FlaRe: Efficient Capability Semantics for Timely Processor Access

Manohar Vanga Felipe Cerqueira Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

Anna Lyons Gernot Heiser
NICTA and University of New South Wales (UNSW)

Abstract

In the seL4 microkernel and other capability-based OSs, the right to use resources is managed using abstract capabilities (i.e., tokens). Capability systems for fungible resources such as memory are well understood, but it has proven difficult to integrate time—i.e., real-time scheduling—into this model, since timeliness (i.e., system-wide schedulability) is a property that is inherently difficult to isolate (without introducing prohibitive inefficiencies). This paper presents FlaRe, a temporal capability system that ensures schedulability and temporal isolation by design. Crucially, FlaRe is shown to be efficient, both analytically (FlaRe does not cause utilization loss) and in practice (two prototypes in seL4 and LITMUS$^{\mathrm{RT}}$ were found to incur only negligible overheads). The key technique employed is the flattening of a hierarchical tree of reservations to regular, non-hierarchical EDF scheduling.

1 Introduction

In a *capability-based operating system* [15, 27, 34], the right to use certain resources (*e.g.*, memory pages, I/O ports, IPC ports *etc.*) is managed using abstract *capabilities*, which are unforgeable tokens that can be passed freely among tasks to grant and revoke access to resources in a secure and decentralized way. Notably, this technique is the *de facto* standard approach to resource management in state-of-the-art kernels for high-integrity embedded systems such as Fiasco-OC [24], OKL4 [22], seL4 [23], Nova [36], and Composite OS [31].

Capability-based resource management is attractive because it has numerous conceptual and practical advantages—it is simple, principled, makes access control explicit, and it can be used to enforce strict isolation [17] and thus lends itself well to compositional system design and formal analysis [17, 23]. Overall, it is a mature and proven technique. In particular, capability management is well-understood for multi-unit resources that can be trivially partitioned (*e.g.*, memory and disks), for logical resources that can be duplicated (*e.g.*, IPC ports), and for virtualizable resources that can be shared by coarse-grained multiplexing (*e.g.*, I/O bandwidth and CPU shares in throughput-oriented systems).

Unfortunately, in a real-time systems context, capability-based resource management is still subject to a crippling limitation: the crucial resource *time*, that is, the ability to guarantee *timely processor availability*, has so far eluded an elegant integration into capability systems. To overcome this limitation, we propose FlaRe, an efficient and simple approach to implementing temporal capabilities based on **fla**ttened **re**servations. A key aspect of FlaRe is that it does not require or introduce any new scheduling theory. Rather, we show how predictable processor scheduling can be integrated into a seL4-style capability system on top of a conventional *earliest-deadline first* (EDF) scheduler. The key technique underlying FlaRe is that a *hierarchy* of processor reservations can be *flattened* without utilization loss, while preserving the isolation and compositional properties afforded by hierarchical allocations (a similar approach was recently used in [25]). We explain the challenges involved and our solution in more detail after establishing some needed background.

Capabilities. As mentioned above, capabilities are unforgeable tokens that enable a process to access (or consume) specific resources. Importantly, capabilities realize access control, that is, all resource access is fully mediated and accounted for, and permitted to proceed only if the accessing process holds an appropriate capability.

In the seL4 model [17, 23], the three principal operations on capabilities (aside from use of the associated resource) are the *granting* and *revocation* of existing capabilities, and the derivation of new (*sub-*)*capabilities*. For instance, consider how memory is managed in seL4 [23]. A process is granted one or more capabilities giving it exclusive access to some region(s) of memory. As long as the process holds the capability, it is permitted to use this memory in any way it chooses (via an operation called *retyping*, where untyped memory is first split and then "cast" into typed objects [23]). In particular, it can allocate a part of its memory for use by others by deriving a sub-capability, which it can then grant to another process (*e.g.*, to provide memory for a new child). A capability once granted can later be revoked to reclaim the associated memory, which can recursively trigger further revocations, as a *tree of derived sub-capabilities* (or sub-allocations) may be associated with each capability.

Decentralized resource management allows for strong isolation guarantees, a desirable property for capability based systems. For instance, instead of having a single, global page allocator (*e.g.*, as in Linux), the available memory can be split among subsystems and managed locally within each subsystem according to application-specific needs. This is possible because the management *policy* (*e.g.*, whom to allocate memory to) is cleanly separated from the underlying, mediated *mechanism* (*e.g.*, updating page tables), and because *full isolation* is ensured (allocations in one subsystem do not affect any other subsystem).

Our objective is analogous: to design a scheduling mechanism that allows the decentralized, fully isolated management of *timely* processor access—a scarce, physical resource.

Managing timeliness. Seen as a resource, timeliness is inherently different from interchangeable resources such as memory in that no two processor allocations are alike. That is, memory pages can typically be substituted for one another, and in principle all memory requirements can be met unless the total available memory is exhausted. In contrast, it is not possible to service two tasks that need *immediate* processing at the same time (on the same processor), regardless of how low the overall processor utilization is. This has profound effects, as an allocation in one subsystem (*e.g.*, admission of a task with a tight deadline) can potentially prevent a later allocation in another, supposedly isolated subsystem. That is, if implemented naïvely, use of one temporal capability could render another, "independent" temporal capability invalid, which clearly violates accepted capability semantics.

Fundamentally, "timely access to the processor" is a resource that is not easily metered, budgeted, or accounted for, because the ability to meet *all* deadlines of *all* admitted tasks in *any* subsystem—overall *schedulability*—is fundamentally a global, system-wide property that is not easily amenable to local reasoning and enforcement, and thus full isolation.

Another challenge arises due to the inherently hierarchical nature of decentralized capability systems like that of seL4. As discussed above, to facilitate decentralized resource management, it must be possible for local resource managers to split a "large" temporal capability into several "smaller" capabilities for use by subsystems, which, if applied recursively, can result in deep capability trees. This is a key feature, but it is quite challenging to support from a scheduling point of view. A straw-man solution would be to apply known *hierarchical scheduling* techniques (*e.g.*, such as *periodic resource* abstractions [35]); however, most hierarchical scheduling techniques were not designed with deep hierarchies in mind and could result in considerable utilization loss and prohibitive runtime overheads if applied in this context. Instead, it is desirable to support temporal capabilities on top of regular (*i.e.*, non-hierarchical) schedulers, as such schedulers are already highly optimized.

Contributions. To this end, we present FlaRe, a capability system for the decentralized management of real-time scheduling on uni- and partitioned multiprocessors. The primary contributions of this work are as follows.

1. We show how to support seL4's capability semantics—including full isolation and derivation trees—on top of a regular, non-hierarchical EDF scheduler (Sec. 3).
2. We provide a formal definition of temporal capabilities and the associated operations (Sec. 3.2). To the best of our knowledge, ours is the first such definition inherently amenable to accurate schedulability analysis.
3. FlaRe is efficient from a schedulability perspective: as we show in Sec. 3.5, FlaRe's support of hierarchical sub-capability trees does not cause any utilization loss.
4. FlaRe is efficient in practice: in Secs. 4 and 5, we report on two FlaRe prototypes, one in LITMUS$^{\mathrm{RT}}$ (a Linux-based research platform for multiprocessor real-time systems), and one in seL4 (a true capability-based microkernel for high-integrity embedded systems). In both environments, FlaRe proved to be easy to support and to have no significant impact on scheduling overheads.

We begin with a review of needed background in Sec. 2 and then present our main results in Secs. 3–5. Related work is discussed in Sec. 6; Sec. 7 discusses remaining challenges.

2 Background

We use *partitioned earliest-deadline first* (P-EDF) scheduling as the basis of our design. We chose P-EDF because EDF is optimal with respect to each individual processor, because it has been shown to have low run-time overheads in practice [5], and because accurate schedulability analysis of EDF is available [4]. When discussing analytical aspects of FlaRe, we consider a recurrent real-time workload modeled as a task set $\tau = \{T_1, \cdots, T_n\}$ consisting of *n* sporadic tasks. Each task $T_i = (e_i, p_i, d_i)$ is characterized by a *worst-case per-job execution time* (WCET) e_i, a *minimum inter-arrival separation* (or *period*) p_i and a *relative deadline* d_i. The *utilization* of a task T_i is defined as $u_i = \frac{e_i}{p_i}$; its *density* is defined as $\delta_i = e_i / \min(d_i, p_i)$. The *total utilization* of a set of tasks τ is defined as $U(\tau) = \sum_{T_i \in \tau} u_i$. We focus on sporadic tasks for the sake of simplicity, but note that FlaRe is applicable to arbitrary process-based workloads since each real-time task (or process) is *temporally isolated* using a server abstraction, which we discuss next. Temporal guarantees could thus also be readily derived using other, more expressive formalisms such as event streams [37].

Constant bandwidth server. Abeni and Buttazzo [1] proposed the *constant bandwidth server* (CBS) algorithm, a two-level hierarchical scheduling framework where a set of *CBS reservations*, each encapsulating one or more tasks, is scheduled with a top-level EDF scheduler. The advantage of CBS is that tasks are *temporally isolated*: if a task within a CBS reservation overruns its budget, it does not affect the timing correctness of tasks outside the CBS reservation.

FlaRe uses CBS-based reservations with two restrictions and one extension: **(i)** each CBS reservation contains only a *single task*, **(ii)** the parameters of each CBS reservation, as discussed next, are identical to the parameters of the contained task, and **(iii)** FlaRe extends the original CBS rules [1] to support tasks with constrained deadlines.

A CBS reservation $R_s = (Q_s, P_s, D_s)$ is defined by a *maximum server budget* Q_s, a *server period* P_s and a *relative server deadline* D_s. The density of a CBS reservation is defined as $\delta_s = Q_s / \min(D_s, P_s)$. A CBS is also associated with a *current server budget* C_s and a *current server deadline* D_s^k (for the k'th server deadline), which are both initially zero. In accordance with (i) and (ii) above, we assume that each CBS reservation R_s contains exactly one task $T_i = (e_i, p_i, d_i)$ and that $Q_s = e_i$, $P_s = p_i$, and $D_s = d_i$.

R_s is *active* if T_i has any pending jobs. That is, R_s is active at time t if there exists a job J_i of T_i with arrival time t_a and finish time t_f such that $t_a \leq t < t_f$. If there are no pending jobs, a CBS is inactive or *idle*. If a new job of task T_i is released at time t_a and R_s is already active, then the job is queued. If a job arrives when R_s is idle, two cases are possible; if $C_s \geq (D_s^k - t_a)\delta_s$, the server assigns a new deadline $D_s^k = t_a + D_s$ and the server budget is replenished to the maximum $C_s = e_s$; otherwise, it executes the job immediately (and its parameters remain unchanged).

R_s's budget C_s is consumed by one unit for every unit of time a job of the contained task T_i executes. Further, we assume suspension-oblivious reservations that treat job suspensions as regular execution time, that is, jobs that are suspended but incomplete continue to consume budget. If the current server budget C_s reaches zero while R_s is active, it is immediately replenished ($C_s = Q_s$) and a new server deadline is assigned: $D_s^{k+1} = D_s^k + P_s$. It should be noted that assigning a single task $T_i = (e_i, p_i, d_i)$ to a reservation R_s with parameters $Q_s = e_i$, $P_s = p_i$ and $D_s = d_i$ ensures that the deadline will not be postponed *unless the task overruns its budget*. In the case of an overrun, R_s continues to execute at a lower priority (*i.e.*, with a postponed deadline) without impacting the timeliness of other tasks.

A key feature of the original CBS [1] is that it is work-conserving, that is, R_s always remains eligible for execution while active, as the budget C_s is immediately replenished (and the deadline postponed) when it reaches zero. Following the terminology established by Rajkumar *et al.* [33] in their work on resource kernels, we refer to this version of CBS as *soft CBS*. A variant of CBS, called *hard CBS* [11], cuts a task off from further supply until the current server deadline has been reached. That is, in the case of hard CBS, the budget is only recharged (and the deadline postponed) after the current server deadline has passed. Intuitively, soft reservations are appropriate for sporadic real-time tasks that should complete as soon as possible in case of an overrun, whereas hard CBS is more appropriate for encapsulating potentially backlogged applications that require rate limiting.

Strictly speaking, a collection of CBS reservations is *schedulable* if each server is guaranteed to be able to consume its entire budget before its current deadline. Since we assume that each task is encapsulated in a CBS reservation with matching parameters, this is equivalent to stating that all tasks that do not exceed their provisioned budget will meet all deadlines; we therefore use the terms CBS reservation and task interchangeably in the remainder of this paper.

Schedulability analysis. In the seminal work on uniprocessor EDF schedulability analysis [4], Baruah *et al.* gave an exact *processor demand test*. They showed that a task set τ of n sporadic tasks is schedulable on a uniprocessor under EDF if and only if $U(\tau) \leq 1$ and

$$\forall t \geq 0 \quad \sum_{i=1}^{n} DBF(T_i, t) \leq t, \qquad (2.1)$$

where $DBF(T_i, t) = \max(0, \lfloor \frac{t-d_i}{p_i} \rfloor + 1) \cdot e_i$ is the *demand bound function* of task T_i [4]. Intuitively, $DBF(T_i, t)$ upper-bounds the cumulative execution requirement of all jobs of T_i that both arrive and have a deadline within any contiguous interval of length t. Crucially, Baruah *et al.* established a bound on t for which Eq. (2.1) must be checked [4]. This interval was further reduced in subsequent work (*e.g.*, [38]).

Approximation of DBFs. Working with precise DBFs can be computationally expensive. Albers and Slomka proposed a method of approximating DBFs [2], which we adopt in our implementation to improve runtime efficiency and reduce memory overheads. The basic idea behind their scheme is that after a certain number of steps *k*, the DBF of a task is bounded by a straight line with a slope equal to the task's utilization. The k-steps approximation of the DBF of a task $T_i = (e_i, p_i, d_i)$ is denoted $\overline{DBF}(T_i, t, k)$, where $\overline{DBF}(T_i, t, k) = DBF(T_i, t)$ if $t < d_i + (k-1)p_i$, and $\overline{DBF}(T_i, t, k) = u_i(t - d_i) + e_i$ otherwise. Importantly, for any $t \geq 0$, $\overline{DBF}(T_i, t, k) \geq DBF(T_i, t)$.

In summary, Abeni and Buttazzo's CBS [1] and Baruah *et al.*'s processor demand analysis [4] constitutes FlaRe's analytical foundation, as we describe next.

3 FlaRe Capabilities

In a nutshell, FlaRe realizes seL4's *hierarchical* isolation and sub-capability derivation semantics [17, 23]—with regard to timely processor access—on top of *flat* (*i.e.*, non-hierarchical) P-EDF. Full isolation is ensured using per-task CBS reservations [1] based on Baruah *et al.*'s exact schedulability condition [4]. We begin with an overview of FlaRe and discuss the intuition behind its design in Sec. 3.1, and then formally define the semantics of temporal capabilities (Sec. 3.2) and describe FlaRe's runtime support (Sec. 3.3).

3.1 Intuition and Overview

First of all, what does it mean to *use* a temporal capability? In FlaRe, using a temporal capability means creating a new CBS reservation on a particular processor. The limited resource controlled by FlaRe's temporal capabilities is thus processor demand, as defined by Baruah *et al.* [4].

Intuitively, a process that holds a temporal capability may thus "become" a real-time task by first creating a CBS reservation with arbitrary parameters, but subject to the limits on maximum processor demand represented by its capability, and by then attaching itself to the reservation. That is, analogous to seL4's memory capabilities, a process may "cast" (a part of) its unallocated (*i.e.*, "untyped") processor demand into a specific (*i.e.*, "typed") CBS reservation with concrete parameters. Similarly to seL4's memory capability semantics, which ensure that the total amount of typed memory never exceeds the total memory supply, FlaRe maintains Eq. (2.1) on each processor, that is, FlaRe ensures that the total demand never exceeds the total processor capacity.

The most challenging operation to support is the splitting of capabilities (*i.e.*, the derivation of sub-capabilities), which is central to seL4's hierarchical resource management semantics and essential for truly decentralized resource allocation. In particular, it must be possible to split a temporal capability into two or more sub-capabilities and/or CBS reservations such that no processor capacity is lost. Returning to the memory analogy, when a memory allocation is split, we expect the sum of the parts to equal the whole. Similarly, dividing a temporal capability should not cause processor capacity to be lost to analytical inefficiencies, which, from a scheduling point of view, is difficult to ensure if done naïvely.

Structurally, a temporal capability in FlaRe consists of one or more *CPU allocations*, each of which represents "a fraction of demand" on a particular processor. Analytically, each CPU allocation is described using an *allowance function* and an *allowed total utilization*.

Each such CPU allocation in turn contains a set of CBS reservations. Since FlaRe is based on P-EDF, each reservation must be created on a specific processor (*i.e.*, each CBS reservation belongs to exactly one CPU allocation); however, since a temporal capability can consist of more than one CPU allocation, it is possible to represent several processors worth of "processor capacity" with a single temporal capability.

Hierarchy is supported by allowing each CPU allocation to contain sub-allocations, subject to the constraint that the sub-allocations may together not represent more processor demand than the parent allocation does. More precisely, the total processor demand due to **(i)** all sub-allocations and **(ii)** the set of client CBS reservations may not exceed the total processor demand represented by the parent CPU allocation. Sub-capabilities can then be derived by creating one or more sub-allocations and "bundling" them into a new capability.

On each processor, there is a *root allocation* that anchors the tree of (local) CPU allocations. The root allocations are created during system initialization and each represent the entire capacity of one processor. By enforcing that the CPU allocation hierarchy on each processor never represents more processor demand than available in the root allocation—and by requiring that real-time tasks hold temporal capabilities—FlaRe ensures that the set of all CBS reservations is always schedulable. This can be seen as safely *flattening* the decentralized, hierarchical resource allocation *policy* onto a non-hierarchical reservation *mechanism*, namely CBS. In effect, FlaRe consists of two *views* of the system: an *allocation view*, which is the tree of CPU allocations, for the purpose of mediating resource access (*i.e.*, admission control), and a *flattened view*, which is simply the set of all admitted reservations, for the purpose of online scheduling.

To stay with the memory analogy, the relationship between the allocation view and the flattened view is akin to memory capabilities and the frame table in seL4: the capability derivation tree in seL4 is inherently hierarchical and can represent arbitrary allocation policies, and describes how parts of the "flat" physical memory are mapped.

We formalize these ideas next, discuss how the two views are managed at runtime in Sec. 3.3, and then provide examples of common use cases in Sec. 3.4 thereafter.

3.2 The Semantics of Temporal Capabilities in FlaRe

In FlaRe, a *temporal capability* CAP_C is a set of l_C *CPU allocations*, where $l_C \geq 1$.[1] Each CPU allocation $A_k = (AF_k(t), a_k, \tau_k, \Delta_k)$ reserves a fraction of processor capacity on a single processor and is characterized by an *allowance function* $AF_k(t)$, an *allowed total utilization* a_k, a set of *sub-allocations* Δ_k, and a set of CBS-based *client reservations* τ_k. τ_k is a set of CBS-based reservations, each containing a single task. Each CBS-based reservation can be either a hard or soft CBS instance, depending on the requirements of the contained task. A CPU allocation also contains a set of sub-allocations Δ_k. Each (sub-)allocation can thus be understood as the root of a (sub-)tree of CPU allocations.

Δ_k and τ_k together represent the workload encapsulated by A_k and contribute to the cumulative demand of A_k, which is constrained by its allowance function $AF_k(t)$. Specifically, as made explicit in Invariants 1 and 2 below, the allowance function $AF_k(t)$ defines the upper bound on permitted cumulative demand, and the allowed total utilization a_k defines the upper bound on the permitted total utilization.

$AF_k(t)$ is a user-specified parameter and given as a function of interval length (analogous to DBFs). In theory, $AF_k(t)$ may be any non-negative, monotonically increasing function; in practice, it is represented as a piece-wise linear function akin to Albers and Slomka's approximation method [2] using a finite number of points and a slope of a_k.

The parameters $AF_k(t)$ and a_k are specified when the CPU allocation is created (and remain unchanged thereafter); the sets Δ_k and τ_k are initially empty. Crucially, any additions to Δ_k and τ_k are governed by two invariants, which are essential to FlaRe's analytical guarantees.

Invariant 1. *For any allocation $A_k = (AF_k, a_k, \tau_k, \Delta_k)$, the total cumulative utilization of its sub-allocations and reservations never exceeds its allowed total utilization:*

$$a_k \geq \sum_{A_j \in \Delta_k} a_j + \sum_{T_i \in \tau_k} u_i.$$

Invariant 2. *For any allocation $A_k = (AF_k, a_k, \tau_k, \Delta_k)$, the cumulative processor demand of its sub-allocations and reservations never exceeds the upper bound specified by its allowance function, that is, for any $t \geq 0$:*

$$AF_k(t) \geq \sum_{A_j \in \Delta_k} AF_j(t) + \sum_{T_i \in \tau_k} DBF(T_i, t).$$

Based on Invariants 1 and 2, we define four basic operations on a given $A_k = (AF_k, a_k, \tau_k, \Delta_k)$ for the addition and removal of sub-allocations and CBS reservations.

A1 Adding a new CBS-based reservation. A new reservation for a given task $T_i = (e_i, p_i, d_i)$ may be added to τ_k if and only if the addition of T_i to τ_k does not violate Invariants 1 and 2.

[1] Note that l_C has no relation to m, the number of processors in the system. We permit temporal capabilities to contain multiple CPU allocations on the same processor, as there is no analytical requirement to prohibit this.

A2 Adding a new sub-allocation. A new sub-allocation $A_j = (AF_j(t), a_j, \emptyset, \emptyset)$, where $AF_j(t)$ and a_j are specified as parameters to the operation, may be added to Δ_k if and only if the addition does not violate Invariants 1 and 2. Arbitrary trees of hieararchical allocations can be created with this operation.

A3 Removing an existing reservation. Removal of an existing CBS reservation for a task $T_i \in \tau_k$ can always be initiated as it only reduces demand and thus cannot violate Invariants 1 and 2. Removing a CBS reservation from a CPU allocation demotes the contained task to best-effort status. This removal is delayed until the current server deadline $d_{s,k}$ has expired, to avoid corner cases in which a task quickly switches between best-effort and real-time status (*i.e.*, it should not be possible to replenish a server's budget ahead of time by destroying and recreating it; see, *e.g.*, [6] for a more nuanced discussion of such *reweighting* issues).

A4 Removing an existing sub-allocation. An existing sub-allocation A_j can be removed from Δ_k if and only if $\Delta_j = \emptyset$ and $\tau_j = \emptyset$. The pre-condition requires the recursive removal of all reservations and sub-allocations of A_j prior to removal of A_j (which is analogous to the recursive revocation of memory in seL4 [23]).

Capability derivation. As discussed in Sec. 3.1, in FlaRe, new capabilities can be derived by creating one or more sub-allocations (using operation **A2**) and "bundling" them into a new capability. To define this precisely, we define the *aggregate allocation set* of an allocation A_k, denoted as $\Delta_{agg}(A_k)$, as the set of all allocations recursively reachable from A_k; formally $\Delta_{agg}(A_k) = \Delta_k \cup \left(\bigcup_{A_j \in \Delta_k} \Delta_{agg}(A_j)\right)$. The derivation of a new capability is then defined as follows: given a capability CAP_C, a new capability CAP_D may be derived from CAP_C such that

$$CAP_D = \{A_y \mid A_y \in \Delta_{agg}(A_x) \wedge A_x \in CAP_C\}. \quad (3.1)$$

In other words, the allocations that constitute the derived capability CAP_D must stem from (sub-)allocations reachable from the original capability CAP_C. Derived capabilities are thus never more permissive than the capability from which they were derived. The operation also permits a single CPU allocation to be a part of multiple capabilities. That is, there can be two distinct capabilities, each containing the same CPU allocation within them, which allows CPU allocations to be selectively shared among multiple processes (just like memory pages may be shared among multiple processes).

Having established a precise definition of the resource allocation rules in FlaRe, we next describe how they are combined to ensure schedulability at runtime.

3.3 The Scheduling of Flattened Reservation Trees

FlaRe enforces Invariants 1 and 2 for all CPU allocations. To guarantee schedulability, FlaRe further ensures that all CPU allocations stem from known *root allocations*, which are special CPU allocations that represent the total capacity available on one processor. There is one such allocation per processor, and the root allocation of the νth processor is denoted as $A_\nu^{root} = (AF_\nu^{root}, a_\nu^{root}, \tau_\nu^{root}, \Delta_\nu^{root})$, where $AF_\nu(t) = t$ and $a_\nu = 1$. The m root allocations are created during system initialization. Importantly, these are the *only* CPU allocations created from scratch; all other CPU allocations must be derived directly or indirectly from a root allocation using operation **A2**.

Allocation view. The m root allocations thus form a set of m trees from which all CPU allocations are reachable; we refer to this set of hierarchies as the *allocation view* of the system. The allocation view can be understood as a form of metadata that provides insight into the current hierarchical reservation structure. Importantly, the allocation view is used only during task admission, but *not* for online scheduling.

Flattened view. Instead, the hierarchical allocation view is flattened by combining the reservations from all CPU allocations at any level into a single task set (on each processor). To this end, we define the *aggregate reservation set* of an allocation A_k as $\tau_{agg}(A_k) = \tau_k \cup \left(\bigcup_{A_j \in \Delta_k} \tau_{agg}(A_j)\right)$. The *flattened view* on the νth processor is then simply $\tau_{agg}(A_\nu^{root})$, the aggregate reservation set of the root allocation.

In FlaRe, the P-EDF scheduler schedules the flattened view. That is, on each of the m processors in the system, it schedules the set of aggregated CBS-based reservations $\tau_{agg}(A_v^{root})$ corresponding to the local root allocation A_v^{root}. By explicitly splitting the allocation view from the flattened view, the allocation *policy* (which is reflected in the allocation view) and the scheduling *mechanism* (which only considers the flattened view) are cleanly separated in FlaRe.

Finally, the *root capability* $CAP_0 = \{A_1^{root}, \ldots, A_m^{root}\}$ describes the entirety of the system's processing capacity. CAP_0 is created during the boot sequence; all other capabilities must stem directly or indirectly from the root capability.

Together with Invariants 1 and 2, these simple allocation and scheduling rules guarantee that the flattened sets of reservation are indeed schedulable under P-EDF. We establish this property formally in Sec. 3.5, after first providing some examples to illustrate the utility of FlaRe.

3.4 Example Uses of FlaRe: Choosing a_k and $AF_k(t)$

When creating a new sub-allocation A_k, the parameters a_k and $AF_k(t)$ are specified by the invoking process. This naturally poses the question—what are appropriate values?

The allowed total utilization a_k is the simpler of the two parameters. It simply provides a hard upper limit on the maximum utilization of all tasks and sub-allocations. Thus, if the intention is to limit a subsystem to reserve at most 20% of a processor, the choice $a_k = 0.2$ would be appropriate.

Determining an appropriate allowance function $AF_k(t)$ is slightly more involved. In the simplest case, if the intention is to simply reserve a fraction of processor capacity for best-effort or implicit-deadline tasks, it is appropriate to choose $AF_k(t) = a_k \cdot t$. (This also makes for an apt default value in concrete implementations of FlaRe.)

If, however, real-time tasks with constrained deadlines are to be supported, it is advisable to choose a more explicit $AF_k(t)$. For instance, suppose that a subsystem initialization task is to create a capability for a set τ_s of (hard) real-time tasks that comprise the subsystem. Assuming the requirements of all tasks in τ_s are known, a well-chosen allowance function would be $AF_k(t) = \sum_{T_i \in \tau_s} DBF(T_i, t)$. In other words, for known task sets, FlaRe can be configured to yield exactly matching allocations, which helps to avoid pessimism, as we establish in Theorem 2 below.

3.5 Temporal Correctness in FlaRe

A key feature of FlaRe is that, by enforcing Invariants 1 and 2 *locally* within each allocation, the schedulability of *all* reservations comprising the flattened view is always guaranteed. This is accomplished without imposing any additional utilization loss (besides that inherent in any partitioned scheduling approach). That is, FlaRe's hierarchical capability system does not impose any additional limitations on the workloads that can be supported. We begin by showing that Invariants 1 and 2 imply schedulability.

Lemma 1. Let $\tau_{agg}(A_k)$ denote the aggregate reservation set of a CPU allocation A_k. Then $U(\tau_{agg}(A_k)) \leq a_k$.

Proof. Let h denote the height of the sub-allocation tree rooted at A_k. We prove the lemma by induction over h.

Base case: if $h = 0$ (*i.e.*, if A_k is a leaf), then the claim follows from Constraint 1 since $\Delta_k = \emptyset$ in leaf nodes.

Induction step: suppose the claim holds for $h - 1$. Then:

$$\begin{aligned}
U(\tau_{agg}(A_k)) &= U(\tau_k) + \sum_{A_j \in \Delta_k} U(\tau_{agg}(A_j)) \\
&\qquad \{\text{ induction hypothesis }\} \\
&\leq U(\tau_k) + \sum_{A_j \in \Delta_k} a_j \\
&\qquad \{\text{ definition of } U(\tau_k) \text{ }\} \\
&= \sum_{T_j \in \tau_k} u_j + \sum_{A_j \in \Delta_k} a_j \\
&\qquad \{\text{ Invariant 1 }\} \\
&\leq a_k.
\end{aligned}$$

□

The same relationship applies to allowance functions, which we summarize as follows.

Lemma 2. Let $\tau_{agg}(A_k)$ denote the aggregate reservation set of a CPU allocation A_k. Then $AF_k(t) \geq \sum_{T_j \in \tau_{agg}(A_k)} DBF(T_j, t)$ for all $t \geq 0$.

Proof. Omitted; follows analogously to Lemma 1. □

The flattened view is thus always schedulable in FlaRe.

Theorem 1. Let $\tau_{agg}(A_\nu^{root})$ denote the aggregated set of reservations scheduled on the νth processor with root allocation A_ν^{root}. Under FlaRe, the flattened set of reservations $\tau_{agg}(A_\nu^{root})$ is schedulable under EDF.

Proof. From Lemma 1, it follows that

$$U(\tau_{agg}(A_\nu^{root})) \leq a_\nu^{root} = 1. \qquad (3.2)$$

From Lemma 2, and since $AF_\nu^{root}(t) = t$, it follows that

$$\forall t: \quad AF_\nu^{root}(t) = t \geq \sum_{T_j \in \tau_{agg}(A_\nu^{root})} DBF(T_j, t). \quad (3.3)$$

Recall that FlaRe employs P-EDF, and recall from Sec. 2 that Eqs. (3.2) and (3.3) together are Baruah *et al.*'s [4] exact schedulability test for uniprocessor EDF, see Eq. (2.1). The aggregated set of reservations $\tau_{agg}(A_\nu^{root})$ is thus schedulable under EDF, which FlaRe employs on each processor. □

Theorem 1 shows that FlaRe's capability system ensures temporal correctness. The second key feature of FlaRe is that there is no need to over-provision resources.

Theorem 2. FlaRe can be configured to have arbitrarily small utilization loss w.r.t. each processor.

Proof. By design, for any desired CPU allocation A_k, it is possible to compute each $AF_k(t)$ and a_k such that $a_k = \sum_{A_j \in \Delta_k} a_j + \sum_{T_j \in \tau_k} u_j$, and, for all t, $AF_k(t) = \sum_{A_j \in \Delta_k} AF_j(t) + \sum_{T_j \in \tau_k} DBF(T_j, t)$ Thus, any feasible task set (*i.e.*, any task set that passes Baruah *et al.*'s schedulability condition [4]) can be arranged into a hierarchy of CPU allocations of arbitrary structure and depth. □

To summarize, FlaRe provides a capability system for timely processor access that has several attractive conceptual properties: it is compliant with the accepted notion of hierarchical isolation and sub-capabilities in seL4 [23], its admission control rules (Invariants 1 and 2) require only capability-local information (*i.e.*, $AF_k(t)$, a_k, Δ_k, and τ_k) to ensure system-wide schedulability, it reduces to simple, non-hierarchical online scheduling, and finally, it does not impose any additional utilization loss beyond that inherent in partitioned scheduling (which is highly attractive from a scheduling overhead point of view [5]).

However, to be a viable solution in practice, it must be possible to implement FlaRe in a real OS with little difficulty and low overheads. To evaluate this aspect, we implemented prototypes of FlaRe in two (quite different) real-time OSs.

4 Proof of Concept: FlaRe in LITMUS$^{\text{RT}}$

To evaluate FlaRe in a multiprocessor context, we implemented it in LITMUS$^{\text{RT}}$ [10], a real-time extension of the Linux kernel. Although LITMUS$^{\text{RT}}$ is not a capability-based operating system, it is a suitable proving ground for demonstrating the simplicity and practicality of FlaRe in a non-trivial OS environment; LITMUS$^{\text{RT}}$ further provides some of the required scheduling infrastructure. In particular, we used the P-EDF scheduler and benchmarking framework available in LITMUS$^{\text{RT}}$ to implement and evaluate FlaRe. We begin by describing key choices in the implementation of FlaRe in LITMUS$^{\text{RT}}$ (hereafter *FLRT* for brevity) and then discuss the practicality of FlaRe based on experimental results from FLRT. We also discuss the notion of *implicit capabilities*, an extension of the basic capability-based model that simplifies the integration of FlaRe into legacy systems.

4.1 Implementation

We implemented FlaRe in the latest version of LITMUS$^{\mathrm{RT}}$, based on Linux 3.0. When the system boots up, a root allocation is created for each of the processors in the system along with the root capability, which is granted to all privileged (root) processes (discussed in more detail in Sec. 4.3 below). FLRT provides a system call interface for user programs which consists of **(i)** system calls that implement operations **A2** and **A4** to allow user-space processes to create arbitrary CPU allocation hierarchies, **(ii)** a system call for deriving a new capability from an existing one (which enforces Eq. (3.1)), and **(iii)** a system call for granting a capability to a particular process. LITMUS$^{\mathrm{RT}}$ already provided system calls for real-time tasks to join and leave the system (*i.e.*, to create and destroy reservations), which we retrofitted to support operations **A1** and **A3**. We also modified the existing P-EDF scheduling plugin in LITMUS$^{\mathrm{RT}}$ to ensure that real-time processes are admitted into the system only if they hold a capability with sufficient unallocated processor demand. Further, FLRT implements both hard and soft reservations, and user processes are free to choose either type.

A key component of FlaRe is the admission control test that is part of operations **A1** and **A2**. The overhead of this test, which checks Invariants 1 and 2, depends primarily on the efficiency of adding and comparing (approximated) DBFs and allowance functions. To control the runtime and memory costs of admission control, the number of points k used to approximate these functions is a configurable, userspace-specified parameter in FLRT.

Like most OSs, Linux (and consequently LITMUS$^{\mathrm{RT}}$) does not support floating-point operations in the kernel. In FLRT, we implemented all operations on allowance functions and demand bound functions using integer arithmetic. The use of integer arithmetic causes a loss of precision in the calculations due to rounding errors. Care must be taken to ensure that these errors do not result in an unschedulable system during task admission. In FLRT, we deal with this by rounding values pessimistically, which consequently introduces some utilization loss. However, this utilization loss is negligible in practice as LITMUS$^{\mathrm{RT}}$ uses nanoseconds to express all temporal parameters; rounding errors are thus minuscule in relation to typical task parameter magnitudes.

FLRT supports both hard and soft CBS reservations. At first, it may seem like implementing these requires significant scheduling machinery. However, due to the constraints FlaRe places on the use of CBS reservations, their implementation is trivial. Recall that in FlaRe, each CBS reservation contains exactly one task and has parameters identical to the contained task. Implementing per-task CBS reservations in FLRT thus consists of initializing a per-job timer which detects when a job has exhausted its budget (using LITMUS$^{\mathrm{RT}}$'s precise budget enforcement for real-time tasks). At this point, if the job is still incomplete (that is, it has overrun its budget), the hard and soft CBS rules (as described in Sec. 2) are applied to extend the deadline and replenish the budget appropriately.

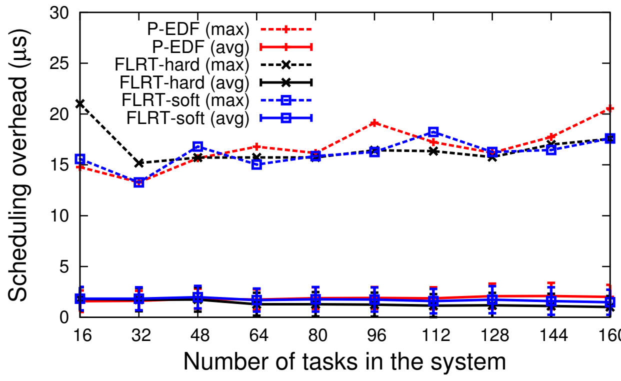

Figure 1: Worst- and average-case scheduling overheads due to regular P-EDF without FlaRe, FlaRe with soft CBS reservations, and FlaRe with hard CBS reservations, as measured in LITMUS$^{\mathrm{RT}}$ on a 16-core Intel Xeon X7550 2.0 GHz platform.

In summary, FLRT is a feature-complete prototype of FlaRe and thus sufficient for providing realistic overheads.

4.2 Overheads

There are two main runtime overheads in FLRT to be considered: scheduling overheads arising from the use of per-task CBS-based reservations, and task admission overheads caused by the introduction of the admission control test. While the former is a recurring, per-job overhead, the latter is a one-time cost that arises only once during task admission. Any increases in scheduling overhead are thus highly undesirable, whereas admission costs are more flexible. For this reason, FlaRe is intentionally designed such that most costs are incurred during task admission.

Scheduling overhead. Due to the simplified implementation of CBS, we expected the scheduling overhead from the use of per-task CBS containers to be negligible. To confirm this, we measured the overhead of scheduling a varying number of tasks under three different configurations of P-EDF: regular P-EDF without FlaRe, and FlaRe with hard and soft CBS reservations (the experimental setup is described in Appendix A). Fig. 1 depicts the observed worst- and average-case scheduling overheads. As is apparent, any scheduling overhead added due to per-task CBS reservations is negligible—no increase in overheads is apparent, that is, the overheads are entirely subsumed by measurement noise. This shows that per-task CBS reservations are indeed an efficient method of ensuring isolation among tasks.

Admission and memory overhead. Due to space constraints, the results from our evaluation of task admission and memory overheads in FlaRe can be found in Appendix A. To summarize our results, both admission and memory overheads vary depending on k, the number of points used to represent allowance functions and DBFs. We checked values of k up to 20, considerably greater than the minimum value we estimated as sufficient to avoid utilization loss (see Appendix A), and the overheads are within an acceptable range for both admission time and memory usage. Admission overhead was on the order of tens of milliseconds for $k = 20$ (recall that this is a one-time cost and that task admission is a rare event in comparison to job scheduling). The memory overhead of storing 3,000 allocations with $k = 20$ was around 2.5 MiB of memory, a low cost relative to the memory typically available in modern multicore platforms.

4.3 Implicit Capabilities

In a normal capability-based system, processes must explicitly grant a capability to each of their child processes during creation. However, this is cumbersome to support in legacy systems such as Linux, as modifying the entire software stack to use the new FlaRe interfaces would be burdensome.

To overcome this issue, we introduced the notion of *implicit capabilities*. The idea behind implicit capabilities is that a process may be admitted into the system as a real-time task, even in the absence of an explicitly granted capability, if another process, termed the *implicit capability supplier*, is assigned to grant its capability to the process "on demand."

For example, with a simple change to X11, all GUI tasks in the system are assigned the X11 server process as their implicit capability supplier. Further, by default, the parent process becomes the implicit capability supplier when a new child is forked. Thus, without further setup and without modification of the shell, a real-time task launched from a shell running within a terminal emulator would transitively have access to X11's temporal capability.

As another example, all privileged (root) processes in the system are assigned the "init" process, a process granted the root capability at system initialization, as their implicit capability supplier. Thus, all privileged processes are implicitly granted the root capability in FLRT.

Using implicit capabilities, FlaRe can be integrated into complex legacy systems with few changes to existing software. Next, we discuss a prototype implementation of FlaRe within a truly capability-based microkernel.

5 Proof of Concept: FlaRe in seL4

To show that FlaRe is practical in a capability-based OS, we developed a prototype in seL4. Multiprocessor features in seL4 are still experimental, so we only deal with the uniprocessor case here. In true microkernel fashion, the seL4 implementation of FlaRe separates policy and mechanisms, by implementing the former at user-level and only the latter inside the kernel. Specifically, the kernel provides scheduling capabilities and enforces their semantics, while the admission procedure and data (DBFs) are contained in a trusted user-level process. Note that the FlaRe implementation is a prototype that is not yet formally verified.

5.1 seL4 Scheduling

Original seL4 uses a priority-based round-robin scheduler with 256 priorities. We introduce a new highest priority level where threads are scheduled according to EDF. To implement the EDF scheduler, we added two binary heaps: a release queue of tasks waiting for job release and a ready queue of released tasks waiting to execute. To prevent interrupts aimed at the round-robin scheduler from disrupting EDF-priority tasks, we disabled timer ticks for non-EDF tasks (making non-EDF priorities non-preemptive), allowing us to use a single 64-bit timer for the EDF implementation (a comprehensive implementation would restore preemption). We have also added a CBS implementation to the kernel.

5.2 Scheduling Capabilities

We introduce two new types of capabilities: scheduling capabilities, and a scheduling control capability. Standard seL4 capability operations apply: these capabilities can be moved, copied, revoked, recycled and deleted.

Scheduling capabilities acts as access tokens to in-kernel *scheduling contexts*, analogous to CPU allocations in FlaRe. Scheduling contexts are either *bindable* or not, and specify hard or soft CBS. Bindable scheduling contexts can be bound to a single thread, which will then be scheduled by EDF using the parameters of that context. If a bindable context becomes unbindable, a bound thread will revert to its previous priority.

The *scheduling control capability* is granted to the initial task in a seL4 system via the startup protocol and gives the possessor complete control over the EDF scheduler. Three operations are possible when invoking this capability: Configure (set parameters in a scheduling context), Bind (bind a scheduling context to a thread) and SetBindable (change the bindable bit in a scheduling context). A process with access to the scheduling control capability can distribute scheduling capabilities throughout the system and revoke at will.

Like other seL4 objects, any user process can create a scheduling context (with all parameters set to 0) by re-typing an *untyped* object (representing a right to use memory). However, scheduling contexts cannot be configured or bound to threads without invoking the scheduling control capability.

Unlike the FLRT, seL4 provides no in-kernel admission test, this policy is implemented in a user-level server (the holder of the scheduling-control capability), similarly to the memory-management policy. This server is trusted, as it can configure the system in an overload state, causing deadlines to be missed, which is again analogous to servers managing memory, which can corrupt the managed (sub-)system.

5.3 Admission Control

For our FlaRe implementation on seL4, task admission is performed by a trusted *time manager*, which possesses the scheduling control capability and runs at the highest non-EDF priority. This ensures that admission control cannot interfere with existing EDF tasks, separates policy from mechanism and supports the use of floating-point arithmetic.

The time manager implements the hierarchical allocation tree and provides split and revoke to clients. Capabilities in the tree are assigned a unique ID by which client and server can identify them (the root has ID 0 and a 100% CPU

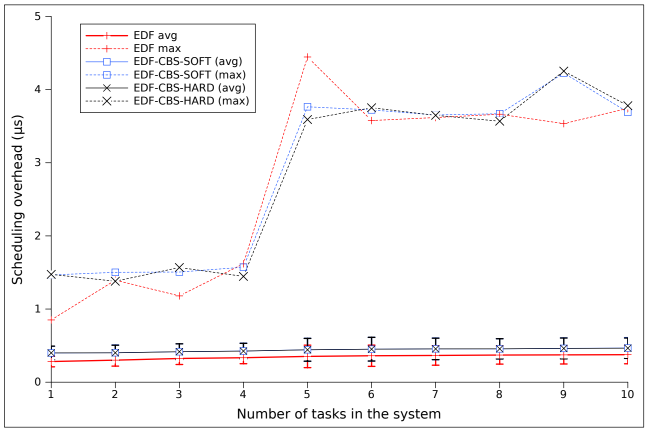

Figure 2: Scheduling overheads in seL4, as measured on a single core of a SabreLite 1 GHz ARM Cortex-A9 MPCore platform.

allocation). During a split operation, the server performs the admission test; if successful, it returns the ID of a capability to a bindable scheduling context. The manager also sets the parent capability as unbindable, to preserve the integrity of the tree. To implement revoke, the manager keeps copies of all capabilities, so they can be revoked without contacting clients. Appendix B illustrates a sample system.

5.4 Evaluation

Admission tests in seL4 use the same code as in FLRT, so their cost is the same (modulo the very small inter-process communication cost and performance differences in the hardware); we did not evaluate them again. Instead, we focus on scheduling overhead, comparing the seL4 EDF scheduler with and without CBS (see Appendix C for full details). Scheduling in seL4 is about four times faster than in FLRT (despite the slower processor), which increases the relative impact of FlaRe. Figure 2 shows that the CBS implementation adds a small (˜0.09μs) *constant* overhead to a scheduling decision an overhead to small to be observed in FLRT. We observe no difference between CBS types.

6 Related Work

Capability-based access control is a classic concept that dates back to Dennis and Van Horn [15], and which has long been employed in OSs [20, 29, 34]. (A detailed historical overview of capability-based access control can be found in [27].) As discussed in Secs. 1 and 3, our work on temporal capabilities is explicitly inspired by seL4 [23] and its resource-management model [17]. In particular, the primary objective in the design of FlaRe was the *efficient* support of *arbitrary* derivation trees as they arise naturally from seL4's sub-capability semantics. In FlaRe, this goal is achieved by *flattening* all allocation hierarchies prior to scheduling.

A similar notion of flattening was recently applied by Lackorzynski *et al*. [25] to the problem of real-time virtualization (and implemented in Fiasco.OC [24]). Lackorzynski *et al*. observed that the hierarchical scheduling of VMs (where the host scheduler selects a VM to execute, and then the VM-internal scheduler selects the actual task to dispatch) can lead to inefficiencies. This is because a single, per-VM host-level processor reservation can be insufficient to express the varied timing requirements of encapsulated workloads [25], especially in a multi-VM scenario. By exporting sufficient scheduling metadata from the virtual machines to the host—that is, by *flattening* the hierarchical scheduler—the utilization loss from over-provisioning was reduced [25]. FlaRe also uses the flattening idea, but significantly expands on it by providing *formal, provably efficient semantics for sub-capabilities* that ensure that the system remains schedulable at all times. (Lackorzynski *et al.* [25] do not provide details on how they integrated flattening into Fiasco.OC's capability system [24], nor do they consider how to enforce schedulability in a non-centralized and efficient manner, which are the two cornerstones of FlaRe.) Another difference is that their system uses fixed-priority scheduling, whereas FlaRe derives its efficiency w.r.t. utilization loss from EDF. We consider Lackorzynski *et al.*'s work to be complementary to FlaRe as their proposal is both very much in alignment in its motivation and also ideally suited to hosting VMs on top of a FlaRe-equipped hypervisor.

While Lackorzynski *et al.* identified benefits in avoiding hierarchical scheduling, there are nonetheless cases where it is explicitly desired as a means of integrating and isolating (legacy) subsystems; consequently, there has been considerable research into hierarchical scheduling in recent years (*e.g.*, see [12–14, 16, 19, 26, 35]). Compared to FlaRe, hierarchical scheduling differs in two ways. First, FlaRe has been designed from the ground up for capability-based systems, and therefore supports *arbitrarily deep allocation hierarchies without utilization loss*. This is important as capabilities can be derived any number of times. On the other hand, FlaRe cannot support application-specific schedulers, as a primary motivation for hierarchical scheduling is the integration of multiple (legacy) applications, *each with its own dedicated local scheduler*. Second, as FlaRe is based on a single-level scheduler, it has a (slight) advantage in runtime efficiency. In contrast, a hierarchical scheduler traverses the entire hierarchy until a suitable leaf node is found, which adds to scheduling overheads (especially in deep hierarchies). While FlaRe technically still uses "hierarchical scheduling" in the form of CBS reservations [1], there are no scheduling decisions to be made at the second "level" since FlaRe uses *per-task* reservations. Thus, while hierarchical scheduling could conceivably be repurposed to support temporal capabilities, FlaRe is simpler and more efficient in practice.

A key technique in FlaRe is the use of allowance functions to characterize the permissible aggregate demand of entire subtrees; Baruah and Fisher [3] similarly applied DBFs to entire hierarchical component trees in an analysis of a component system hosted on top of global EDF.

Finally, Parmer and West [30] recently presented HiRes, a complete system architecture for the hierarchical management of processor reservations, memory, and I/O resources.

Similar to the seL4 model [23] adopted in FlaRe, HiRes allows the granting and revocation of resources and can thus be considered to be a capability system. In terms of scope, Parmer and West's work [30] is complementary to ours: whereas FlaRe does not address I/O and memory resources (although seL4 [23] does), HiRes does not address schedulability or the semantics of hierarchical processor reservations. It would thus be interesting to combine both approaches.

7 Conclusion, Limitations, and Future Work

We presented FlaRe, a simple and efficient temporal capability system for secure, safe, and decentralized management of timely processor access. The defining characteristic of FlaRe is that it supports arbitrary sub-capability derivation trees without utilization loss, which it accomplishes by flattening them prior to scheduling. The simplicity of this approach translates into low runtime overheads, as evidenced by the two implementations in LITMUS$^{\mathrm{RT}}$ and seL4.

FlaRe provides isolated temporal capabilities based on sound scheduling theory, but it is certainly not the only piece of the isolation and predictability puzzle. In particular, various other factors such as cache interference (*e.g.*, [28]) and bus contention (*e.g.*, [21]) need to be taken into account. These issues, however, are orthogonal to the problem addressed by FlaRe, and proper temporal capabilities are still useful even if some level of hardware interference remains.

We did not consider the issue of overhead accounting in detail. In addition to cache interference, scheduling and interrupt overheads must be accounted for. Scheduling overheads can be dealt with by inflating execution budgets [7]; in FlaRe, such overheads are implicitly accounted for by charging all preemption costs to the budget of the preempting reservation.

Hard interrupts associated with job releases are more challenging, and similar issues are encountered in hierarchical scheduling as well, as observed by Phan *et al.* [32]. A pragmatic solution in LITMUS$^{\mathrm{RT}}$ is to use a dedicated core for interrupt handling to shield real-time tasks [7]. A more sophisticated solution, left to future work, similar to Phan *et al.*'s approach, associates each task with a separate CPU allocation describing the arrival characteristics of task-related interrupts. We also seek to introduce support for locking and/or IPC primitives that guarantee temporal isolation across reservations despite budget overruns [8].

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS '98*, 1998.

[2] K. Albers and S. Slomka. An event stream driven approximation for the analysis of real-time systems. In *ECRTS '04*, 2004.

[3] S.K. Baruah and N. Fisher. Component-based design in multiprocessor real-time systems. In *ICESS '09*, 2009.

[4] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS '90*, 1990.

[5] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *RTSS '10*, 2010.

[6] A. Block. *Adaptive Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill, 2008.

[7] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[8] B. Brandenburg. Virtually exclusive resources. Technical Report MPI-SWS-2012-005, MPI-SWS, May 2012.

[9] B.B. Brandenburg and J.H. Anderson. Feather-trace: A light-weight event tracing toolkit. In *OSPERT '07*, 2007.

[10] B.B. Brandenburg, A.D. Block, J.M. Calandrino, U. Devi, H. Leontyev, and J.H. Anderson. LITMUS RT: A status report, 2007.

[11] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency (Series in Computer Science)*. Plenum Publishing Co., 2005.

[12] S. Chen, L.T.X. Phan, J. Lee, I. Lee, and O. Sokolsky. Removing abstraction overhead in the composition of hierarchical real-time systems. In *RTAS '11*, 2011.

[13] R.I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *RTSS '05*, 2005.

[14] Z. Deng and J.W.S. Liu. Scheduling real-time applications in an open environment. In *RTSS '97*, 1997.

[15] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9:143–155, 1966.

[16] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using edp resource models. In *RTSS '07*, 2007.

[17] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In *VSTTE*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, October 2008.

[18] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *WATERS 2010*, 2010.

[19] X. Feng and A.K. Mok. A model of hierarchical real-time virtual resources. In *RTSS '02*, 2002.

[20] N. Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*

[21] Y. Heechul, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS'13*, 2013.

[22] G. Heiser. Hypervisors for consumer electronics. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5. IEEE, 2009.

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an os kernel. In *SOSP '09*, 2009.

[24] A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in L4. In *IIES '09*, 2009.

[25] Adam Lackorzynski, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *EMSOFT*, pages 93–102, Tampere, Finland, October 2012.

[26] H. Leontyev and J.H. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In *ECRTS '08*, 2008.

[27] M.H. Levy. *Capability-Based Computer Systems*. 1984.

[28] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS'13*, 2013.

[29] S.J. Mullender and A.S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4), 1986.

[30] G. Parmer and R. West. HiRes: A system for predictable hierarchical resource management. In *RTAS '11*, 2011.

[31] G.A. Parmer. *Composite: A component-based operating system for predictable and dependable computing*. PhD thesis, Boston University, 2010.

[32] L.T.X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky. Overhead-aware compositional analysis of real-time systems. In *RTAS'13*, 2013.

[33] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Readings in multimedia computing and networking. chapter Resource kernels: a resource-centric approach to real-time and multimedia systems. 2001.

[34] J.S. Shapiro, J.M. Smith, and D.J. Farber. EROS: a fast capability system. In *SOSP '99*, 1999.

[35] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS '03*, 2003.

[36] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

[37] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, 2000.

[38] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Trans. Comput.*, 58(9), September 2009.

A FLRT: Extended Evaluation

In this appendix, we present an extended evaluation of FLRT. We first describe the experimental setup for obtaining scheduling overheads and then present results of experiments measuring the memory and admission overheads in FLRT. Finally, we present results from an experiment measuring the utilization loss under FlaRe as a function of k, the number of points used to represent allowance functions and DBFs.

A.1 Scheduling Overheads – Experimental Setup

To measure scheduling overheads, we generated 25 task sets with $n \in [16, 32, \ldots, 160]$ tasks (for a total of 250 samples), using the *randfixedsum* algorithm [18], with periods ranging from 10-100ms (log-uniform distribution) and implicit deadlines. The First-Fit DU partitioning heuristic was used to partition tasks. We used Feather-Trace [9] to trace the execution time of scheduling decisions under regular P-EDF, FLRT with soft CBS reservations, and FLRT with hard CBS reservations (each task set was executed for one minute in each configuration). We measured these overheads on a 16-core Intel Xeon X7550 2.0GHz platform with 1TiB RAM, after disabling CPU features that affect predictability (*e.g.* Turbo Boost and Hyper-threading). The entire experiment lasted 12.5 hours and produced 42 GiB of binary data that was used to extract the scheduling overheads.

A.2 Admission Overheads

In order to understand the overhead of enforcing admission control in FlaRe, we must consider the factors that contribute to the time complexity of the admission decision. Recall that when a new task is being admitted, the introduced demand must not violate the schedulability of the system, subject to the allowance function and the maximum utilization of the allocation to which it is being added (see Invariants 1 and 2). Thus the admission cost is a function of the time taken to add and compare allowance functions and DBFs. The time complexity of these operations depends on the number of points k used in their representation.

We evaluated the runtime costs of adding a new sub-allocation to an allocation with different numbers of existing sub-allocations (respectively 50, 100 and 150), while varying k between 2 and 20. As illustrated in Fig. 3, the time taken to perform the admission control is reasonable (≈55ms when there are 150 existing sub-allocations and $k = 20$.). This can be trivially improved by caching the final summed function, so that future operations avoid the recalculation. This would make the overhead independent of the number of sub-allocations in a particular allocation and only depend on k, which can be tuned for the desired accuracy.

A.3 Memory Overheads

In FLRT, each CPU allocation object contains an allowance function and DBFs for tasks are stored in memory to improve the speed of the admission test. Storing these functions is the primary source of memory overhead. We analyzed these costs by varying the total number of allocations in the system (500, 1500 and 3000) and the parameter k (as in the admission overheads evaluation). From Fig. 4, it can be seen that the cost of storing 3000 allocations, each with 20 points in their respective functions, is around 2.6MiB, a low amount given the memory in modern multicore machines.

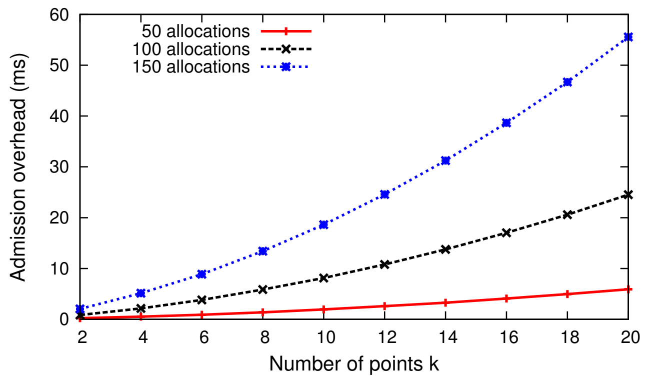

Figure 3: Admission overhead for adding a new allocation into an existing one, plotted as a function of the number of points k in each allowance function and demand bound function. The graph shows three curves based on how many sub-allocations are previously present in the allocation, as this affects overheads. The evaluation was performed on 16 Intel Xeon X7550 2.0GHz cores.

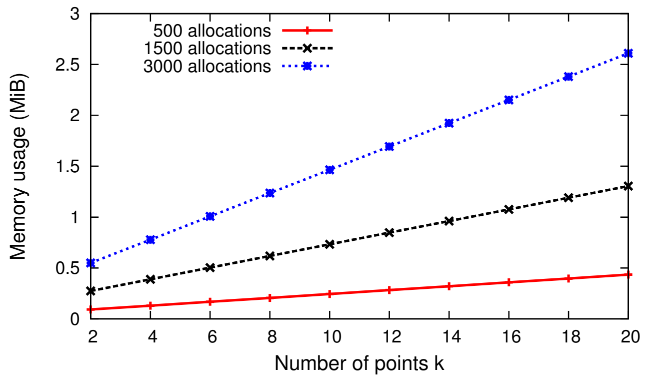

Figure 4: Memory overhead of storing 500, 1500 and 3000 allocations as a function of k, the number of points used to represent allowance functions and demand bound functions.

A.4 Utilization Loss

Since FlaRe approximates allowance functions and DBFs, it is important to understand the tradeoff between k, the number of points used to represent these functions, and the utilization loss in the system. To measure this, we checked the schedulability of 19,200 task sets, generated by the *randfixedsum* algorithm with utilizations ranging from 0 to 1 (in increments of 0.1). The number of tasks per task set varied in increments of 5 between 5 and 150. The periods of the tasks were log-uniform between 10-100ms, and deadlines were chosen uniformly within the first half of the interval between the execution cost and the period.

Fig. 5 shows the percentage of task sets schedulable un-

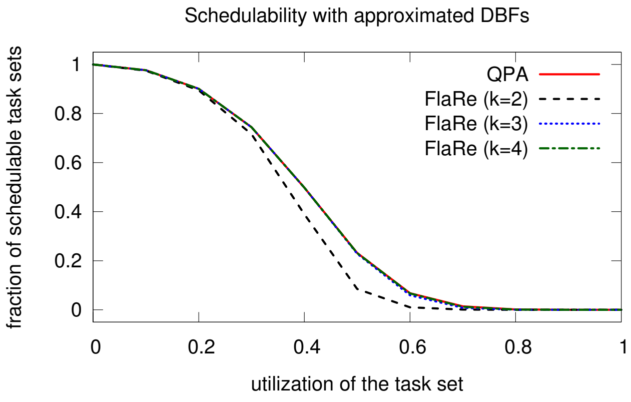

Figure 5: This graph shows the utilization loss in FlaRe for different values of the parameter k. It shows the percentage of schedulable task sets as a function of task set utilization. The results from FlaRe were compared against the QPA algorithm.

der FlaRe with different values of k, as a function of task set utilization. We compared our results with the QPA algorithm [38], which is an exact test. As can be seen from Fig. 5, the fraction of schedulable task sets converges with the results of the QPA algorithm for $k > 2$. This shows that, in the evaluated scenario, the utilization loss arising from the use of approximated allowance functions and DBFs is negligible even for small values of k. This result, along with the previous discussion on overheads, makes a strong case for the minimal utilization loss of FlaRe, and for its practicality in terms of admission time and memory usage.

B seL4: Implementation Example

This section supports the seL4 implementation presented in Section 5. Figure 6 shows the layout of a system just after system initialization. The initial process started with capabilities to untyped memory and the scheduling control capability. It creates two threads: the client and the time manager, and transfers the scheduling control capability to the time manager, before exiting. The client makes a split call to the time manager, who performs the admission test, retypes some memory to create a capability to a scheduling context. It then configures the scheduling context, and makes a copy which is transferred to the client. The client binds this capability to a newly created thread and resumes it, creating the real-time task.

C seL4: Extended Evaluation

In this appendix, we present more details about the evaluation of seL4. First, we describe the experimental setup for obtaining scheduling overheads. And then we present the results of analyzing the standard scheduling overheads of seL4 without EDF.

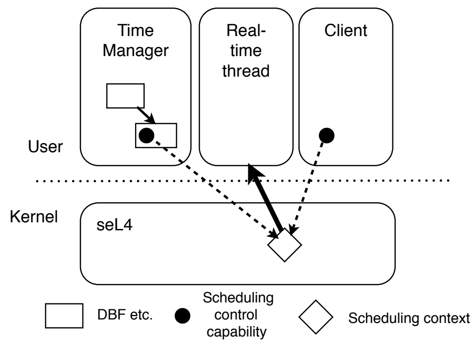

Figure 6: A sample seL4 system with FlaRe.

C.1 Scheduling Overheads – Experimental Setup

The scheduling overhead experiment involved adding tracepoints to the kernel before and after a scheduling decision. These tracepoints consisted of reading the cycle count register of the ARM performance monitor unit, located on the coprocessor. The impact of measurement was quantified to be 4 cycles as coprocessor access is fast on the Cortex A9.

Our test environment consists of a SabreLite development board with a quad-core ARM Cortex A9 MPCore. As indicated, we ran tests with only one core enabled, the number of tasks per core as the same is in the FLRT benchmarks. The difference between the x-axis scales in Figure 1 and Figure 2 reflect this. We generated the same number of task sets with the same parameters used to evaluate FLRT and ran them on seL4 to obtain overhead numbers.

C.2 Original seL4 Scheduler

The original seL4 scheduler was not implemented for performance and conducted a linear search of 256 priorities, which is inefficient for all but the highest priority. We have measured the overhead of an optimized (unverified) version of the scheduler for comparison with the EDF implementation in seL4. The optimized version uses a 2-level bitmap to find the highest priority thread. We measured the cost of this lookup 1000 times and found it to be on average 0.27μs, with a standard deviation of 0.1μs – we put the variation down to branch mispredictions, as the variance goes down as the benchmark progresses.