Priority-Based Concurrency and Shared Resource Access Mechanisms for Nested Intercomponent Requests in CAmkES

Marion Sudvarg, Zhuoran Sun, Ao Li, Chris Gill and Ning Zhang

Department of Computer Science and Engineering, Washington University in St. Louis, 1 Brookings Drive, St. Louis, 63130, MO, USA.

*Corresponding author(s). E-mail(s): msudvarg@wustl.edu; Contributing authors: zhuoran.sun@wustl.edu; ao@wustl.edu; cdgill@wustl.edu; zhang.ning@wustl.edu;

Abstract

Component-based design encapsulates and isolates state and the operations on it, but timing semantics cross-cut these boundaries when a real-time task's control flow spans multiple components. Under priority-based scheduling, inter-component control flow should be coupled with priority information, so that task execution can be prioritized appropriately end-to-end. However, the CAmkES component architecture for the seL4 microkernel does not adequately support priority propagation across intercomponent requests: component interfaces are bound to threads that execute at fixed priorities provided at compile-time in the component specification. In this paper, we present a new library for CAmkES with a thread model that supports (1) multiple concurrent requests to the same component endpoint; (2) propagation and enforcement of priority metadata, such that those requests are appropriately prioritized; (3) implementations of Non-Preemptive Critical Sections, the Immediate Priority Ceiling Protocol, and the Priority Inheritance Protocol for components encapsulating critical sections of exclusive access to a shared resource; and (4) extensions of these mechanisms to support nested lock acquisition. We measure overheads and blocking times for these new features, use existing theory to discuss schedulability analysis, and present a new hyperbolic

bound for rate-monotonic scheduling of tasks with blocking times that allows tasks to be assigned non-unique priorities. Evaluations on both Intel x86 and ARM platforms demonstrate that our library allows CAmkES to provide suitable end-to-end timing for real-time systems.

Keywords: real-time systems, component middleware, priority protocols

1 Introduction

As the complexity of software systems has increased, component-based software engineering has emerged as a key approach for providing structure, modularity, and reusability in system design [1]. Components encapsulate state, computation, and communication, allowing for (1) separation of functional concerns and (2) isolation of resource utilization within components to ensure timing and other para-functional properties, while allowing (3) sophisticated behaviors to be realized, and (4) desired properties to be enforced locally and end-to-end, through composition and coordination of multiple components. To achieve all those benefits at once, component frameworks tailored for real-time and embedded systems, ranging from the Component-Integrated ACE ORB (CIAO) [2, 3] specialization of the CORBA Component Model (CCM) [4] standard, to the Component Architecture for microkernel-based Embedded Systems (CAmkES) [5] framework, extend traditional component models to also consider attributes (e.g. priorities and execution times) and constraints (e.g. deadlines) for timing and other para-functional properties.

In particular, CAmkES, which targets the seL4 microkernel [6], provides a description language for the functional requirements of a component-based embedded system, and for static assignment of para-functional attributes such as priorities to component threads. Such static assignment, however, may be problematic in systems where real-time task execution crosses component boundaries. Under priority-driven scheduling, tasks are assigned priorities to ensure their deadlines are met. Tasks and components may be orthogonal; a task may be decomposed into execution across multiple components, and a single component may execute on behalf of multiple tasks, but by assigning priorities to *components* rather than to *tasks*, CAmkES does not fully support priority-driven scheduling of multi-component tasks.

To address this limitation, in [7] we presented a new library to enable priority-aware inter-component requests in CAmkES running atop seL4. The library provides a concurrency framework that allows multiple concurrent tasks to execute across shared components, while retaining end-to-end task prioritization.[1] It supports (1) multiple concurrent requests to the same component procedural interface endpoint; (2) priority propagation, which couples requests with priority metadata and ensures that each component thread is prioritized according to the task for which it executes; and (3) implementations of

[1] Available from https://www.sudvarg.com/priority-aware-camkes

Non-Preemptive Critical Sections, Immediate Priority Ceiling Protocol, and Priority Inheritance Protocol, for components encapsulating exclusive access to a shared resource. The concurrency framework includes new extensions to the CAmkES specification language, allowing users to easily specify the desired real-time behavior of a component. It is implemented entirely in userspace, so it can take advantage of existing formally verified kernel mechanisms in seL4.

In this paper, we extend our prior work, introducing mechanisms to support nested lock acquisition via intercomponent requests. The mechanisms our library provides are designed to be both fast and predictable in execution time. Our protocols use priority semantics to guarantee consistency over lock acquisition without additional atomic operations. We measure the overhead induced by our protocols, and validate that it is appropriately bounded. We also provide an overview of how to do schedulability analysis for a component-based task system specified with our extensions to CAmkES, taking into account blocking times induced by both library overhead and shared resource access under our supported protocols. New to this extension, we present (and prove in Appendix A) a formulation of the hyperbolic bound for rate-monotonic scheduling of tasks with blocking times, which allows tasks to be assigned non-unique priorities. We also demonstrate, through empirical timing measurements of task sets running on both Intel x86 and ARM hardware platforms, that our implementation, coupled with this analysis, is successful in meeting end-to-end deadlines for cross-component task execution in real-time systems.

The rest of this paper is organized as follows. Section 2 gives relevant background about the seL4 microkernel and the CAmkES framework, and provides an overview of related work. Section 3 describes our target task model and existing approaches for performing response-time and schedulability analysis under this model. Section 4 details the design and implementation of our library, and Section 5 provides a brief summary of its usability, integration with CAmkES, and auxiliary tools. Section 6 presents (1) measurements on two different hardware platforms of the overheads introduced by the protocols supported by our library, (2) empirical evaluations on those same platforms (in which no deadline misses were observed) of synthetic task sets with harmonic periods using those protocols, and (3) schedulability analyses which incorporate our measurements of protocol overheads across a broader set of synthetic task sets; all three of which demonstrate the suitability of our library for real-time systems. Finally, Section 7 concludes the paper, and discusses directions for future work.

2 Background and Related Work

CAmkES provides a description language for the functionality of a component-based embedded system. It is designed to incur minimal execution time and memory overhead. While addressing these para-functional requirements for overall system design, explicit real-time specifications for individual tasks and components were not a part of the original model. CAmkES has since

been extended atop **seL4** [6], allowing for specification of components' thread priorities statically at compile time [8]. The seL4 microkernel is a widely used, lightweight OS kernel [6, 9] with capability-based access control to broker all user-level functionality, and its functional specification and implementation have been formally verified [10, 11]. Additionally, all kernel pathway worst-case execution times have been analyzed and bounded [12]. This makes seL4 well-suited for real-time systems, and it is a natural target for CAmkES, allowing for separation between components, while providing efficient IPC channels to handle the explicitly-defined connections between them. In this work, we extend our contributions presented in [7], providing a framework that expands CAmkES' support for real-time task sets executing end-to-end across shared components atop the seL4 kernel. We also show how real-time tasks can be mapped to a component model and implemented in CAmkES and seL4 *without changes to seL4's verified codebase or existing CAmkES software*, thus allowing easy adoption.

The Component-Integrated ACE ORB (**CIAO**) [2, 3] extends and specializes the CORBA Component Model [4] with component QoS specifications provided as additional metadata, separate from functional specifications. In both CAmkES and CIAO, RPC invocations are realized as synchronous IPC between threads in separate components, though if components are specified to exist within the same protection domain, both CAmkES and CIAO can resolve RPCs between them into direct function calls.

The **Patina** API [13] provides priority-aware synchronization primitives for shared resource access in seL4. It includes a mutex service that provides an implementation of the Priority Inheritance Protocol; threads obtaining a lock must invoke the service via an RPC. Similarly to our approach described in Section 4.2, it enables priority-ordered locking, circumventing seL4's native FIFO wait-queue. It also keeps track of which thread holds the mutex so that it can elevate its priority if a higher-priority thread requests the lock. In contrast, however, our framework extends the existing CAmkES design to encapsulate all execution over a shared resource in its own component. This allows access to a shared resource to be defined at the component level, and each component manages its own priority-based locking protocols with the common framework. This avoids the need to interpose a separate mutex server component. Additionally, Patina does not support nested locking; our framework provides mechanisms to support nested locking over multiple priority-based shared resource access protocols, including the Priority Inheritance Protocol.

The **AUTOSAR** specification [14, 15] also supports priority-based access to shared resources, assigning a ceiling priority to each mutually-exclusive shared resource, then elevating a task's priority to that ceiling when it obtains the resource, thus implementing the Immediate Priority Ceiling Protocol [16]. However, it does not specify native Priority Inheritance Protocol support, nor does it offer native priority propagation with RPC calls, both of which are features of our implementation.

An alternative to inter-thread RPC is **thread migration** between protection domains, which some OS kernels enable by decoupling a thread's execution context (e.g. register values, stack, address space, etc.) from its scheduling context (e.g. priority, resource accounting statistics, temporal reservations, etc.). In the **Mach 3.0** kernel, RPC is realized by having the requesting thread immediately continue executing in the context of the server; a partial context switch is needed to separate execution contexts, but the scheduling context maintains continuity across the call [17]. A similar, efficient thread migration mechanism was later realized for inter-component requests in the **Composite** component-based OS [18]. These approaches let end-to-end task execution retain scheduling semantics across component boundaries, but do not directly support priority protocols for shared resource access. A migrating scheduling context must also acquire an execution context and related resources (e.g. a stack) from the target component's scope. It is argued [19] that access to the allocated stacks in a component can induce priority inversion, unless each component allocates a stack for each thread in the system. Because CAmkES explicitly defines all intercomponent request paths, our framework is able to allocate threads (and associated stacks) in a way that avoids such contention.

Capacity-reserve donation (**Credo**) [20], implemented in the original L4 microkernel [21], uses scheduling context migration to propagate priorities with intercomponent requests, while also supporting shared resource access protocols (in particular, the Priority Inheritance Protocol [22] and the Immediate Priority Ceiling Protocol [23, 24]). A similar approach [25] was later implemented to support the Priority Inheritance Protocol and bandwidth inheritance [26] in the **NOVA** microhypervisor [27]. These approaches, unlike ours, require the kernel to track the full migration path of the scheduling context. In contrast, ours is a userspace framework that allows coordinated control over the implemented protocols. Request messages are coupled with a priority parameter and a unique identifier for the originating task, allowing each component to manage and track its incoming and outgoing requests. Runtime traversal of request chains for nested priority inheritance is managed via message passing among cooperative components. This allows RPC to be realized as synchronous IPC using a thread model that enables immediate request-passing where appropriate, while appropriately blocking on access to locked resources.

3 System Model

3.1 Task and Component Model

In this work, we target an implicit-deadline, sporadic task system, using fixed-priority, preemptive scheduling on a uniprocessor. Our system is composed of a set Γ of n tasks $\{\tau_i = (C_i, T_i, p_i)\}$ characterized by a worst-case execution time C_i and a minimum interarrival time T_i, and assigned a priority p_i. We assume, for schedulability considerations, that task execution is nonblocking

(except when waiting for a lock held by another task). In other words, jobs do not self-suspend except on completion.

Our target OS platform is the seL4 microkernel [6], which supports fixed-priority preemptive scheduling. The seL4 kernel, compiled with default settings, schedules threads of the same priority in round-robin fashion; in this work we instead consider a version in which the round-robin timeslice is set large enough that threads will always run to completion unless preempted by a strictly higher priority thread. In Sections 4 and 6 respectively, we describe our implementation and evaluation of this version.

We define a mapping from our task system Γ to *originating components* and sets of *component procedure interfaces* (CPIs), described in CAmkES, as follows. First, for each task $\tau_i \in \Gamma$, we define a component c_i that we say *originates* the task. In CAmkES, that component is specified as *active* (using the `control` directive), giving it an associated thread to run the task. The thread is assigned (via a CAmkES attribute) the priority p_i of the task. Common functionality or resources, shared among multiple tasks, may be encapsulated behind CPIs within other components.[2] Each such task τ_i is thus decomposed into multiple subtasks: an initial subtask executing in its originating component c_i, with control flow then passing out of it to zero or more shared CPIs, then returning back through the request chain before finally completing execution in c_i, as illustrated in Fig. 1. Requests can be nested: CPIs may send requests to other shared CPIs. Further, a request chain may branch: a CPI or the component originating a task may make subsequent requests to multiple other CPIs, or even multiple requests to the same CPI, within the control flow of a single job.

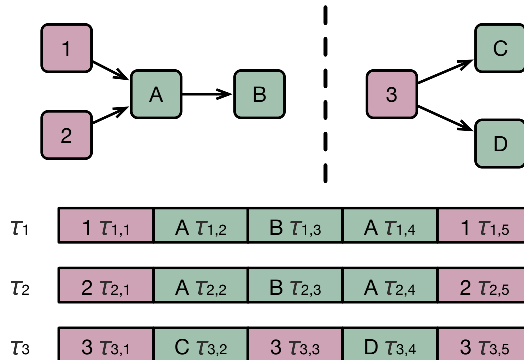


Fig. 1: Tasks $\{\tau_1, \tau_2, \tau_3\}$ originate in active components $\{1, 2, 3\}$ respectively. Components 1 and 2 share common functionality, realized through a request to the same CPI in component A which itself sends a request to a CPI in B. Component 3 sends a request to a CPI in C, then to one in D. This defines a decomposition of each task into subtasks.

[2] Our system model does not allow an originating component to specify any CPIs since it is by definition the root of all request chains emanating from it.

Components hosting one or more shared CPIs are realized in CAmkES by defining them as *passive* (lacking the `control` directive). Explicit connections — from an originating component or CPI that uses it, to the CPI — must be defined in CAmkES. Connections are backed by an underlying *endpoint*, an seL4 kernel object that enables RPC calls between threads through synchronous IPC, where the requesting thread blocks until it receives a reply.[3] Endpoints, being synchronous, require a sending and receiving thread to rendezvous. Thus, task execution will be blocked at the transition between subtasks if no threads in the target CPI are waiting on the endpoint.

CAmkES components, using the built-in connector types, establish CPIs as endpoints with a single listening thread that handles all requests; its priority is specified as an attribute of the CAmkES configuration. This presents fundamental incompatibilities with our task model: multiple tasks executing end-to-end across shared CPIs are not guaranteed to execute subtasks according to the priority of the task, and may be blocked from progress if a procedure on its request path is already executing, even if that execution is for a request from a task of lower priority. The framework we provide addresses these problems, providing appropriate priority propagation across CPIs that encapsulate shared functionality and mechanisms for additional resource access protocols for CPIs encapsulating exclusive access to shared resources.

3.2 Resource Access Protocols and Schedulability

To analyze schedulability of our end-to-end task model, we consider several possibilities under the system model we have described. For each of them, we describe how the existing theory for rate-monotonic scheduling, including blocking time analysis for shared resource access protocols, applies to our model.

3.2.1 Priority Propagation

In Section 4.3 we describe our framework's mechanisms to support priority propagation — whereby threads belonging to CPIs encapsulating shared functionality execute at the priority of the requesting task, and can preempt execution to handle requests of higher priority — allowing the component execution model to match our priority-based task model presented above. However, overhead induced by the protocol introduces a brief blocking time during the transition, as if the task holds a lock according to the Immediate Priority Ceiling Protocol (IPCP). Under IPCP, a task may be blocked, at most, for the duration of a single critical section [23]. This allows us to compute blocking times, and hence perform schedulability analysis, for a task system (mapped to originating components and CPIs as we describe next) using priority propagation.

[3] Another version of this is also possible, where a request is sent asynchronously to an *event* interface, in which case no reply is necessary, and control need not return to the requesting active component or CPI. We defer consideration of this alternative to future work.

We say that task τ_i originates in active component c_i and additionally executes across a set of CPIs (hosted by passive components) $\hat{c}_i$ (of size $\|\hat{c}_i\|$). The worst-case overhead for sending a request with a propagated priority to a CPI is denoted $C_{\text{prop_send}}$, and for replying is $C_{\text{prop_reply}}$, with worst case total overhead $C_{\text{prop}} = C_{\text{prop_send}} + C_{\text{prop_reply}}$ (measured in Section 6). For a CPI c, we denote the worst-case procedure execution time as $C(c)$. Thus, a task τ_i has total WCET:

$$C_i = C(c_i) + \sum_{c \in \hat{c}_i} C(c) + \|\hat{c}_i\| \cdot C_{\text{prop}} \tag{1}$$

Each CPI c has a minimum priority $p_{\min}(c)$ among tasks for which it executes, and a maximum priority $p_{\max}(c)$. The blocking time B_i induced on a task τ_i is therefore $\max(C_{\text{prop_send}}, C_{\text{prop_reply}})$ if there exists a CPI c for which $p_{\min}(c) < p_i$ and $p_i \leq p_{\max}(c)$; otherwise, the CPI experiences no blocking time. Schedulability analysis of task sets where each task has a unique priority then can be performed using the Hyperbolic Bound with blocking factors [28]:

$$\forall \tau_i \in \Gamma \qquad \prod_{\tau_j : p_j > p_i} \left(\frac{C_j}{T_j} + 1 \right) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2 \tag{2}$$

A more pessimistic bound, though one that applies to a task system Γ ($|\Gamma| = n$) where multiple tasks may have the same priority, is presented in [22] (Corollary 17), as a generalization of the rate-monotonic utilization bound in [29]:

$$\sum_{\tau_i} \frac{C_i}{T_i} + \max_{\tau_i} \left\{ \frac{B_i}{T_i} \right\} \leq n \left(2^{1/n} - 1 \right) \tag{3}$$

In theory, for RM schedulability analysis of task sets where some tasks have equal periods, these tasks can be assigned unique priorities in some arbitrary order. In practice, however, tasks of equal periods are often assigned equal priorities. Systems are typically limited to a fixed number of priority levels (one version of our implementation is limited to 128 priorities, as described in Section 4.2); the pigeon-hole principle dictates that for large enough task sets, some tasks cannot have unique priorities. Further, assigning unique priorities to tasks of equal periods involves a decision about the ordering which may have undesirable implications (e.g., a task might be preempted by another task with the same period, resulting in an undesirable increase in context switching). To address these issues, we provide the following schedulability condition (which we prove in Appendix A) for task sets with non-unique priorities and blocking times:

$$\forall \tau_i \in \Gamma \qquad \prod_{\tau_j : p_j \geq p_i, j \neq i} \left(\frac{C_j}{T_j} + 1 \right) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2 \tag{4}$$

3.2.2 Immediate Priority Ceiling Protocol

Our framework allows a CPI to encapsulate execution of a critical section with the Immediate Priority Ceiling Protocol (IPCP). Such CPIs have a worst-case request overhead time $C_{\text{fix}} = C_{\text{fix_send}} + C_{\text{fix_reply}}$. We introduce a new term, $B(c)$, for the worst-case blocking time that a CPI c can induce. For a CPI c to which priorities are propagated, $B(c) = \max(C_{\text{prop_send}}, C_{\text{prop_reply}})$ as before. For a CPI c having a fixed priority, e.g. one using IPCP, blocking time must be computed recursively as the sum of its execution time and protocol overhead $(C_{\text{fix}} + C(c))$, plus the execution times and protocol overheads for all CPIs to which it makes requests. Now, the blocking time B_i induced on task τ_i is the maximum worst-case blocking time induced by any CPI:

$$B_i = \max_c \{B(c) \mid p_{\text{min}}(c) < p_i \leq p_{\text{max}}(c)\} \tag{5}$$

IPCP is an improved version of Non-Preemptive Critical Sections (NPCS), which assigns the maximum system priority to execution in all critical sections. Under NPCS, then, the blocking time induced by any CPI becomes:

$$B_i = \max_c \{B(c) \mid p_{\text{min}}(c) < p_i\} \tag{6}$$

Task WCETs must account for the different overheads, C_p and C_f, induced by requests to CPIs that propagate priorities and have fixed priorities, respectively. We say that a task τ_i executes across a set of fixed-priority CPIs $\hat{c}_{i,f}$ and a set of CPIs that propagate priority $\hat{c}_{i,p}$. This results in a new equation for task WCET, slightly modified from Eqn. 1:

$$C_i = C(c_i) + \sum_{c \in \hat{c}_i} C(c) + \|\hat{c}_{i,p}\| \cdot C_{\text{prop}} + \|\hat{c}_{i,f}\| \cdot C_{\text{fix}} \tag{7}$$

Schedulability analysis, using Eqns. 3 or 4, can be performed using these new blocking times and WCETs.

3.2.3 Priority Inheritance Protocol

Our framework also supports CPIs that use the Priority Inheritance Protocol (PIP), as described in Section 4.2. As we show in Section 6.1, our mechanism induces protocol overhead that depends on whether the lock is already acquired, and if so, on the number of tasks that execute on the CPI. For such a CPI c, we denote this $C_i(c)$.

Because a task can be blocked for the duration of multiple critical sections under PIP, CPIs implementing PIP may induce longer worst-case blocking times than those using IPCP [22]. However, under PIP, higher-priority tasks may preempt lock-holders in situations where this preemption could not happen under IPCP, which may make PIP attractive, especially for some soft real-time applications.

In [7], which this paper extends, our implementation restricted the nesting of CPIs such that a CPI implementing PIP could only send requests to CPIs with a fixed priority ceiling, i.e., IPCP or NPCS. In this paper, we extend the mechanisms to allow for nested priority inheritance: if a thread in a CPI implementing PIP is blocked on a nested request, and it inherits a higher priority from a new request to its CPI, it will propagate the inherited priority to the CPI handling the nested request. As described in Section 4.2, this propagation requires a sequence of updates across the request chain. Each update induces an overhead of C_{up}, for a total of $l(c){\cdot}C_{\text{up}}$, where $l(c)$ is the length of the longest request chain rooted at CPI c. Thus, the worst-case overhead $C_i(c)$ can be computed as:

$$C_i(c) = \max\left\{C_i^{\text{unlocked}}\ ,\ C_i^{\text{locked}} + C_i^{\text{locked}}(c) + l(c){\cdot}C_{\text{up}}\right\} \tag{8}$$

Here, C_i^{unlocked} denotes the overhead of the protocol when the lock is available, C_i^{locked} denotes the overhead when the lock is acquired and no additional tasks are waiting on the lock, and $C_i^{\text{locked}}(c)$ denotes the additional worst-case overhead induced by the CPI's Notification Manager priority queue (described in Section 4.2) when full.

4 Design and Implementation

The CAmkES framework [8] provides a specification language to describe a system as a collection of components and connections between them. CAmkES generates the necessary seL4 system calls to create components and IPC described by a user-provided system specification and component source code, then compiles everything into an Executable and Linkable Format (ELF) binary packaged with an seL4 kernel image.

Our goal in this work, as in our prior work [7] that it extends, is to elaborate on the CAmkES framework without changes to its underlying parser or to the seL4 kernel. The design and implementation of our approach provides **priority propagation** across thread-safe, reentrant components executing similarly to sequential, non-componentized versions. We also support several priority-based locking protocols — including the **Immediate Priority Ceiling Protocol** (IPCP), **Non-Preemptive Critical Sections** (NPCS), and **Priority Inheritance Protocol** (PIP) — to provide synchronization over component-encapsulated shared state. Component execution is replicated across subtasks and control flows, with multiple subtasks in a single component, so that functionality itself need not be replicated. Each component provides spatial isolation via its own separate address space, which is shared among its threads' unique stacks.

4.1 Shared Resource Access Protocols

CPIs that encapsulate exclusive access to a shared resource must provide appropriate priority semantics for the associated critical section. We assume

that each such CPI encapsulates a complete critical section. Encapsulation of a shared resource for which only a portion of execution must be locked can be realized with one or more CPIs propagating priority for reentrant access to the resource, and other CPIs encapsulating locking semantics for nonreentrant, exclusive access.

Nested locking (acquiring a second lock while already holding a lock) then can be achieved through a chain of requests: a CPI encapsulating one lock can make a request to another CPI encapsulating the second lock. This is illustrated in Fig. 2.

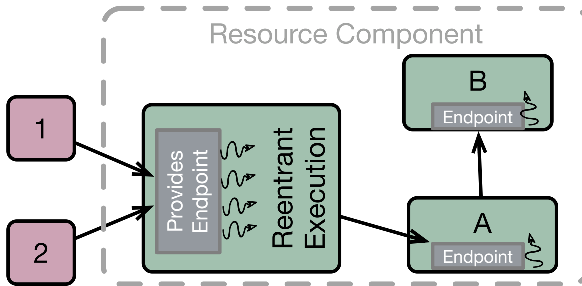


Fig. 2: Active components 1 and 2 send requests to a CPI for Reentrant Execution that is provided by a passive Resource Component, which also provides and uses CPIs for exclusive execution protected by Lock A, and nested locking by Lock B. Lock acquisition ordering is enforced by the defined connections; an acyclic connection digraph is deadlock free.

It is straightforward to implement the Non-Preemptive Critical Sections Protocol (NPCS) and the Immediate Priority Ceiling Protocol (IPCP). Both are achieved in our framework by tagging an interface with the "fixed" priority protocol attribute (as described in Section 5) and providing a single listening thread to its endpoint, assigning the thread a fixed priority. This wraps the free implementation of these protocols that is provided by both the MCS and non-MCS builds of the seL4 kernel [30].[4] Under either protocol, if the CPI's procedure performs a nested request to another downstream CPI, the priority of the *thread*, not the *originating task*, must be passed with the request message. This was omitted from our prior work in [7], and has been added in this extension.

The NPCS is realized under traditional, fixed-priority, preemptive scheduling by assigning a CPI the maximum system priority (255 in seL4). Once a request is received by the interface, it cannot be preempted. Under round-robin

[4] We did not enable MCS features when building the kernel. Budget depletion during execution in a CPI encapsulating nonreentrant critical sections can starve higher-priority requests, as noted in [13]. Further, the seL4 sporadic server implementation induces budget fragmentation even when threads are preempted by others of higher priority [30]; the standard sporadic server should only schedule a replenishment when a thread voluntarily yields the processor [31, 32].

scheduling of threads having equal priorities, NPCS comes with the additional constraint that all tasks (and their originating components' threads) are restricted to priorities less than the maximum (i.e. 0-254 in seL4). This guarantees that execution in a critical section is not preempted by a new request, which implies that two critical sections cannot execute concurrently: one critical section would necessarily have to begin execution before the other, and for the second critical section to execute, it would have to be in response to a request preempting the first. Otherwise, if another task has priority 255, round-robin scheduling could allow preemption of the non-preemptive section.

Because NPCS induces blocking time on all tasks in a system, the IPCP is typically preferred as an alternative fixed-priority resource access protocol. As noted in [30], IPCP is straightforward to implement by providing an endpoint with a single thread, assigned a priority equal to the priority ceiling of the CPI. With only a single thread listening on the endpoint, no additional lock variable is necessary. IPCP is, as defined in [23], a deadlock avoidant protocol. However, under seL4's priority-based round-robin scheduler, deadlocks can occur. Consider a task, τ_1, that acquires some lock A, then lock B while still holding A (lock acquisition is nested). Another task, τ_2, acquires lock B, then lock A. If τ_1 and τ_2 have priorities equal to the priority ceiling and τ_1 acquires A, it may be switched out for τ_2, which could then acquire B and proceed to wait on lock A. At this point, τ_1 is switched back in, and attempts to obtain lock B, causing deadlock.

One solution to this is to assign "fixed" CPIs a priority equal to PC+1. However, under our component model, deadlock could still occur, even without round-robin scheduling. The possibility of deadlock requires execution paths that acquire locks in opposite orders. This would imply two CPIs, each encapsulating a lock, that each have connections to the other's interface. Given misconfigured CPIs, one might request the other, which could request the first in turn, causing deadlock within a single task's control flow. To guarantee the absence of deadlock one would have to ensure that no cycles exist in the digraph of connections. New to this extension, we provide a tool to parse and detect cycles in a provided system specification, which alerts to possible deadlock.

We do not implement the original Priority Ceiling Protocol, as described in [22]. The Immediate Priority Ceiling Protocol assigns static priorities to component interfaces according to the priority ceiling of the corresponding lock. Because connections are defined statically in the CAmkES specification, the priority ceiling can be computed offline using our parser. However, the original Priority Ceiling Protocol requires the tracking of a priority ceiling among all *currently acquired* locks; because this introduces additional online global state even among non-interacting components, we do not provide this protocol as an option.

4.2 Priority Inheritance Protocol

An interface will provide locking with Priority Inheritance Protocol semantics if tagged with the "inherited" priority protocol attribute. For these interfaces,

our framework supplies the CPI with five variables: a boolean lock variable (which, as we will explain, need not be accessed using atomic operations), a pointer to the thread holding the lock (implemented as an `seL4_CPtr` to its Thread Control Block), the current inherited priority of that thread, a unique identifier corresponding to the task for which the component is currently executing, and a function pointer that facilitates nested priority inheritance.

To allow for priority inheritance, these CPIs must (1) execute a request at the priority of the requesting thread, and (2) handle concurrent requests, allowing temporary preemption to enable the lock holder to inherit any higher priorities associated with these requests. To achieve these goals, we give each CPI a pool of threads, all waiting for requests on the underlying endpoint. To ensure thread availability whenever a request arrives, the size of the pool is set equal to the number of possible concurrent requests, as illustrated in Fig. 3. Because CAmkES provides a static specification of CPIs and request connections, this value is straightforward to determine.

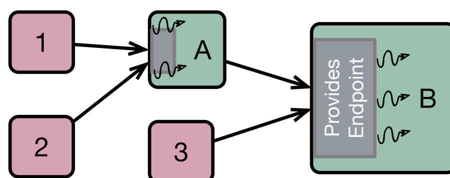


Fig. 3: Passive component A's CPI is used by active components 1 and 2 so a pool of 2 threads waits on the underlying endpoint. Passive component B's CPI is used by both of component A's CPI threads and by active component 3, so it has 3 threads for 3 tasks.

The threads belong to the same CPI and share an address space, so they all have access to the CPI-scoped variables used by the protocol mechanisms. Threads wait on the endpoint at the highest priority among all tasks that use the interface, referred to as its priority ceiling (PC). This ensures that if a request preempts existing execution in the CPI on behalf of another request through the interface, the thread handling the new request will be of sufficiently high priority to begin execution.

Our implementation of Priority Inheritance Protocol is illustrated in Fig. 4. When a request arrives, the responding thread ① checks the lock. If the lock is already held (**State Locked**), it proceeds to ② check the inherited priority variable against the priority of the requesting thread, which is passed to it over the endpoint as part of the request message.[5] If the request priority is higher, it is inherited by the thread currently holding the lock: the responding thread ③ updates the inherited priority variable, then ④ elevates the priority of the locking thread's Thread Control Block (TCB) with a call to

[5]In CAmkES, procedure interfaces are declared similarly to C-style functions: they may include one or more parameters, which specify the set of arguments that must be passed as part of the IPC message data.

`seL4_TCB_SetPriority`. If the locking thread is currently blocked on a downstream request, a corresponding pointer ⑤ will have been set to a function that facilitates propagating the inherited priority to the requested CPI (described below); the responding thread calls this function. At this point, it ⑥ waits for a signal indicating that the lock has been freed.

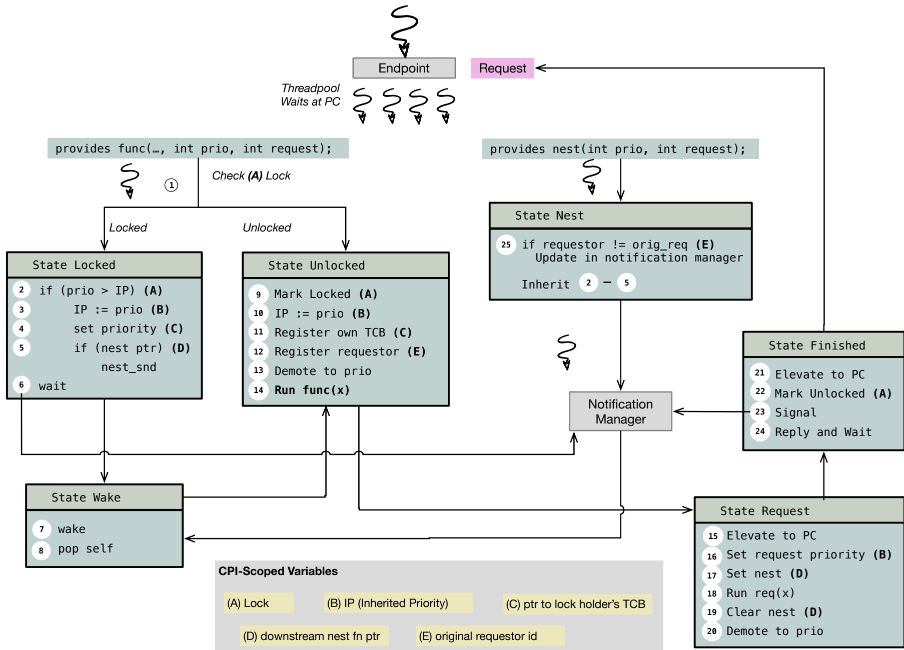


Fig. 4: Implementation of Priority Inheritance Protocol

If, however, the lock is unlocked (**State Unlocked**), the thread ⑨ marks the lock as locked, ⑩ sets the inherited priority variable to the request priority, ⑪ sets the TCB pointer to itself, and ⑫ registers the unique identifier of the originating task (passed as part of the request message). It then ⑬ demotes its priority to the request priority and ⑭ runs the interface's procedure code to handle the request. Once complete (**State Finished**), it ㉑ promotes itself back to the priority ceiling, ㉒ marks the lock as unlocked, ㉓ signals any threads waiting for the lock, then finally ㉔ replies to the requestor and returns to waiting on the endpoint.

The seL4 kernel provides *notification objects*, which are simple signaling mechanisms that support blocked waiting. When a notification object receives a signal, a single waiting thread (if there are any) is awakened. If the seL4 kernel is compiled with default settings, it wakes waiting threads in FIFO order. Notification objects are priority aware when compiled with MCS settings; in this case, waiting threads are tracked in a priority-ordered linked list [33]. In either case, it is unsuitable to provide a single notification object upon which

all threads requesting a held lock must wait. Threads wait at the priority ceiling of the interface, but the thread handling the request with the highest priority must be guaranteed to be the first to obtain the lock when it becomes available. As such, we implement a request-priority-aware signaling mechanism that we call a *notification manager*.

The notification manager contains a priority queue (implemented as a max-heap) of notification objects, sorted by priority. Because a max-heap does not maintain a stable sort, we additionally track the insertion order of all objects into the priority queue;[6] two objects of equal priority are sorted so that the first object inserted is higher in the heap's ordering. When initialized, the notification manager creates an array of notification objects, equal to the size of the thread pool, by using the CAmkES seL4 object allocator. The notification manager reveals two public functions, wait and signal, similar to the seL4 system calls of the same names for notification objects. A pointer to the request priority and the unique identifier of the task originating the request are passed with the wait call, allowing the notification manager to retrieve a notification object from the free list, then insert it into the heap. The wait function then uses a system call to wait on that notification object. The notification manager's signal function signals the notification object at the head of the heap (**State Wake**). The awakened thread ⑦ returns from the seL4 wait system call; its control flow remains in the notification manager's wait function, which ⑧ pops its notification object from the head of the priority queue. At this point, the thread ⑨ proceeds as if it had found the lock available.

If a nested request to a downstream CPI is performed as part of the interface's procedure code (**State Request**), our implementation of the protocol wraps the request so that the thread ⑮ elevates its priority to the PC, ⑯ sets the request priority to the current inherited priority, ⑰ sets the function pointer to facilitate downstream nested inheritance, then ⑱ sends the request. On receiving a reply, it ⑲ clears the function pointer, then ⑳ demotes its priority back to its current inherited priority. This avoids possible data races associated with a higher priority request arriving when the pointer is in an inconsistent state (i.e., when the lock-holder has set the pointer but hasn't yet made the request, or when the lock-holder has received a reply but hasn't yet cleared the pointer).

To enable nested priority inheritance, any CPI in a nested request chain downstream of a CPI implementing PIP (except those with a fixed priority ceiling, i.e., IPCP or NPCS) provides an additional method, `nest`, to receive inherited priority updates from upstream components. When sending a request to the CPI, an upstream component ⑰ sets a function pointer to the corresponding `nest` method; this allows new incoming requests with higher priorities to ⑤ perform nested propagation of the inherited priority, passing the new priority and the identifier of the thread waiting on the request to the downstream CPI. In a CPI implementing PIP, the `nest` method ㉕ checks

[6]The insert order increments with every insertion into the priority queue. It is is implemented as a 64-bit unsigned integer to avoid overflow.

if the corresponding requestor currently holds the lock. If not, it finds the requestor's node in the notification manager, elevates its priority, then recursively swaps the node with its parents until the max-heap property is satisfied. The method then performs steps ②–⑤ as necessary to elevate the priority of the lock holder and propagate the inherited priority to further downstream requests. Note that CPIs providing the `nest` method require an additional thread.

In the absence of round-robin scheduling of threads at the same priority, all execution of our protocol (steps ①–⑬, ⑯–⑳, and ㉒–㉕ in Fig. 4) occurs at the priority ceiling, and so cannot be preempted by new requests. The only time that execution can be preempted by a request is when the thread is executing the CPI procedure (steps ⑭–⑮ and ㉑). If preempted here, it will remain preempted while the responding thread executes steps ①–⑤ for the primary method or steps ㉕ and ②–⑤ for the `nest` method. Thus, there can only be two threads from the pool active at any given time: either when there is one thread executing (steps ⑭–⑮ or ㉑) and one at the priority ceiling (steps ①–⑤ or ㉕), or when the thread holding the lock signals the notification manager, waking another thread. In the latter case, the signaled thread will proceed through steps ⑦–⑨, while the signaling thread proceeds to ㉔. As both threads are executing at the priority ceiling, no new requests can arrive, and so the thread just awakened will be guaranteed that when it pops the head of the heap, it will have its own notification object, and that the lock will not be acquired by another thread before it proceeds to set the lock. Thus, by priority semantics, our protocol is race-free.

However, under round-robin scheduling of same-priority threads, a race may occur: a responding thread running the mechanisms of our protocol can be swapped out for a requestor at the priority ceiling, which would wake another thread from the pool. In our prior work, we addressed this problem in the absence of nested locking [7], but these same arguments do not apply to our implementation of nested Priority Inheritance Protocol. We therefore implement and evaluate our protocols in the context of traditional fixed-priority preemptive scheduling, and defer consideration of round-robin scheduling to future work.

4.3 Priority Propagation

We also support CPIs that encapsulate reentrant functionality shared among multiple tasks. So that end-to-end task execution follows the semantics of fixed-priority, preemptive scheduling as described in Section 3, we require that task priority propagates with control flow across request paths. Such a CPI, having an interface tagged with the "propagated" priority protocol attribute, must (1) execute requests at the priority of the requesting thread, and (2) handle concurrent requests in a preemptive fashion, i.e. a CPI may preempt its own procedure's execution if it receives a request from a higher priority task. Similarly to PIP, these CPIs are again supplied with a pool of threads, the size of which is equal to the number of possible concurrent requests. Under

traditional fixed-priority preemptive scheduling, these threads are set to wait on the endpoint at the priority ceiling.

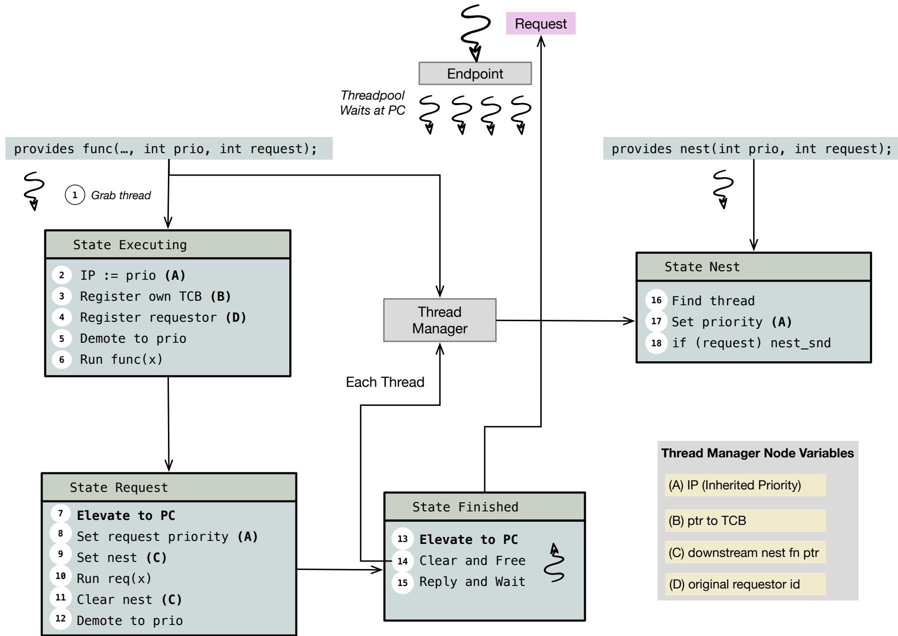


Fig. 5: Implementation of Priority Propagation

Our implementation of Priority Propagation is illustrated in Fig. 5. When a request arrives, the thread handling the request ① retrieves a node from a Thread Manager, a data structure that allows the CPI to track requests and associated priorities. The Thread Manager maintains two singly-linked lists to track free and in-use nodes. After retrieving a free node (**State Executing**) and adding it to the in-use list, the thread ② sets the node's priority to the request priority, ③ sets the node's TCB pointer to itself, and ④ registers the unique identifier of the originating task (passed as part of the request message). It ⑤ sets its priority to that of the requesting thread, per the priority information that is passed to it over the endpoint as part of the request message, and then ⑥ executes its procedure, running the subtask at the originating task's priority. On completion (**State Finished**), it ⑬ elevates its priority back to its original waiting priority, ⑭ clears the TCB of its corresponding Thread Manager node and returns it to the free list, then ⑮ replies to the requestor and returns to waiting on the endpoint. By receiving a request and sending the reply at the priority ceiling of the interface, these transitions between subtasks are equivalent to critical sections with IPCP semantics (under traditional fixed-priority preemptive scheduling) and induce equivalent blocking time as was discussed in Section 3.2.1.

If a CPI implementing Priority Propagation handles nested requests that originate from a CPI implementing PIP, it must also provide a `nest` method. Similarly to a PIP CPI, an additional thread must be provided to its pool. The method (16) finds the node in the Thread Manager executing on behalf of the corresponding requestor, (17) elevates its priority, then (18) if the thread is blocked on a further downstream request, it calls that CPI's `nest` function. Similarly to the PIP implementation, when sending a request (**State Request**), a thread must (7) elevate its priority to the PC, (8) set its Thread Manager node to the current inherited priority, (9) set the function pointer to facilitate downstream nested inheritance, then (10) send the request. On receiving a reply, it (11) clears the function pointer, then (12) demotes its priority back to its current inherited priority.

5 Implementation and Usability Enhancements

Our userspace implementation targets closed embedded real-time systems based on a non-MCS build of the seL4 kernel, running atop unicore or fully-partitioned multicore hardware. We defer exploration of kernel-enforced properties in open systems with untrusted components, or using MCS kernel features, to future work.

We implemented our framework with the goal of staying as true to the CAmkES language and design philosophy as possible. Our implementation leverages existing techniques used by the CAmkES framework, including the Jinja template engine, to provide support for several protocols using only 659 lines of code, as summarized in Table 1. It minimizes, as much as possible, the extent of changes necessary for existing CAmkES application systems to incorporate its functionality. We now (1) describe how we met this goal, and (2) provide an overview of how a developer would use our framework.

Implementation	Lines of Code
Base Framework C Code	102
Priority Inheritance Protocol C Code	118
Notification Manager C Code	139
Priority Propagation C Code	126
C Macros	16
CAmkES Macros	9
CAmkES Connector Declarations	100
CAmkES Connector Jinja Templates	49
Total	659

Table 1: Implementation Lines of Code

CAmkES allows components to be declared with a set of attributes, to which values can be assigned. These attributes are compiled into symbols in the component binary, and the user-provided source code for the component can use them as variables. CAmkES additionally provides several built-in attributes. For example, if a component provides a procedure interface named

"iface", CAmkES automatically defines an attribute `iface_priority`, the value of which defines the priority of any thread that handles requests on the underlying endpoint. An active component has an automatic attribute simply called `_priority` that sets the priority of its execution thread. A user can add either of these to a component's declared set of attributes to make the priority available as a variable in the source code. This is necessary for priority introspection without modification to underlying CAmkES or seL4 code (and, therefore, for priority to be passed with a request message) because seL4 does not provide a system call for threads to read their current priority level.

We provide three additional attributes. For components originating a task, `requestor` provides a unique identifier to track the task's control flow across components, as described in Section 4. Each CPI under our framework must be assigned the attributes `_num_threads` (which defines the number of threads in the pool waiting on the endpoint) and `_priority_protocol` (which can be one of "propagated," "inherited," or "fixed"). These attribute names must be prefixed with the name of their associated interface, similarly to `_priority.` Because CAmkES supports C preprocessor commands, we provide a function macro for ease of use, that automatically generates the attributes for each provided task and interface. Since each task's identifier has no semantic meaning, beyond being a unique integer identifier, the `__COUNTER__` symbol can be used to easily assign a value.

CAmkES provides a library of standard connectors to component interfaces. Among these is the `seL4RPCCall`, which establishes a connection for RPC invocations as synchronous IPC to a CPI. CAmkES uses Jinja templates [34] to generate much of the underlying code, including seL4 system calls, to broker communication over a given connector type. We define a new class of connectors, `seL4RPCCallPrioritized`, that inherits much of its functionality from the seL4RPCCall's templates.

In CAmkES, a particular connector type must specify the number of threads bound to the underlying endpoint of the target CPI. Because our interfaces may require different numbers of threads, depending on the number of possible requestors, we cannot limit ourselves to a single connector type. To avoid changing the underlying CAmkES parser to support providing this as an attribute to the connector, we provide a CAmkES connector definition file with 100 declared seL4RPCCallPrioritized connector types, supporting threadpools from size 1 to 100. We additionally provide a function macro that creates a connection with the appropriate connector type, when provided the number of threads. Because the number of threads must be provided twice (to the `_num_threads` attribute and to the connector macro), we support defining this as an object macro. We use appropriate stringification such that it can be passed to the function macro. For example, to establish a connection from a client to a server interface, "iface," that uses 2 threads, a user would write the following in the CAmkES language:

```
#define server_iface_num_threads 2
...
connection rpc(server_iface_num_threads)
```

```
    conn(from client.iface, to server.iface);
...
server.iface_num_threads = server_iface_num_threads;
```

Our template code additionally inserts hooks into the appropriate functions in our library: initialization, and function calls before and after the interface procedure runs. This ensures that users of our framework do not have to remember to manually insert the necessary hooks into the provided component source code. For initialization, we leverage the existing `__init` function that CAmkES declares for each procedure interface. Normally, a user would provide an appropriate function definition; our template defines it instead, ensuring that it is called at component initialization. To allow additional user-defined initialization, we provide an `_init` function declaration (note the single, rather than double, underscore) that is called at the end of our template's initialization.

Both priority and the task identifier must be passed as function arguments for an RPC call to a CPI implementing our protocols, which is realized by requiring that the procedure's C-style function declaration includes both as the last input parameters. This means that (1) the user-supplied source code for the requesting component procedure must include these as function arguments when a procedure is called, and (2) the user-supplied source code for the handling component must have a function definition with these as the last parameter. Instead of passing it directly through the functional interface, these parafunctional properties could be passed alongside the functional attributes/parameters in the request message, leveraging the CAmkES model and encoding for passing parameters between components. However, we defer that refinement to future work. Both approaches, while different in the requirements they impose on the user of the framework, would be equivalent when compiled down to seL4 binaries.

To enable the `nest` function, a user of our framework must add the method to the procedure associated with each CPI that must provide it. This is done simply by adding the following line to the CAmkES procedure declaration:

```
void nest(in int priority, in int requestor);
```

The procedure must also be defined for any component using the procedure. We provide a C function macro to simplify this: for a component that provides a procedure interface named "iface", invoke the macro with `NEST(iface)`. To perform nested requests, i.e., to achieve the functionality of steps (16-21) in Fig. 4 and (8-13) in Fig. 5, we provide another C function macro. Because the CAmkES parser describes connections from *components* to *CPIs*, and a component may have multiple CPIs from which a downstream request can originate, we were not able to wrap the functionality entirely in Jinja templates. Nonetheless, the macro is intended to be easy to use in place of the default C-style function calls for invoking requests. Its signature is as follows:

```
REQUEST(interface_from,interface_to,method,...)
```

Here, `interface_from` is the name of the CPI sending the request, `interface_to` is the recipient CPI, and `method` is the name of the recipient procedure's method being invoked. As a variadic function macro, additional parameters required by the method can also be passed. The macro will also return any value returned by the invoked method. The integration of the macro's functionality with our additions to the CAmkES connector templates constrain connections to only being declared with a single "to" end; existing connections declared with multiple "to" ends must therefore be broken up into separate connections. Our templates check for this, and will fail to compile (with appropriate error messages) if this requirement is not met.

Thus the framework, in its current form, relies on the user to make a few changes to their component source code, and not just to the component-level and application-level CAmkES specification. However, these changes are minimal and largely necessary to avoid modifying the underlying CAmkES parser and seL4 kernel. Additionally, our framework contains appropriate checks, such that if attributes are incorrectly specified, or parameters and method definitions are not provided, the application system will fail to compile. Together, its ease of use and its compilation checks to avoid misconfiguration make our framework a good option for developing closed real-time systems.

We additionally provide a parser to help an application designer determine the priority ceiling and number of threads to assign each CPI that uses our protocol and to detect possible cycles. It recursively traverses the connection graph defined by a CAmkES system specification file, determining the maximum task priority among possible requestors to each CPI. It additionally counts the number of possible concurrent requests, for each CPI enumerating the tasks for which it may execute. It identifies CPIs using the "fixed" priority protocol, assigning them only a single thread. It also identifies CPIs using the "inherited" protocol, and remains aware of the fact that a CPI encapsulating a locking protocol cannot send multiple concurrent nested requests to downstream CPIs. Finally, it assigns an additional thread to each non-"fixed" CPI downstream of a PIP interface to guarantee availability for the `nest` method.

It is worth noting that because CAmkES describes connections from *components* to *CPIs*, the CAmkES specification by itself lacks the information needed to determine the transitive closure of a request chain, as a component with multiple CPIs might make nested requests as part of the procedure of only one of those; this would not be evident from the digraph. This means, without changes to the CAmkES parser, that (1) the presence of a cycle does not necessarily imply a call chain loop with deadlock potential,[7] and (2) that a component on the "from" side of a connection that has multiple CPIs does not necessarily send requests from both CPIs, meaning that priority or thread counts might be overestimated. Our parser recognizes these cases, and warns the user of the potential issue when it arises.

[7] For example, two components might each have two CPIs, one which sends a request to the other component, and one which receives a request from the other component. Despite the digraph of requests forming a cycle between the two components, each request path involves a distinct set of CPIs, and therefore cannot deadlock.

6 Evaluation

We evaluated our library using the CAmkES 3.10.0 framework, targeting version 12.1.0 of the seL4 kernel, testing synthetic task sets on both Intel x86-64 and ARMv8 AARCH32 ISA hardware platforms. We compiled using the `RELEASE=TRUE` and `SIMULATION=FALSE` directives and enabled kernel printing. For the Intel platform, we used a system with two Intel Xeon Gold 6130 Skylake processors running at 2.1 GHz, and with 32GB of memory. We disabled HyperThreading, SpeedStep, and TurboBoost. For the ARM platform, we used a Raspberry Pi Model 3 B+, which has a 64-bit ARMv8 Cortex-A53 Broadcom BCM2837B0 SoC with 1GB of RAM. We disabled the L2 cache, and clocked its four cores to 700 MHz.[8] Despite the hardware supporting the AARCH64 instruction set, seL4 currently only supports 32-bit mode on the Raspberry Pi, so we compiled using the `AARCH32=TRUE` directive.[9] On the Raspberry Pi, we additionally enabled userspace access to the ARM Performance Monitor Unit (PMU) to allow our system to measure and print elapsed cycles. We ran each task set according to traditional, fixed-priority preemptive semantics (i.e. a thread is not preempted or switched out for another thread of equal priority unless it yields the processor). This was realized by configuring the seL4 kernel with the round-robin timeslice set to a sufficiently large value.[10]

6.1 Protocol Overheads

We begin by measuring the overheads induced by our protocol. To support fine-grained microbenchmarking, we measure elapsed cycles (using rdtsc on Intel, and reading directly from the cycle count register on the ARM PMU) for all measurements. Because reading from the cycle counter incurs its own overhead, we first benchmark these reads by measuring the elapsed cycles between two successive cycle counts. Results are summarized in Table 2.

Dividing the maximum cycles measured between two back-to-back cycle counter reads, the clock speed of each platform gives a bound on the temporal resolution of our measurements of a little under 14 nsec on the Intel Xeon, and 12 nsec on the Raspberry Pi. We individually measure the overheads for both sending requests over an endpoint (Call) and replying to the request (Reply), separately measuring the overheads of our PIP implementation for requests to a CPI with an already-acquired lock (locked) versus those with an available lock (unlocked). We additionally measure the overheads of nested requests from a CPI implementing PIP to CPIs implementing both PIP and priority propagation, respectively, reporting both the call and reply times, as well as the time to send a nested priority inheritance update (Nest). We compare these

[8]The processor supports a CPU clock speed of 1.4 GHz. However, as noted in [35], this frequency cannot be sustained continuously, and may lead to throttling and instability. To maintain predictability, we boot the Raspberry Pi with a constant 700 MHz CPU clock speed, set the GPU processor core to 250 MHz, and disable throttling. Details can be found at https://www.raspberrypi.com/documentation/computers/config_txt.html

[9]https://docs.sel4.systems/Hardware/Rpi3.html

[10]We set the `CONFIG_TIMER_TICK_MS` kernel configuration parameter to 1,000,000 (1000 seconds), sufficiently long to ensure appropriate behavior in our tests.

	Intel Xeon Gold 6130				Raspberry Pi Model 3 B+			
	min	max	mean	std	min	max	mean	std
Read cycle counter	22	28	25	1.3	8	8	8	0
Call, built-in	2478	4568	2528	211	563	3359	619	278
Reply, built-in	2494	2836	2519	34	416	1298	449	86
Call, fixed	3178	4810	3306	189	834	2591	1010	179
Reply, fixed	2590	3342	2659	142	422	954	466	94
Call, propagated	4348	6830	4574	287	2085	5516	2368	351
Reply, propagated	3536	3960	3567	41	1606	2335	1694	76
Call, inherited, unlocked	4344	6834	4625	288	1935	5381	2226	335
Call, inherited, locked	4372	6282	4603	245	1966	5052	2242	314
Reply, inherited, unlocked	3520	4028	3549	50	1483	2095	1531	65
Reply, inherited, locked	3514	3788	3547	28	1470	2319	1531	96
Call, PIP to PIP	5364	6878	5618	240	3079	5696	3355	279
Reply, PIP to PIP	5402	6300	5462	89	2898	4109	3019	121
Nest, PIP to PIP	5144	6720	5346	303	1175	3315	1844	446
Call, PIP to propagated	5292	6306	5539	196	3165	6068	3452	308
Reply, PIP to propagated	4558	5318	4679	130	2563	3552	2668	98
Nest, PIP to propagated	6068	7530	6385	243	1838	5088	2243	351
Dispatcher Overhead	58	240	63	19	74	136	74	6

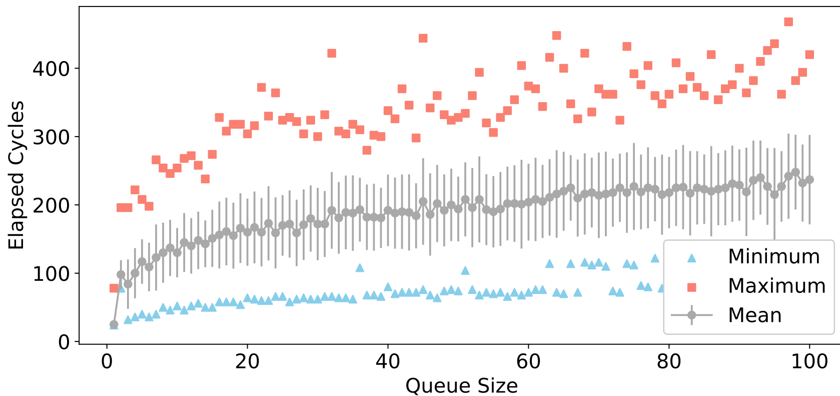
Table 2: Overheads (in cycles) for Protocol Mechanisms

overheads for our various protocols (propagated, inherited for PIP, and fixed for IPCP and NPCS) to the overhead of a request over the CAmkES built-in seL4RPCCall connector; while our protocols do induce additional overhead, the maximum values we measured (a nested call and reply to a CPI with priority inheritance induced up to about 13,200 cycles of overhead on Intel and about 9,800 cycles on ARM) equates to less than 6.3 μs on Intel and 14.1 μs on ARM, which is suitably low for task sets running with periods as small as 10 ms.

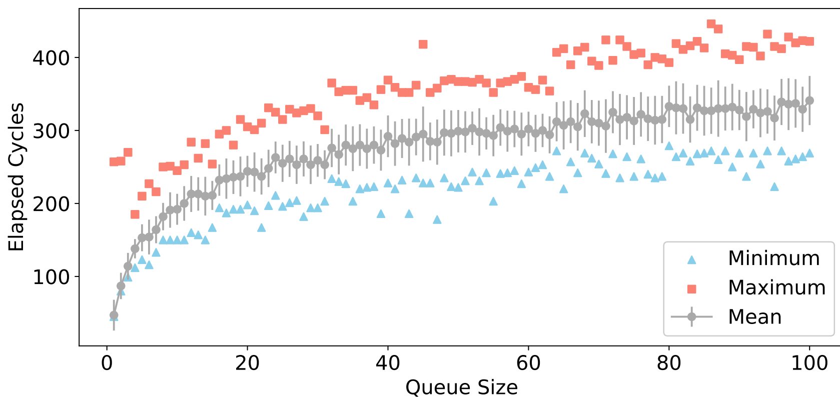
Furthermore, benchmarked performance numbers for the seL4 kernel without the CAmkES framework report an average overhead of 383 and 389 cycles, respectively for IPC call and reply between threads on an Intel x86_64 Skylake platform; and 404 and 409 cycles, respectively on the ARMv8 platform in 64-bit mode [36]. Even with nested priority inheritance, our mean overheads are only about 14.4× this on the Xeon Gold 6130 and 7.84× this on the Raspberry Pi Model 3 B+. Benchmarked performance numbers from Patina [13] are only available from an ARM Cortex-A9 processor running on the Zynq-70000 XC7Z020, and as Patina's seL4 implementation is not open-sourced, we cannot perform a direct comparison. However, their maximum reported overheads for requests to the mutex service were 11,165 cycles (unlocked) and 13,918 cycles (locked); more than the maximum observed overhead on ARM of our mechanisms, even for nested locking (which Patina does not support).

We additionally measure the overhead induced by priority queues realized by different heap sizes within our notification manager. For a given heap size n, we initialize the heap to hold $n-1$ notification objects with random priorities, then measure the elapsed cycles to push one more notification object into the heap, then pop the notification object with the greatest priority (and, among

those of equal priority, the lowest insertion order). Times are plotted in Fig. 6, with error bars indicating one standard deviation about the mean. Even the maximum values observed are upper-bounded by 468 cycles (less than one fourth of a microsecond) on Intel and 446 cycles (less than three fourths of a microsecond) on ARM. This (1) demonstrates suitably low overhead of our notification object heap itself even as the number of elements it holds grows to 100 (a larger value than many realistic scenarios would experience), and (2) suggests that the overheads for our priority inheritance protocol are dominated by the costs of the system calls it uses.



(a) Intel Xeon Gold 6130



(b) Raspberry Pi Model 3 B+

Fig. 6: Measured Priority Queue Overheads (in cycles)

6.2 Empirical Evaluation of Synthetic Task Sets

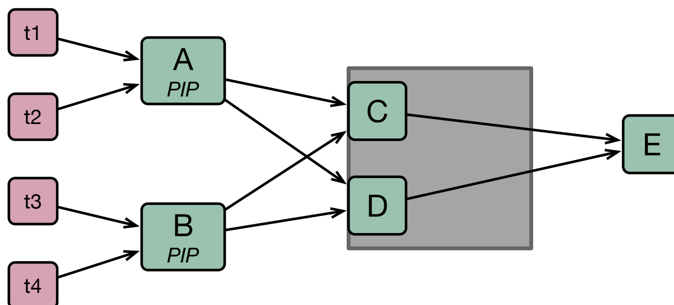


Fig. 7: Task and Component Test System

To facilitate checking the schedulability of actual task sets running in CAmkES atop seL4 on our selected hardware platforms, we generate synthetic task sets over a representative topology of interacting components (the one illustrated in Fig. 7), all running on a single core. In each task set, components originating tasks t1 and t2 both request a CPI provided by component A, while those originating t3 and t4 both request a CPI provided by component B. Both A and B encapsulate exclusive execution using PIP. We evaluate the 4 configurations outlined in Table 3.

	CPI C	**CPI D**	**CPI E**
Configuration 1	PIP	Propagation	PIP
Configuration 2	PIP	Propagation	Propagation
Configuration 3	IPCP	PIP	Propagation
Configuration 4	IPCP	Propagation	PIP

Table 3: Synthetic Taskset CPI Configurations

For each configuration, we generate task sets with utilizations ranging from 0.1 to 1.0. For each utilization value, we generate 10 task sets: we (1) assign task utilizations according to the UUniSort algorithm [37],[11] (2) randomly select task periods from a set of harmonic values from 10 ms to 1 second,[12] then (3) assign task workloads and priorities appropriately, and (4) sort tasks by increasing workload. Each task is then decomposed into subtasks according to the component CPIs it traverses: we generate the workloads of each subtask (and therefore CPI) according to UUniSort (with the sum of the subtask workloads equal to the task workload). For each task where a CPI's workload has

[11] An alternative to UUniSort, the UUniFast algorithm, runs in linear time, where UUniSort is quasilinear. However, the elegance and simplicity of UUniSort, combined with our small input sizes, make it ideal for our offline synthetic task set generation.

[12] Restricting to a set of harmonic harmonic periods, instead of generating task periods according to the log-uniform distribution described in [38], allows trials with repeated hyperperiods to be performed efficiently.

already been determined (the CPI being shared with another task, for which it executes subtasks), we use the previously assigned value, and then generate remaining subtask workloads with UUniSort.

Each task set was run for 10 hyperperiods, with each task releasing up to 2000 jobs. We implemented periodic tasks by defining a component, the **Dispatcher**, which registers a periodic timeout with the CAmkES library's **TimeServer** component. The TimeServer is included among the reusable components released with CAmkES, though it does not natively support the Raspberry Pi. We developed a platform-specific header for the Raspberry Pi Model 3's BCM2837 firmware, which was realized in only 40 lines of code by hooking into the seL4 library's existing drivers for the board's timer hardware.

An instance of the Dispatcher is created for each task, jobs of which it dispatches via the built-in seL4RPCCall connector. Dispatchers are assigned a priority higher than the three tasks, which ensures that all Dispatcher initialization occurs before any task can execute, and that any task can be preempted by job release, such that the exact time of release can be recorded. Each Dispatcher sets a periodic timer according to its task's period. When the timer expires, the Dispatcher (1) issues an instruction to read from the cycle counter, (2) sends an RPC request to its associated task component, then, when it receives a reply, (3) reads again from the cycle counter. The worst-case overhead incurred by the Dispatcher to wait on the timer's underlying notification object, as well as the time it takes to determine job completion (both aggregated as the last line of Table 2) are subtracted from the elapsed time. If the resulting value does not exceed the task's period, we consider the job to have met its deadline.

Task workloads were synthesized by looping on subsequent floating point multiplication and addition operations. We measured the execution time, in cycles, for 10^5 iterations. On the Raspberry Pi, these values were 8.46×10^5, 8.49×10^5, and 8.46×10^5 cycles respectively. The measured distributions were relatively narrow, showing a standard deviation of only 40 cycles (under 60 ns) on the Raspberry Pi. The worst-case overhead of the Dispatcher's communication over the endpoint with its task component, as well as its two reads from the cycle counter (shown in the first 3 rows of Table 2) are subtracted from the execution times assigned to each task, before workload iterations are assigned to individual subtasks.

To enable the testing of multiple task systems in a single run, we encapsulate the components that compose each task system, along with an instance of a **System Dispatcher** component, into a single hierarchical component, as illustrated in Fig. 8. We then define a **Root Dispatcher** component, a single instance of which runs at the highest system priority and (1) triggers the release of each task system in turn via a synchronous RPC request to that task system's System Dispatcher. Because of the function call semantics of these requests, the Root Dispatcher only releases the next task system (9) when the previous one has completed. The System Dispatcher sends an asynchronous notification (2-4) to each Dispatcher component in its task system. It runs

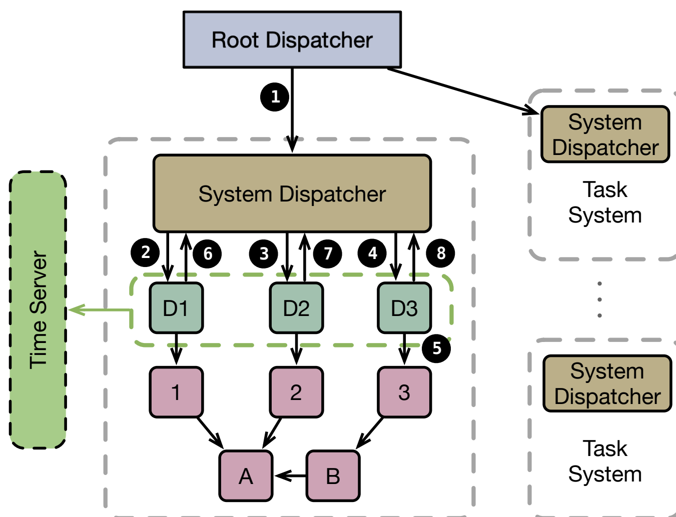


Fig. 8: Component Configuration Hierarchy to Sequentially Evaluate Multiple Task Systems

at the highest system priority, ensuring that all notifications are sent before any of the Dispatcher components can proceed. It then sets itself to the lowest system priority, so that it cannot interfere with task execution. Once the Dispatchers have (5) released jobs for 10 hyperperiods, they update deadline miss statistics in a page of memory shared with the System Dispatcher, then (6-8) notify it of their completion. Once all tasks have completed, the System Dispatcher prints the miss statistics to the COM interface, then replies to the Root Dispatcher's request. Note that, due to memory constraints, we were limited to configurations of only 5 task systems, requiring multiple separate configurations to be statically compiled and independently evaluated to cover all generated task systems.

Perhaps unsurprisingly given the predictable and well-bounded nature of the task execution times, and of the overheads exhibited by our library, when running on both hardware platforms, no deadlines were missed for any of our tested task sets, even those having a utilization of 1.0.

7 Conclusion and Future Work

In this paper, we extended our concurrency framework presented in [7] to support nested lock acquisition, including nested priority inheritance. The results of our evaluations demonstrate that our extensions to the CAmkES component framework can prioritize cross-component control flows effectively. Reentrant CPIs execute at the priorities of the requesting tasks, while CPIs encapsulating critical sections use priority-based locking protocols without the need for additional atomic operations. With no deadline misses seen even at full CPU utilization for the task sets we evaluated, our evaluations show that those

extensions allow CAmkES to provide suitable end-to-end timing guarantees for real-time systems.

As future work, we intend to further extend the concurrency framework presented in this paper in several ways: prevention of race conditions under nested locking with round-robin scheduling of threads at the same priority (e.g., via appropriate priority laddering techniques); expansion of our framework to support end-to-end timing guarantees across asynchronous event notifications; modification of the CAmkES parser to automatically add priority and task identifier metadata to RPC request messages, removing the need for the component programmer to add this parameter to each CPI signature; and a mechanism supporting transitive closure over request chains, allowing more robust deadlock detection and alerting to invalid component request configurations.

Appendix A Hyperbolic Bound with Blocking Times

The hyperbolic bound for rate-monotonic (RM) schedulability is given by Theorem 1 of [28]:

Theorem 1 *Let $\Gamma = \{\tau_1, \ldots, \tau_n\}$ be a set of n periodic tasks, where each task τ_i is characterized by a processor utilization U_i. Then, Γ is schedulable with the RM algorithm if*

$$\prod_{i=1}^{n}(U_i + 1) \leq 2 \tag{A1}$$

The proof of this theorem in [28] assumes that tasks are ordered by increasing periods; while this does not lose generality in the case that all tasks are assigned unique priorities, we are interested in the case where some tasks may have equal periods, and are therefore assigned equal priorities. We begin by considering schedulability for each task.

Lemma 1.1 *In the absence of blocking induced by shared resource constraints, for a set Γ of periodic tasks, ordered by strictly decreasing priority, a task τ_i is schedulable under RM if*

$$\prod_{j=1}^{i}(U_j + 1) \leq 2 \tag{A2}$$

Proof By Theorem 1, the task system $\Gamma_i = \{\tau_1, \ldots, \tau_i\}$ is schedulable if

$$\prod_{j=1}^{i}(U_j + 1) \leq 2 \tag{A3}$$

This implies that task τ_i is schedulable. The addition of tasks of lower priority, i.e. any task τ_j for which $j > i$, will not affect the schedulability of τ_i. Thus, Eqn A3 holds for all $\tau_i \in \Gamma$. □

Lemma 1.2 *In the absence of blocking induced by shared resource constraints, for a set Γ of periodic tasks, a task τ_i with priority P_i is schedulable under RM if*

$$\prod_{\tau_j : P_j \geq P_i}(U_j + 1) \leq 2 \tag{A4}$$

Proof For any $\tau_i \in \Gamma$ for which $\nexists j : P_i = P_j, i \neq j$, Eqn A2 is equivalent.

Assume that there exists some τ_i for which other tasks in Γ have equal priority, i.e., there exists a set $K_i = \{k : P_i = P_k, i \neq k\}$. In the worst case, these tasks will be released before τ_i, and thus τ_i must wait for their completion as if they are higher priority tasks. In this case, without loss of generality, assume that tasks are ordered

by non-decreasing priorities such $\forall k \in K_i, k < i$. Then, by this ordering, Eqns A2 and A4 are equivalent. □

The proof of the following theorem follows closely with that of Theorem 16 in [22].

Theorem 2 *In the presence of blocking induced by shared resource constraints, a set* Γ *of* n *periodic tasks is schedulable under RM if*

$$\forall i, 1 \leq i \leq n, \qquad \prod_{\tau_j : P_j \geq P_i, j \neq i} (U_j + 1) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2 \tag{A5}$$

Proof Suppose that the equation is satisfied for all $i, 1 \leq i \leq n$. Then it follows that for each task τ_i, Eqn A4 will also be satisfied with $n = i$ and C_i replaced by $C_i* = (C_i + B_i)$. That is, in the absence of blocking, τ_i will be schedulable even if it executes for $(C_i + B_i)$ units of time. It follows that task τ_i, if it executes for only C_i units of time, can be delayed by B_i units of time and still meet its deadline. Hence, the theorem follows. □

References

[1] McIlroy, M.D.: Mass-produced software components. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct 1968, 79–85 (1969)

[2] Subramonian, V., Wang, N., Shen, L.-J., Gill, C.: The design and performance of configurable component middleware for distributed real-time and embedded systems. In: IEEE Real-Time Systems Symposium (RTSS), pp. 252–261 (2004)

[3] Subramonian, V., Deng, G., Gill, C., Balasubramanian, J., Shen, L.-J., Otte, W., Schmidt, D., Gokhale, A., Wang, N.: The design and performance of component middleware for qos-enabled deployment and configuration of dre systems. Journal of Systems and Software **80**(5) (2007)

[4] CORBA Component Model (version 3.0). https://www.omg.org/spec/CCM/3.0. OMG Document formal/02-06-65 (Accessed: 24 May, 2001)

[5] Kuz, I., Liu, Y., Gorton, I., Heiser, G.: Camkes: A component model for secure microkernel-based embedded systems. Journal of Systems and Software **80**(5), 687–699 (2007). https://doi.org/10.1016/j.jss.2006.08.039. Elsevier

[6] The seL4 microkernel. https://docs.sel4.systems/projects/sel4/. Accessed: 23 Jan, 2022. https://doi.org/10.5281/zenodo.591727

[7] Sudvarg, M., Gill, C.: A concurrency framework for priority-aware intercomponent requests in camkes on sel4. In: 2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 1–10 (2022). https://doi.org/10.1109/RTCSA55878.2022.00007

[8] CAmkES Manual. https://docs.sel4.systems/projects/camkes/manual.html. Accessed: 23 Jan, 2022

[9] Elphinstone, K., Heiser, G.: From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13, pp. 133–150. Association for Computing Machinery, New York, NY, USA (2013)

[10] Klein, G., Elphinstone, K., Heiser, G., *et al.*: Sel4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. SOSP '09, pp. 207–220. Association for Computing Machinery, New York, NY, USA (2009)

[11] Klein, G., Andronick, J., Elphinstone, K., et al.: Comprehensive formal verification of an os microkernel. ACM Trans. Comput. Syst. **32**(1) (2014)

[12] Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., Heiser, G.: Timing analysis of a protected operating system kernel. In: 2011 IEEE 32nd Real-Time Systems Symposium, pp. 339–348 (2011). https://doi.org/10.1109/RTSS.2011.38

[13] Jero, S., Furgala, J., Pan, R., *et al.*: Practical principle of least privilege for secure embedded systems. In: 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 1–13 (2021). https://doi.org/10.1109/RTAS52030.2021.00009

[14] AUTOSAR Classic Platform. https://www.autosar.org/standards/classic-platform/. Accessed: 25 Jan, 2022

[15] Specification of Operating System. https://www.autosar.org/fileadmin/user_upload/standards/classic/21-11/AUTOSAR_SWS_OS.pdf. Classic Platform Standard Release R21-11. Accessed: 25 Jan, 2022

[16] Kluge, F., Yu, C., Mische, J., Uhrig, S., Ungerer, T.: Implementing autosar scheduling and resource management on an embedded smt processor. In: Proceedings of Th 12th International Workshop on Software and Compilers for Embedded Systems. SCOPES '09, pp. 33–42. Association for Computing Machinery, New York, NY, USA (2009)

[17] Ford, B., Lepreau, J.: Evolving mach 3.0 to a migrating thread model. In: USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference). USENIX Association, San Francisco, CA (1994)

[18] Parmer, G.: The case for thread migration: Predictable ipc in a customizable and reliable os. In: Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 91–100 (2010)

[19] Wang, Q., Song, J., Parmer, G.: Execution stack management for hard real-time computation in a component-based os. In: 2011 IEEE 32nd Real-Time Systems Symposium, pp. 78–89 (2011). https://doi.org/10.1109/RTSS.2011.15

[20] Steinberg, U., Wolter, J., Hartig, H.: Fast component interaction for real-time systems. In: 17th Euromicro Conference on Real-Time Systems (ECRTS'05), pp. 89–97 (2005). https://doi.org/10.1109/ECRTS.2005.16

[21] Liedtke, J.: Improving ipc by kernel design. SIGOPS Oper. Syst. Rev. **27**(5), 175–188 (1993)

[22] Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers **39**(9), 1175–1185 (1990). https://doi.org/10.1109/12.57058

[23] Buttazzo, G.C.: Hard Real-Time Computing Systems, 3rd edn., pp. 205–248. Springer, New York (2011). Chap. 7

[24] Baker, T.P.: Stack-based scheduling for realtime processes. Real-Time Syst. **3**(1), 67–99 (1991)

[25] Steinberg, U., Böttcher, A., Kauer, B.: Timeslice donation in component-based systems. In: Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 16–23 (2010)

[26] Lipari, G., Lamastra, G., Abeni, L.: Task synchronization in reservation-based real-time systems. IEEE Transactions on Computers **53**(12), 1591–1601 (2004). https://doi.org/10.1109/TC.2004.120

[27] Steinberg, U., Kauer, B.: Nova: A microhypervisor-based secure virtualization architecture. In: Proceedings of the 5th European Conference on Computer Systems. EuroSys '10, pp. 209–222. Association for Computing Machinery, New York, NY, USA (2010)

[28] Bini, E., Buttazzo, G.C., Buttazzo, G.M.: Rate monotonic analysis: the hyperbolic bound. IEEE Transactions on Computers **52**(7), 933–942 (2003). https://doi.org/10.1109/TC.2003.1214341

[29] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)

[30] Lyons, A., McLeod, K., Almatary, H., Heiser, G.: Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time. In: ACM EuroSys Conference, pp. 1–16 (2018). https://doi.org/10.1145/3190508.3190539

[31] Sprunt, B.: Aperiodic task scheduling for real-time systems. Technical report, Carnegie Mellon University (1990)

[32] Stanovich, M., Baker, T.P., Wang, A.-I., Harbour, M.G.: Defects of the posix sporadic server and how to correct them. In: 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 35–45 (2010). https://doi.org/10.1109/RTAS.2010.34

[33] Mergendahl, S., Jero, S., Ward, B.C., Furgala, J., Parmer, G., Skowyra, R.: The thundering herd: Amplifying kernel interference to attack

response times. In: 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 95–107 (2022). https://doi.org/10.1109/RTAS54340.2022.00016

[34] Jinja. https://jinja.palletsprojects.com/. Accessed: 23 Jan, 2022

[35] Blass, T., Hamann, A., Lange, R., Ziegenbein, D., Brandenburg, B.B.: Automatic latency management for ros 2: Benefits, challenges, and open problems. In: 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 264–277 (2021). https://doi.org/10.1109/RTAS52030.2021.00029

[36] seL4 Benchmarks. https://sel4.systems/About/Performance/. Accessed: 02 June, 2023

[37] Bini, E., Buttazzo, G.C.: Measuring the performance of schedulability tests. Real-Time Syst. **30**(1–2), 129–154 (2005)

[38] Emberson, P., Stafford, R., Davis, R.I.: Techniques for the synthesis of multiprocessor tasksets. In: WATERS Workshop at the Euromicro Conference on Real-Time Systems, pp. 6–11 (2010)