A Concurrency Framework for Priority-Aware Intercomponent Requests in CAmkES on seL4

Marion Sudvarg
Department of Computer Science & Engineering
Washington University in St. Louis
msudvarg@wustl.edu

Chris Gill
Department of Computer Science & Engineering
Washington University in St. Louis
cdgill@wustl.edu

***Abstract*—Component-based design can encapsulate and isolate 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; and (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. We measure overheads and blocking times for these new features and use existing theory to perform schedulability analysis. Evaluations on both Intel x86 and ARM platforms show that our new library allows CAmkES to provide suitable end-to-end timing for real-time systems.**

I. 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]. *Real-time component frameworks* encapsulate state, computation, and communication, allowing for separation of functional concerns and isolation of resource utilization within components to ensure that timing and other para-functional properties are enforced both locally and end-to-end, based on attributes (e.g. priorities and execution times) and constraints (e.g. deadlines) spanning multiple components.

In particular, CAmkES [2], which targets the seL4 microkernel [3], 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: a single component may execute on behalf of multiple tasks, and 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, we present a new library for priority-aware inter-component requests in CAmkES running atop seL4, with a concurrency framework that allows multiple tasks to execute across shared components, while retaining end-to-end task prioritization.[1] To do so, we provide a system model that 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 the task for which it executes; and (3) implementations of Non-Preemptive Critical Sections, Immediate Priority Ceiling Protocol, and Priority Inheritance Protocol, for components encapsulating exclusive access to a shared resource. Our concurrency framework includes new extensions to the CAmkES specification language, allowing users to easily specify the desired real-time behavior of a component. The framework is implemented entirely in userspace, so it can take advantage of existing formally verified kernel mechanisms in seL4.

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. We 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 II surveys background information and related work. Section III presents our task model and associated schedulability analysis techniques. Section IV details the design and implementation of our library. Section V presents overhead measurements and empirical evaluations of synthetic task sets on two different hardware platforms, and schedulability analyses which incorporate the measured overheads across a broader set of synthetic task sets; these demonstrate the suitability of our library for real-time systems. Section VI concludes the paper, and discusses directions for future work.

The research presented in this paper was supported in part by NSF grants CSR-1814739 and CNS-17653503 and NASA grant 80NSSC21K1741.

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

II. 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. CAmkES is implemented atop **seL4** [3], [4], and allows compile-time specification of component thread priorities [5]. The seL4 microkernel is a widely used [6], [7], lightweight, formally-verified [8], [9] OS kernel with capability-based access control to broker all user-level functionality. All kernel pathway worst-case execution times have been analyzed and bounded [10]. 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 provide a framework that expands CAmkES' support for real-time task sets executing end-to-end across shared components atop the seL4 kernel. We aim to 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 based application software*, thus allowing easy adoption.

The Component-Integrated ACE ORB (**CIAO**) [11] [12] extends and specializes the CORBA Component Model [13] with QoS specifications provided as 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 within the same protection domain, both CAmkES and CIAO can resolve RPCs between them into direct function calls.

The **Patina** API [14] provides priority-aware synchronization primitives for shared resource access in seL4. It includes a mutex service that implements the Priority Inheritance Protocol; threads obtaining a lock must invoke the service via an RPC. In contrast, our framework extends the existing CAmkES design to encapsulate all execution over a shared resource in its own component, allowing each component to manage its own priority-based locking protocols. Patina does not support nested locking; because our framework provides multiple protocols, nested locking can be achieved via nested requests among correctly-configured components.

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 [15]. A similar, efficient thread migration mechanism was later realized for inter-component requests in the **Composite** component-based OS [16]. 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 [17] 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**) [18], implemented in the original L4 microkernel [19], uses scheduling context migration to propagate priorities with intercomponent requests, while also supporting shared resource access protocols (in particular, the Priority Inheritance Protocol [20] and the Immediate Priority Ceiling Protocol [21], [22]). A similar approach [23] was later implemented to support the Priority Inheritance Protocol and bandwidth inheritance [24] in the **NOVA** microhypervisor [25]. These approaches, unlike ours, require the kernel to track the full migration path of the scheduling context. Ours is a userspace framework, and its resource access protocol mechanisms require only a single priority parameter to be passed as part of the request message, eliminating the need for runtime traversal of request chains.

In contrast to previous work, our implementation enables user-level propagation of priorities between components and user-level mechanisms to implement shared resource access protocols. It does not require kernel-level tracking of complete request chains, with priority meta-data instead passed as a single-word parameter as part of the RPC message. 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.

III. System 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 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).

Our target OS platform is the seL4 microkernel [3], which supports fixed-priority preemptive scheduling. The seL4 kernel, compiled with default settings, schedules threads of the same priority in round-robin fashion; however, we also 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 IV and V respectively, we describe our implementation and evaluation of both versions.

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*, giving it an associated thread with priority p_i to run the task. Common

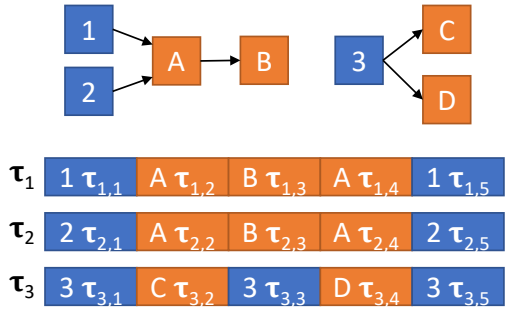


Fig. 1: Tasks $\{\tau_1, \tau_2, \tau_3\}$ originate in active components $\{1, 2, 3\}$ respectively. Components 1 and 2 request common functionality in component A which itself requests B. Component 3 sends a request C, then to one in D. This defines a decomposition of each task into subtasks.

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 and request chains need not be linear: an originating component or CPI may make multiple subsequent requests within the control flow of a single job.

Components hosting one or more shared CPIs are realized in CAmkES by defining them as *passive*. 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. 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.

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

[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.

Priority Propagation. 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_{p_send}, and for replying is C_{p_reply}, with worst case total overhead $C_p = C_{p\_send} + C_{p\_reply}$ (measured in Section V). For a CPI c, we denote the worst-case procedure execution time as $C(c)$. Thus, a task τ_i has total WCET:

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

Each CPI c has a minimum priority $p_{min}(c)$ among tasks for which it executes, and a maximum priority $p_{max}(c)$. A CPI's threads wait on its associated endpoint at $p_{max}(c)$; the blocking time B_i induced on a task τ_i is therefore $\max(C_{p\_send}, C_{p\_reply})$ if there exists a CPI for which $p_{min}(c) < p_i$ and $p_i \leq p_{max}(c)$; otherwise, τ_i 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 [26]:

$$\forall \tau_i \in \Gamma \quad \prod_{P_h > P_i} \left(\frac{C_h}{T_h} + 1 \right) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2 \tag{2}$$

A more pessimistic bound, though one that applies to task sets where multiple tasks may have the same priority, is presented in [20] (Corollary 17), as a generalization of the rate-monotonic utilization bound in [27]:

$$\sum_{\tau_i \in \{\tau_1, \ldots, \tau_n\}} \frac{C_i}{T_i} + \max \left(\left\{ \frac{B_i}{T_i} \right\} \right) \leq n \left(2^{1/n} - 1 \right) \tag{3}$$

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 $C_f = C_{f\_send} + C_{f\_reply}$. 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, $B(c) = \max(C_{p\_send}, C_{p\_reply})$ 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 $(C_f + C(c))$, 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:

$$B_i = max\left(\{B(c) \mid p_{min}(c) < p_i \leq p_{max}(c)\}\right) \tag{4}$$

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:

$$B_i = max\left(\{B(c) \mid p_{min}(c) < p_i\}\right) \tag{5}$$

Task WCETs must account for the different overheads, C_p and C_f, induced by priority propagation and requests to fixed-priority CPIs, 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:

$$C_i = C(c_i) + \sum_{c \in \hat{c}_i} C(c) + \|\hat{c}_{i,p}\| \cdot C_p + \|\hat{c}_{i,f}\| \cdot C_f \qquad (6)$$

Schedulability analysis, using Eqns. 2 or 3, can be performed using these new blocking times and WCETs.

Priority Inheritance Protocol. Our framework also supports CPIs that use the Priority Inheritance Protocol (PIP), as described in Section IV. As we show in Section V, our mechanism induces protocol overhead that depends 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 [20]. 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 for some soft real-time applications.

Our model restricts the nesting of CPIs such that a CPI implementing PIP can only send requests to CPIs with a fixed priority ceiling, i.e., IPCP or NPCS; we do not currently provide a mechanism for nested priority inheritance. The implementations of the other priority protocols we provide (priority propagation, IPCP, and NPCS) do not change the priorities of threads executing request procedures, and therefore can invoke nested requests to CPIs of any type.

IV. Design and Implementation

The CAmkES framework [5] 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 ELF binary packaged with an seL4 kernel image.

Our goal in this work is to elaborate on the CAmkES framework without changes to its underlying parser or 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.

Priority Propagation. We first consider 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 III, 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 (as described later in this section), must execute requests at the priority of the requesting thread, and 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.

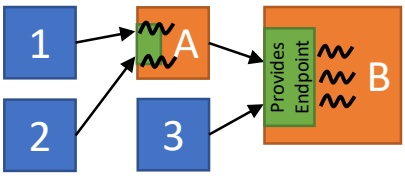


Fig. 2: Passive component A is used by active components 1 and 2 so a pool of 2 threads waits on the underlying endpoint. Component B is used by component 3, and indirectly by 1 and 2 through A, so it has 3 threads.

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. 2. Because CAmkES provides a static specification of CPIs and request connections, this value is straightforward to determine.

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. Upon receiving a request, the handling thread immediately 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. It then executes its procedure, running the subtask at the originating task's priority. After execution is complete, it elevates its priority back to its original waiting priority, replies to the requestor, then goes back 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 III.

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. Fig. 3 shows how nested locking (acquiring a second lock while already holding a lock) can then be achieved through a chain of requests: a CPI encapsulating one lock can make a request to another CPI encapsulating the second lock.

It is straightforward to implement NPCS and IPCP. Both are achieved by tagging an interface with the "fixed" priority protocol attribute and providing a single listening thread to its

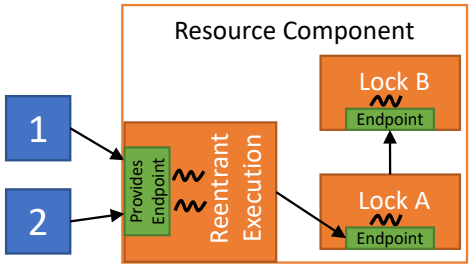


Fig. 3: Active components 1 and 2 send requests to a CPI for Reentrant Execution that is provided by a passive Resource Component, which also provides and uses CPIs for exclusive execution protected by Lock A, and nested locking by Lock B. Lock acquisition ordering is enforced by the defined connections; an acyclic connection digraph is deadlock free.

endpoint, assigning the thread a priority equal to the priority ceiling. This wraps the implementation of these protocols that is provided by both the MCS and non-MCS builds of the seL4 kernel [28]. NPCS is realized under traditional, fixed-priority, preemptive scheduling by assigning a CPI the maximum system priority (255). Once a request is received by the interface, it cannot be preempted. Under round-robin 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. 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.

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 [28], the 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. The IPCP is, as defined in [21], a deadlock avoidant protocol; however, under seL4's priority-based round-robin scheduler, deadlocks can occur.[3] One solution to this is to assign "fixed" CPIs a priority equal to PC+1. However, deadlock could still occur if two CPIs, each using the IPCP, request the other's interface; this could result in a task attempting to acquire a lock it already holds. To guarantee the absence of deadlock one would have to ensure that no cycles exist in the digraph of connections. For a given system specification, CAmkES can generate a digraph representation in the DOT language [29]; this may be used to detect cycles, which alert to possible deadlock.

In this work, we do not implement the original Priority Ceiling Protocol, as described in [20]. The Immediate Priority Ceiling Protocol assigns static priorities to component interfaces; because connections are defined statically in the

[3]Consider a task, τ_1, that needs to acquire locks A then B in a nested fashion. Another task, τ_2, needs to acquire lock B then A. If τ_1 and τ_2 have priorities equal to the priority ceiling, τ_1 could be switched out for τ_2 while holding A. τ_2 could then acquire B and proceed to wait on lock A. At this point, when τ_1 attempts to obtain lock B, deadlock occurs.

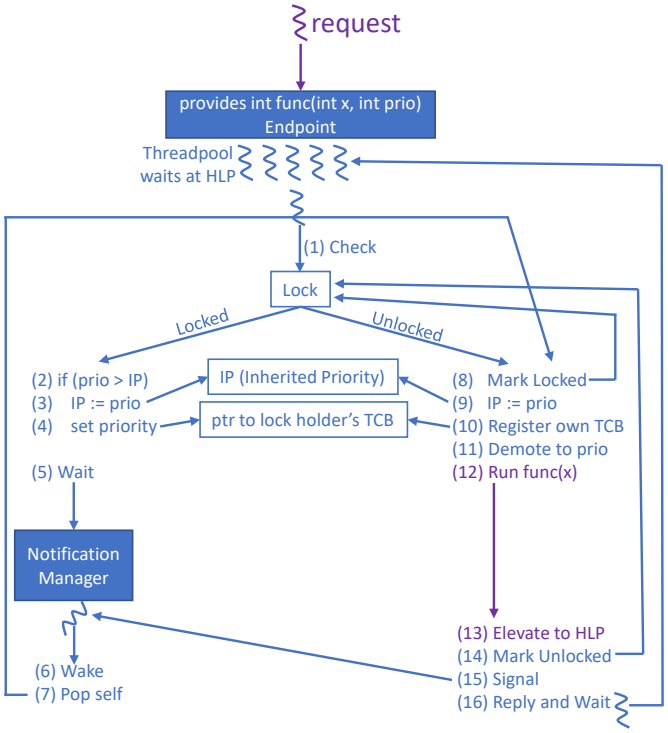


Fig. 4: Priority Inheritance Protocol Implementation

CAmkES specification, these priority ceilings can be computed offline. However, the original Priority Ceiling Protocol requires the tracking of a priority ceiling among all *currently acquired* locks; this would require online global state even among non-interacting components.

Priority Inheritance Protocol. An interface will provide locking with PIP semantics if tagged with the "inherited" priority protocol attribute. For these interfaces, our framework supplies the CPI with three variables: a non-atomic boolean lock, a pointer to the Thread Control Block (TCB) of the lock-holder, and the current inherited priority. These CPIs are again supplied with a pool of threads, the size of which is equal to the number of possible concurrent requests. 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. Under traditional fixed-priority preemptive scheduling, these threads are set to wait on the endpoint at the priority ceiling.

Our implementation of that protocol is illustrated in Fig. 4. When a request arrives, the responding thread (1) checks the lock. If the lock is already held, it proceeds to (2) check the inherited priority variable against its own request priority. 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 TCB. At this point, it (5) waits for a signal indicating that the lock has been freed.

If, however, the lock is unlocked, the thread (8) marks the lock as locked, (9) sets the inherited priority variable to the request priority, (10) sets the TCB pointer to itself, then (11) demotes its priority to the request priority, (12) runs

the interface's procedure code to handle the request. Once complete, it (13) promotes itself back to the priority ceiling, (14) marks the lock as unlocked, (15) signals any threads waiting for the lock, then finally (16) 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. Notification objects are not priority aware if the seL4 kernel is compiled without MCS features: when a signal is received, the kernel wakes the first waiting thread. Thus, a single notification object is insufficient for signalling the threads waiting on a held lock, as the highest priority waiting thread is not guaranteed to be the first to obtain the lock when it becomes available. For the default kernel, we implement a 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, with ties broken by earliest insertion. 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. The request priority is 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 signals the notification object at the head of the heap. The awakened thread (6) returns from the seL4 wait system call; its control flow remains in the notification manager's wait function, which (7) pops its notification object from the head of the priority queue. The thread (8) then proceeds as if it had found the lock available.

In the absence of round-robin scheduling of threads at the same priority, all execution of our protocol (steps 1-11 and 14-16 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 (12-13). If preempted here, it will remain preempted while the responding thread executes steps (1-5). Thus, there can only be two threads from the pool active at any given time: either when there is one thread executing (12-13) and one at the priority ceiling in (1-5), or when the thread holding the lock signals the notification manager, waking another thread. In the latter case, the signaled thread will proceed through (6-8), while the signaling thread proceeds to (16). 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 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. We can solve this problem by setting the thread pool's priority to be PC+1. Now, being a strictly greater priority, the race-free arguments we presented again hold true: execution of our protocol mechanisms occurs at PC+1, and therefore cannot be preempted by new requests. Even with nested locking, a request made by a thread already holding a lock would necessarily be at a priority inherited from a requestor, or from a "fixed" priority CPI thread running at the priority ceiling).

Implementation	Lines of Code
Base Framework C Code	68
Priority Inheritance Protocol C Code	60
Notification Manager C Code	123
CAmkES Macros	7
CAmkES Connector Declarations	100
CAmkES Connector Jinja Templates	30
Total	388

TABLE I: Implementation Lines of Code

Using PC+1 priority, however, can induce undesirable interference with tasks of higher priorities. To avoid this, we adopt a priority-laddering scheme [30] similar to ones used in other prior work (e.g., LynxOS [31]), whereby tasks (and their originating components' threads) are restricted to even priority values (from seL4's 0-254 priority range), and so the thread pools assigned to interfaces specified with "inherited" priority are thereby restricted to the odd values (from 1 to 255). While this reduces the number of effective priority levels from 256 to 128, this still leaves more priorities available than are provided by Linux's fixed-priority scheduling classes [32], which should be sufficient for most realistic task systems.

Implementation and Usability Enhancements. Our userspace implementation targets closed embedded real-time systems running atop the seL4 kernel on unicore or fully-partitioned multicore hardware. We implemented our framework with the goal of staying as true to the CAmkES language and design philosophy as possible, leveraging existing techniques used by the CAmkES framework to provide support for several protocols in only 388 lines of code, as summarized in Table I. It minimizes (to the extent possible) changes needed for existing CAmkES application systems to incorporate its functionality. We now describe how we met this goal and summarize how a developer would use our framework.

CAmkES allows components to be declared with a set of attributes which are compiled into symbols in the component binary, and the user-provided source code for the component can use them as variables. It provides several built-in attributes; e.g., **_priority** sets the priority of an active component's execution thread. If a user manually declares this attribute in a component's definition, its priority becomes available as a variable in the source code. This is necessary for priority introspection without modification to underlying CAmkES or seL4 code because seL4 does not provide a system call for threads to read their current priority level. We define additional attributes which control the number of threads waiting on the endpoint and the CPI's priority protocol; a provided function macro generates all required attributes for a given interface.

CAmkES provides a library of standard connectors to

component interfaces, using the Jinja template engine [33] to generate much of the underlying code (including syscalls) that brokers communication over a given connector type. We define a new class of connectors that inherits much of its functionality from the built-in templates for synchronous IPC. Each connector of this class defines the number of threads bound to the underlying endpoint of the target CPI; a provided function macro creates a connection with the appropriate connector type when given an object macro specifying the number of 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.

Thus the framework, in its current form, requires the user to make few changes to their component source code; changes are minimal and largely necessary to avoid modifying the underlying CAmkES parser and seL4 kernel. Our framework also contains appropriate checks such that incorrectly-specified attributes will cause the application to fail to compile (with appropriate error messages). Its ease of use and its compilation checks to avoid misconfiguration make our framework suitable for developing closed real-time systems.

It is worth noting that the digraph representation generated by the CAmkES parser describes connections from *components* to *CPIs*; this 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 the presence of a cycle does not necessarily imply a call chain loop with deadlock potential, and that our compilation checks cannot detect the presence of a nested request from a CPI implementing PIP to another that does not implement a "fixed" priority protocol. Thus, even if the exported DOT representation of the digraph reveals a cycle, or a possibly invalid nested request from a CPI implementing PIP, we leave it up to the user to decide if the configuration is, indeed, problematic. We defer a modification of the CAmkES parser to provide transitive closure to future work.

V. 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 ran each task set using two configurations of the seL4 kernel: the default configuration, which provides round-robin scheduling for threads of equal priority; and a configuration that schedules threads according to traditional, fixed-priority preemptive semantics, which was achieved by setting the round-robin timeslice to a sufficiently large value. So that we could evaluate the efficacy and overhead of our notification manager, we did not enable the kernel's MCS features. For the Intel platform, we used a system with two Intel Xeon Gold 6130 Skylake processors running at 2.1 GHz (Hyperthreading, Speedstep, and Turboboost disabled) and 32GB of memory. For the ARM platform, we used a Raspberry Pi Model 3 B+, which has a 4-core ARMv8 Cortex-A53 and 1GB of RAM. We disabled the L2 cache, and clocked it to 700 MHz.[4] Despite the 64-bit CPU, seL4 only supports a 32-bit ISA on the Raspberry Pi.[5] We enabled kernel printing and userspace access to the ARM PMU to allow our system to measure and print elapsed cycles.

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 II.

Dividing the maximum cycles measured between two back-to-back cycle counter reads, by the clock speed of each platform, gives a bound on the temporal resolution of our measurements of a little under 25 ns on the Intel Xeon, and 13 ns 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 compare these 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 equates to slightly less than 6 μs on Intel and less than 16 μs on ARM, which is suitably low for task sets running with periods as small as 5 ms. Further, the mean overheads induced by our protocol are within an order of magnitude of benchmarked raw IPC invocation values in the seL4 kernel.[6]

We additionally measure the overhead induced by priority queues realized by different heap sizes within our notification

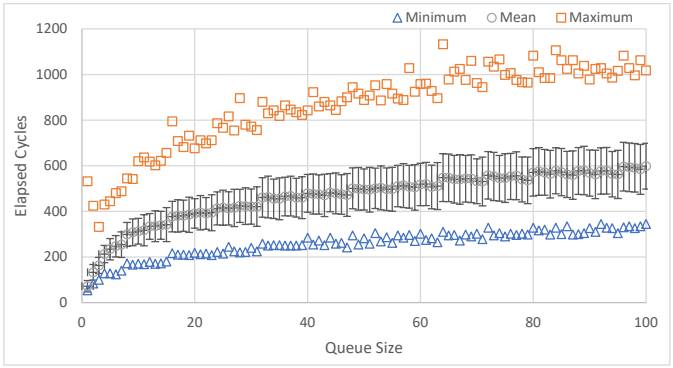
[4]The processor supports a CPU clock speed of 1.4 GHz. However, as noted in [34], this frequency cannot be sustained continuously, and may lead to throttling and instability. To maintain predictability, we boot the Raspberry Pi with a constant 700 MHz CPU clock speed, set the GPU to 250 MHz, and disable throttling. Details can be found at https://www.raspberrypi.com/documentation/computers/config_txt.html

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

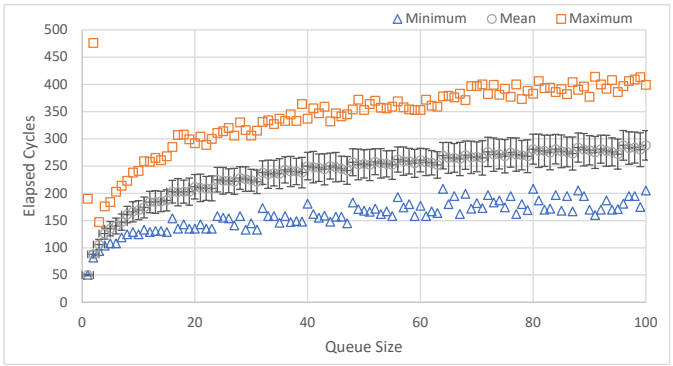
[6]Benchmarked performance numbers listed at https://sel4.systems/About/Performance/ report an average overhead of 382 and 387 cycles, respectively for IPC call and reply between threads in the seL4 kernel without the CAmkES framework. For the ARMv8 platform (in 64-bit mode) the reported values are 395 and 402. Even using PIP, our mean overheads are only about $6.5\times$ this on Intel, and $4.8\times$ this on ARM.

	Intel Xeon Gold 6130				Raspberry Pi Model 3 B+			
	min	max	mean	stddev	min	max	mean	stddev
Read cycle counter	22	52	24	1.5	8	9	8	0.01
Call, built-in	1142	3344	1170	30	469	3168	471	29
Reply, built-in	1108	2266	1129	14	414	1625	414	12
Call, fixed	1028	3664	1187	161	604	2691	689	82
Reply, fixed	1002	2888	1050	66	434	1137	461	42
Call, propagated	2252	6870	2436	156	1628	5740	1755	92
Reply, propagated	2178	4464	2221	30	1433	2191	1518	35
Call, inherited, unlocked	2296	6286	2481	156	1711	5943	1835	85
Call, inherited, locked	2298	6200	2518	179	1683	5060	1827	83
Reply, inherited, unlocked	2222	4258	2274	29	1518	2249	1589	23
Reply, inherited, locked	2224	4010	2275	29	1526	2555	1593	29
Dispatcher Overhead	928	1638	958	13	75	304	75	2

TABLE II: Overheads (in cycles) for Protocol Mechanisms



(a) Intel Xeon Gold 6130



(b) Raspberry Pi Model 3 B+

Fig. 5: Measured Priority Queue Overheads (in cycles)

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. 5, with error bars indicating one standard deviation about the mean. Even the maximum values observed are upper-bounded by 1132 cycles (less than three fifths of a microsecond) on Intel and 476 cycles (less than three fourths of a microsecond) on ARM. This 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 suggests that the overheads for our priority inheritance protocol are dominated by the costs of the system calls it uses.

Empirical Evaluation. 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. 2), all running on a single core. We use the following 3 configurations: (1) The CPIs of components A and B both propagate priorities, (2) the CPIs of A and B both encapsulate exclusive access to shared resources using the IPCP, and (3) A's CPI propagates priorities, while B's CPI (which contains the terminal endpoint in its request chain) encapsulates a lock using PIP. 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 [35], (2) randomly select task periods from a set of harmonic values from 5 ms to 1 second (allowing trials with repeated hyperperiods to be performed efficiently), 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 minus the measured overheads induced by our protocol for requests between components). For each task where a CPI's workload has already been determined (if it's 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, realized in only 40 lines of code by hooking into existing drivers for the board's timer hardware.

An instance of the Dispatcher is created for each task, jobs of which it dispatches via an seL4RPCCall. 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 notification object, as well as the time it takes to determine job completion (aggregated as the last line of Table II) are subtracted from the elapsed time. If the resulting value does not exceed the task's period, the job 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 10^6 iterations on the Intel Xeon, and 10^5 iterations on the slower Raspberry Pi. On the Xeon, the maximum execution time was 9.61×10^5 cycles with a standard deviation of only 2478 cycles (under 1.2 μs). For the Raspberry Pi, the maximum was 8.46×10^5 cycles, with a standard deviation of just 40 cycles (under 60 ns). 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 II) are subtracted from the execution times assigned to each task, before workload iterations are assigned to individual subtasks.

Perhaps unsurprisingly given the harmonic periods of the tasks and the predictable and well-bounded synthetic workload and library overhead times 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.

Schedulability Analysis. To gauge the broader theoretical schedulability beyond the task set configurations evaluated above, we generated synthetic task sets according to same methodology, but with task periods from 5 ms to 1 second selected from the log-uniform distribution described in [36] (Eqn 4), assigning unique rate-monotonic priorities to each task. We also considered NPCS, for which the CPIs of A and B are both assigned the highest system priority. For each configuration, we generated task sets with utilizations ranging from 0.01 to 1.0, with 1000 task sets for each utilization value. We plotted the percentage of task sets that are guaranteed to be schedulable at each total utilization according to the hyperbolic bound without blocking, the hyperbolic bound with blocking (Eqn. 2), and the Liu and Layland bound with blocking (Eqn. 3). Note that for systems of 3 tasks, the Liu and Layland utilization bound without blocking is 0.780. Blocking times are calculated according to the discussion in Section III using the empirical overhead measurements presented in Table II. The schedulability results were similar for the different hardware platforms, and so in this section we only plot results for the Intel Xeon platform, and report any differences seen on the Raspberry Pi in the text.

In Fig. 6, we show the proportion of schedulable tasks according to each bound when both CPIs propagate priority. The overhead and blocking times induced by the underlying protocol are low enough to make the hyperbolic and Liu and Layland schedulability tests with blocking times virtually indistinguishable from equivalent task sets without blocking. Fig. 7 shows the proportions of schedulable tasks when both CPIs are assigned fixed priorities according to IPCP. We observed similar schedulability when component A's CPI simply propagates priorities and component B's CPI inherits priorities according to PIP: for both configurations, the gap between the hyperbolic bounds with and without blocking is narrow; this shows that for both implementations, many task sets that would be schedulable under this sufficiency condition without mutually exclusive critical sections are still schedulable even with the blocking times they induce.

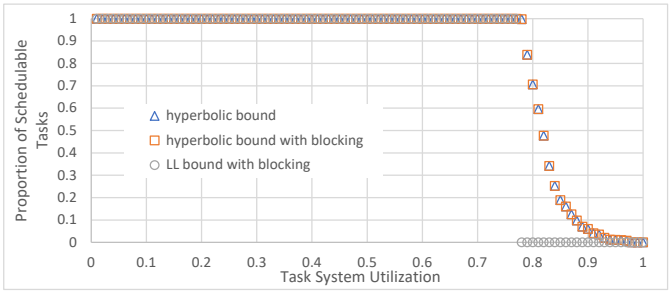


Fig. 6: Schedulability with both CPIs propagating priorities

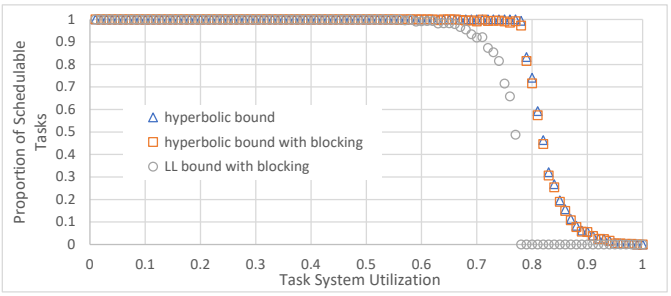


Fig. 7: Schedulability with both CPIs using IPCP

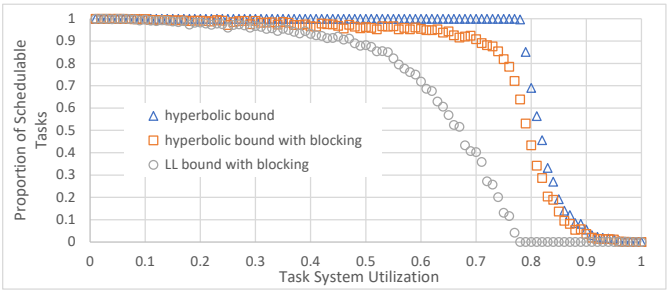


Fig. 8: Schedulability with both CPIs using NPCS

From the results plotted in Fig. 8, we see that when both CPIs are assigned the maximum system priority, such that they implement NPCS, the proportion of schedulable tasks decreases significantly compared to IPCP. This demonstrates the clear disadvantages of the protocol: we begin to see task sets with total utilizations as low as 0.04 that are not guaranteed schedulable under NPCS; under IPCP all generated task sets with total utilization above 0.55 on the Intel Xeon (and above 0.6 on the Raspberry Pi), were guaranteed to be schedulable under the hyperbolic bound with blocking. This justifies our decision to exclude NPCS from evaluation on the

target hardware platforms, as its schedulability guarantees are significantly limited.

VI. Conclusions and Future Work

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. This allows CAmkES to provide suitable end-to-end timing guarantees for real-time systems. As future work, we intend to extend our concurrency framework in several ways: the addition of threads and notification mechanisms to CPI endpoints that use Priority Inheritance Protocol, so that PIP can be extended across multiple CPIs; expansion of our framework to support end-to-end timing guarantees across asynchronous event notifications; modification of the CAmkES parser to allow the user-supplied initialization function to remain the default `__init`; a mechanism supporting transitive closure over request chains, allowing more robust deadlock detection and alerting to invalid component request configurations; and support for request cancellation, as in [23]. We additionally plan to evaluate in comparison with the Patina API [14], and we are considering formal verification of our mechanisms.

References

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

[2] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "Camkes: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687—699, May 2007.

[3] "The sel4 microkernel," https://docs.sel4.systems/projects/sel4/, seL4 Foundation, accessed: 23 Jan, 2022.

[4] K. Elphinstone and G. Heiser, "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*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 133–150.

[5] "Camkes manual," https://docs.sel4.systems/projects/camkes/manual.html, seL4 Foundation, accessed: 23 Jan, 2022.

[6] G. Klein, J. Andronick, M. Fernandez *et al.*, "Formally verified software in the real world," *Commun. ACM*, vol. 61, no. 10, p. 68–77, sep 2018.

[7] G. Heiser, G. Klein, and J. Andronick, "Sel4 in australia: From research to real-world trustworthy systems," *Commun. ACM*, vol. 63, no. 4, p. 72–75, Mar 2020.

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

[9] G. Klein, J. Andronick, K. Elphinstone *et al.*, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, Feb 2014.

[10] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 339–348.

[11] K. Balasubramanian, N. Wang, C. Gill, and D. Schmidt, "Towards composable distributed real-time and embedded software," in *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, Jan 2003.

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

[13] "Corba component model (version 3.0)," https://www.omg.org/spec/CCM/3.0, Object Management Group, oMG Document formal/02-06-65 (Accessed: 24 May, 2001).

[14] S. Jero, J. Furgala, R. Pan *et al.*, "Practical principle of least privilege for secure embedded systems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 1–13.

[15] B. Ford and J. Lepreau, "Evolving mach 3.0 to a migrating thread model," in *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*. San Francisco, CA: USENIX Association, Jan 1994.

[16] G. Parmer, "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)*, 2010, pp. 91–100.

[17] Q. Wang, J. Song, and G. Parmer, "Execution stack management for hard real-time computation in a component-based os," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 78–89.

[18] U. Steinberg, J. Wolter, and H. Hartig, "Fast component interaction for real-time systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, 2005, pp. 89–97.

[19] J. Liedtke, "Improving ipc by kernel design," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, p. 175–188, Dec. 1993.

[20] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[21] G. C. Buttazzo, *Hard Real-Time Computing Systems*, 3rd ed. New York: Springer, 2011.

[22] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, no. 1, p. 67–99, Apr 1991.

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

[24] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1591–1601, 2004.

[25] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 209–222.

[26] E. Bini, G. Buttazzo, and G. Buttazzo, "Rate monotonic analysis: the hyperbolic bound," *IEEE Transactions on Computers*, vol. 52, no. 7, pp. 933–942, 2003.

[27] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, p. 46–61, Jan. 1973.

[28] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: a principled, light-weight operating-system mechanism for managing time," in *ACM EuroSys Conference*, Apr 2018, pp. 1—16.

[29] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.

[30] K. M. Obenland, "The use of posix in real-time systems, assessing its effectiveness and performance," *The MITRE Corporation*, 2000.

[31] "Lynxos — posix real time operating system," https://www.lynx.com/products/lynxos-posix-real-time-operating-system-rtos, Lynx Software Technologies, accessed: 23 Jan, 2022.

[32] "The linux kernel documentation – cfs scheduler," https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, The Linux Kernel Organization, accessed: 23 Jan, 2022.

[33] "Jinja," https://jinja.palletsprojects.com/, The Pallets Projects, accessed: 23 Jan, 2022.

[34] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic latency management for ros 2: Benefits, challenges, and open problems," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 264–277.

[35] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1–2, p. 129–154, May 2005.

[36] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *WATERS workshop at the Euromicro Conference on Real-Time Systems*, Jul. 2010, pp. 6–11.