server and enforces transactional access semantics without relying on trust assumptions or cooperation between clients. This protocol targets only resources that are subject to mutual exclusion at the software level. In contrast, SCs can represent arbitrary resources, both in hardware and in software. However, the generic nature and focus on performance of our model comes at the cost of fewer guarantees compared to specialized interfaces like MC-IPC.

Other systems also implement similar interfaces to handle contention for shared system resources. S3K [24], a RISC-V partitioning kernel, enforces temporal isolation for CPU time with a constant-time, kernel-resident scheduler and custom hardware instructions [46] to mitigate side-channels. In contrast, the SC primitive presented in this work provides capability control from userspace or kernel space over arbitrary resources across CPU cores. In seL4, scheduling contexts [32] allow capability control over CPU time. This abstraction implements budgets in the kernel according to a sporadic servers algorithm and mitigates contention for CPU time in mixed-criticality settings. It is static and limited to the CPU time resource, so it is not universal. By contrast, SCs do not enforce such a policy. Rather than controlling scheduling contexts with fixed parameterization, our model provides capability access to SCs, and flexible control over which SCs constitute an individual SCX. In Composite, temporal isolation and user-level scheduling is implemented with TCaps [15]. This abstraction provides capability access to individual slices of CPU time that can be delegated in the system. Each subsystem must have access to a set of TCaps, and thread execution is always associated with a TCap whose underlying CPU time is being consumed. In addition, TCaps must be explicitly replenished by repeatedly delegating more timeslices, and they cannot be revoked unless they are empty. The mechanism is static, also limited to CPU time, and cannot cross CPU core boundaries. In contrast, SCs can be driven by multi-core resource management techniques for arbitrary resources. Further, in accordance with L4Re's capability semantics, delegation and attachment can be revoked at any time.

The L4Re microkernel was the first L4 system to implement scheduling contexts as a universal interface to configure thread scheduling and access to CPU time [26]. Subsequent work has confirmed the applicability of this mechanism in mixed-criticality settings [45]. However, as described in Section II, the implementation is static, limited to CPU time and compile-time parameterization. The SC interface presented in this work can be viewed as an extension to this concept. For future work, we plan to enhance our design with a refinement operation to derive resource subsets in hierarchical subsystems. We also plan to formally analyze and improve the performance of controlling SCs from userspace.

VII. Conclusion

We have presented a model that uses *scheduling constraints (SCs)* as a main operating system primitive to bind the runnable state of threads to arbitrary execution conditions. This abstraction is very powerful, providing a flexible interface between threads and resource servers to simplify the accessible state of shared system resources. The SC-based scheduling model achieves a clear separation of concerns, where resource state can be managed independently of complex thread dependencies, while thread scheduling remains efficient and free of resource management overhead. We implemented SCs in the L4Re real-time framework and evaluated our implementation for SC overhead and the ability to establish temporal isolation on a modern multi-core platform. Our evaluation showed that the IPC performance of the L4Re microkernel is unaffected by SCs, and that realistic application workloads experience minimal runtime overhead compared to unmodified L4Re. More importantly, the evaluation showed that SCs can be driven by a range of different resource servers to implement timing verification techniques and establish temporal isolation for individual subsystems. With these results, we claim that SCs are a universal OS mechanism for managing the diverse landscape of shared resources in real-time systems.

VIII. Acknowledgements

We thank the anonymous reviewers and our shepherd for their valuable feedback and suggestions. This work was co-funded by the European Union and by budget adopted by the Saxon State Parliament under project MikroRZ.

References

[1] FFmpeg. https://ffmpeg.org/, 2025. [Online; accessed 2025-11-11].

[2] L4Re Operating System Framework. https://l4re.org/, 2026. [Online; accessed 2026-02-09].

[3] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. An empirical survey-based study into industry practice in real-time systems. In *41st IEEE Real-Time Systems Symposium, RTSS 2020, Houston, TX, USA, December 1-4, 2020*, pages 3–11. IEEE, 2020.

[4] Matteo Andreozzi, Giacomo Gabrielli, Balaji Venu, and Giacomo Travaglini. Industrial challenge 2022: A high-performance real-time case study on arm. In *34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy*, volume 231 of *LIPIcs*, pages 1:1–1:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[5] ARM. *Arm Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification*, issue a.a edition, 2024.

[6] Michael Garrett Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 357–367. IEEE, 2019.

[7] Michael Garrett Bechtel and Heechul Yun. Cache bank-aware denial-of-service attacks on multicore ARM processors. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023*, pages 198–208. IEEE, 2023.

[8] Michael Garrett Bechtel and Heechul Yun. Analysis and mitigation of shared resource contention on heterogeneous multicore: An industrial case study. *IEEE Trans. Computers*, 73(7):1753–1766, 2024.

[9] Björn B. Brandenburg. A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 196–206. IEEE Computer Society, 2014.

[10] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multicore platforms. *ACM Trans. Embed. Comput. Syst.*, 13(4s):124:1–124:29, 2014.

[11] Intel Corporation. *Intel® Resource Director Technology (Intel® RDT) Architecture Specification*, revision 1.0 edition, 2023.

[12] Robert I. Davis, David Griffin, and Iain Bate. Schedulability analysis for multi-core systems accounting for resource stress and sensitivity. In *33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference*, volume 196 of *LIPIcs*, pages 7:1–7:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[13] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, 1966.

[14] Johannes Freitag, Sascha Uhrig, and Theo Ungerer. Virtual timing isolation for mixed-criticality systems. In *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 13:1–13:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[15] Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. Temporal capabilities: Access control for time. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 56–67. IEEE Computer Society, 2017.

[16] Georgia Giannopoulou, Pengcheng Huang, Rehan Ahmed, Davide B. Bartolini, and Lothar Thiele. Isolation scheduling on multicores: model and scheduling approaches. *Real Time Syst.*, 53(4):614–667, 2017.

[17] Robert Gifford, Abby Eisenklam, Georgiy A. Bondar, Yifan Cai, Tushar Sial, Linh Thi Xuan Phan, and Abhishek Halder. CORD: co-design of resource allocation and deadline decomposition with generative profiling. *CoRR*, abs/2501.08484, 2025.

[18] Ana Guasque, José María Aceituno, Patricia Balbastre, José E. Simó, and Alfons Crespo. Schedulability analysis of dynamic priority real-time systems with contention. *J. Supercomput.*, 78(12):14703–14725, 2022.

[19] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Trans. Comput. Syst.*, 34(1):1:1–1:29, 2016.

[20] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, pages 217–230. USENIX, 2001.

[21] Michael Hohmuth and Michael Peter. Helping in a multiprocessor environment. In *Proceeding of the Second Workshop on Common Microkernel System Platforms*, 2001.

[22] Rafia Inam, Nesredin Mahmud, Moris Behnam, Thomas Nolte, and Mikael Sjödin. The multi-resource server for predictable execution on multi-core platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2014.

[23] Michael Jacobs, Sebastian Hahn, and Sebastian Hack. A framework for the derivation of WCET analyses for multi-core processors. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 141–151. IEEE Computer Society, 2016.

[24] Henrik Karlsson and Roberto Guanciale. Partitioning kernel with capability controlled temporal and spatial partitioning. In *2025 IEEE Real-Time Systems Symposium (RTSS)*, pages 68–81, 2025.

[25] Adam Lackorzynski and Alexander Warg. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, IIES '09, Nuremburg, Germany, March 31, 2009*, pages 25–30. ACM, 2009.

[26] Adam Lackorzynski, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening Hierarchical Scheduling. In *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 93–102. ACM, 2012.

[27] Ao Li, Jinwen Wang, Sanjoy K. Baruah, Bruno Sinopoli, and Ning Zhang. An empirical study of performance interference: Timing violation patterns and impacts. In *30th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2024, Hong Kong, May 13-16, 2024*, pages 320–333. IEEE, 2024.

[28] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 175–188. ACM, 1993.

[29] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 237–250. ACM, 1995.

[30] Jane W.-S. Liu. *Real-time systems*. Prentice Hall, 2000.

[31] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang.*, 4(POPL):20:1–20:31, 2020.

[32] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 26:1–26:16. ACM, 2018.

[33] Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019.

[34] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. Wcet(m) estimation in multi-core systems using single core equivalence. In *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*, pages 174–183. IEEE Computer Society, 2015.

[35] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. Automated generation of time-predictable executables on multicore. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS 2018, Chasseneuil-du-Poitou, France, October 10-12, 2018*, pages 104–113. ACM, 2018.

[36] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 269–279. IEEE Computer Society, 2011.

[37] Ashutosh Pradhan, Daniele Ottaviano, Alexander Zuepke, Andrea Bastoni, and Marco Caccamo. Work-in-progress: A first practical look at arm's mpam for real-time systems. In *2025 IEEE Real-Time Systems Symposium (RTSS)*, pages 592–595, 2025.

[38] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Multimedia Computing and Networking 1998*, volume 3310, pages 150–164. SPIE, 1997.

[39] Simon Reder and Jürgen Becker. Interference-aware memory allocation for real-time multi-core systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*, pages 148–159. IEEE, 2020.

[40] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[41] Parul Sohal, Michael Garrett Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A closer look at intel resource director technology (RDT). In *RTNS 2022: The 30th International Conference on Real-Time Networks and Systems, Paris, France, June 7 - 8, 2022*, pages 127–139. ACM, 2022.

[42] Brinkley Sprunt, Lui Sha, and John P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real Time Syst.*, 1(1):27–60, 1989.

[43] Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *ACM SIGOPS Oper. Syst. Rev.*, 41(4):59–68, 2007.

[44] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Computers*, 44(1):73–91, 1995.

[45] Marcus Völp, Adam Lackorzynski, and Hermann Härtig. On the expressiveness of fixed-priority scheduling contexts for mixed-criticality scheduling. *Proc. WMC, RTSS*, pages 13–18, 2013.

[46] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. Microarchitectural timing channels and their prevention on an open-source 64-bit risc-v core. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 627–632, 2021.

[47] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2008, April 22-24, 2008, St. Louis, Missouri, USA*, pages 80–89. IEEE Computer Society, 2008.

[48] Ying Ye, Richard West, Jingyi Zhang, and Zhuoqun Cheng. MARACAS: A real-time multicore VCPU scheduling framework. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*, pages 179–190. IEEE Computer Society, 2016.

[49] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 55–64. IEEE Computer Society, 2013.

[50] Yuting Zhang and Richard West. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS) 2006, 5-8 December 2006, Rio de Janeiro, Brazil*, pages 191–201. IEEE Computer Society, 2006.

[51] Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing Arm's MPAM From the Perspective of Time Predictability. *IEEE Trans. Computers*, 72(1):168–182, 2023.