Scheduling-Context Capabilities

A Principled, Light-Weight Operating-System Mechanism for Managing Time

Anna Lyons
Data61, CSIRO
UNSW Sydney
Anna.Lyons@data61.csiro.au

Kent McLeod
Data61, CSIRO
Sydney
Kent.McLeod@data61.csiro.au

Hesham Almatary
Data61, CSIRO
UNSW Sydney
heshamelmatary@gmail.com

Gernot Heiser
Data61, CSIRO
UNSW Sydney
Gernot.Heiser@data61.csiro.au

ABSTRACT

Mixed-criticality systems (MCS) combine real-time components of different levels of criticality – i.e. severity of failure – on the same processor, in order to obtain good resource utilisation. They must be able to guarantee deadlines of highly-critical threads without any dependence on less-critical threads. This requires strong temporal isolation, similar to the spatial isolation that is traditionally provided by operating systems, without unnecessary loss of processor utilisation. We present a model that uses *scheduling contexts* as first-class objects to represent time, and integrates seamlessly with the capability-based protection model of the seL4 microkernel. We show that the model comes with minimal overhead, and supports implementation of arbitrary scheduling policies as well as criticality switches at user level.

CCS CONCEPTS

• **Computer systems organization** → **Real-time operating systems**; *Embedded software*; • **Software and its engineering** → **Scheduling**;

KEYWORDS

Access Control, Capabilities, Mixed-Criticality Systems, Microkernels, seL4

ACM Reference Format:
Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3190508.3190539

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190539

1 INTRODUCTION

Emerging cyber-physical systems have conflicting requirements that challenge operating system (OS) design: they require mutually distrusting components of different criticalities to share resources, and must degrade gracefully in the face of failure. For example, an autonomous aerial vehicle (AAV) has multiple inputs to its flight-control algorithm: object detection, to avoid flying into obstacles, and mission control (navigation), to get to the desired destination. Clearly the object detection is more critical than navigation, as failure of the former can be catastrophic, while the latter would only result in a non-ideal route. Yet the two subsystems must cooperate, accessing and modifying shared data, thus cannot be fully isolated.

The isolation requirements for such systems go beyond the spatial isolation and sharing that is well-supported by present OSes; cyber-physical systems have strict temporal requirements, for which OS support is lacking. Specifically, existing OSes conflate the notions of priority, time sensitivity and criticality (i.e. importance).

For example, the AAV's flight-control algorithm may only run ten times a second with 2 ms to complete, while network and bus drivers should be serviced at a much higher rate and should be able to preempt flight control as long as there is sufficient time for flight control to execute. This implies that networking runs at higher priority than flight control. However, while occasional packet loss is tolerable, a failure of the flight control algorithm to complete on time is not. This requires a notion of temporal protection, even in the presence of sharing, that is not available in present OSes.

The AAV is an example of a *mixed-criticality system* (MCS), a notion that originates in avionics and its need to reduce space, weight and power (SWaP), by consolidating growing functionality onto a smaller number of physical processors. In safety-critical systems, *criticality* is a well-defined concept, indicating the severity of failure. Standards require that the safe operation of a particular component must not depend on any less-critical components [ARINC 2012]; in other words, a critical component must not trust a less-critical one to behave correctly.

While MCS are becoming the norm in avionics, this is presently in a very restricted form: the system is orthogonally partitioned spatially and temporally, and partitions are scheduled with fixed time slices [ARINC 2012]. This limits integration and cross-partition communication, and implies long interrupt latencies and poor resource utilisation.

High system utilisation is essential for addressing SWaP challenges, and high responsiveness is important for much of a system's desirable functionality. But high utilisation is a challenge in critical real-time systems, whose core *integrity property* is that deadlines must always be met, meaning that there must always be time to let threads execute their full *worst-case execution time* (WCET). This may be orders of magnitude larger than the typical execution time, and computation of safe WCET bounds for non-trivial software tends to be highly pessimistic [Wilhelm et al. 2008]. Consequently, most of the time the highly-critical components leave plenty of slack, which should be usable by less critical components. In terms of schedulability analysis, this constitutes an *overcommitted* system, where not everything is guaranteed to be schedulable. In case of actual overload, the system must guarantee sufficient time to the critical components at the expense of the less critical ones.

These challenges are not unique to avionics: top-end cars exceeded 70 processors ten years ago [Broy et al. 2007]; with the robust packaging and wiring required for vehicle electronics, the SWaP problem is obvious, and will be magnified by the move to more autonomous operation. Other classes of cyber-physical systems, such as smart factories, will experience similar challenges.

The upshot is that MCS require OS support for a form of *temporal isolation*, where (lower criticality) high-priority threads can preempt (highly critical) threads, but cannot monopolise the processor; the OS must limit their time consumption, in order to guarantee sufficient time to the critical threads. This isolation must be enforced even when components of different criticality share data or services, e.g. flight control must be able to access a consistent waypoint state while remote operators and collision avoidance of the AAV concurrently update waypoints.

Our goal is to design and implement an OS that provides the right mechanisms for efficiently supporting MCS, and reason about their safety. For spatial isolation, capabilities [Dennis and Van Horn 1966] have become the accepted mechanism that allows reasoning about access control at a fine granularity, and are therefore used in security-oriented designs, such as KeyKOS [Bomberger et al. 1992], EROS [Shapiro et al. 1999] and seL4 [Klein et al. 2009]. We similarly aim to provide a capability-based model for access to CPU time, without forcing a particular scheduling model on the system.

Specifically, we make the following contributions:

- A capability system for time that imposes low overhead;
- a notion of CPU-time budgets that is compatible with fast IPC implementations traditionally used in high-performance microkernels, and is compatible with established real-time resource-sharing policies;
- an exploration of implementation in the non-preemptible seL4 microkernel and its interaction with the kernel's model of user-level management of kernel memory, which is a critical enabler of strong spatial isolation;
- an implementation of a dynamic-priority system in a user-level scheduler on top of the kernel's static priorities, demonstrating that arbitrary scheduling policies can be implemented at low overhead;
- a demonstration that, without further kernel mechanisms, a notion of criticality and criticality switches can be efficiently implemented at user level.

2 BACKGROUND

2.1 Real time theory basics

Standard real-time (RT) theory uses the term "task", which effectively maps onto the OS concept of a thread. To avoid confusion we will stick with the OS terminology.

RT scheduling theory generally assumes *periodically executing threads*, which maps well onto typical control systems, where different activities execute periodically albeit with different periods. Aperiodic ("sporadic", i.e. interrupt-driven) threads are incorporated in such a model by requiring a defined *minimum arrival time*, corresponding to a maximum interrupt rate, which is used as the thread's period for the schedulability analysis. RT threads have a *deadline* by which a computation must be finished, which may be an *explicit* (specified), or *implicit* (end of the period).

Priorities can be static (fixed) or dynamic. The optimal static scheme is *rate-monotonic priority assignment* (RMPA), where the rate (the inverse of the period) becomes the priority; RMPA can guarantee schedulability of arbitrary thread sets as long as the total utilisation is below a limit that is asymptotically $\log 2 \approx 69\%$ [Liu and Layland 1973]. The overall optimal scheme is *earliest-deadline first* (EDF), a dynamic-priority scheme which always picks the thread whose deadline is closest; EDF can schedule any set of threads whose total utilisation does not exceed 100%.

A system whose total utilisation exceeds the schedulability threshold is *overcommitted*, and can become *overloaded*, i.e. unable to meet all deadlines. By definition of priority, the deadlines missed will be of the lowest-priority threads. For RMPA this affects a well-defined subset (the threads with the longest period). EDF victimises whatever thread happens to be furthest from its deadline, leading to apparently random threads being victimised. This causes EDF's reputation as being poorly behaved under overload, the main reason EDF is unpopular in industry [Buttazzo 2005].

An established way of limiting the amount of CPU time a thread can consume are *resource reservations* [Mercer et al. 1993; Oikawa and Rajkumar 1998], which can be implemented as periodic [Lehoczky et al. 1987] or sporadic servers [Sprunt et al. 1989] for RMPA and *constant-bandwidth servers* (CBS) [Abeni and Buttazzo 2004] for EDF.

Reservations present a guarantee by the kernel that the reserved bandwidth is available. This means that they do not support overcommitting. Also, the kernel must perform a schedulability analysis as *admission control* whenever a reservation is created. Schedulability tests can be complicated and frequently constitute a trade-off between cost of the test and achievable utilisation.

2.2 Criticality, time-sensitivity and trust

Criticality and time-sensitivity are attributes of system components that are often conflated. However, they are different, and, in general, not aligned.

The *criticality* of a component reflects its importance to the overall system mission. Criticality may reflect the impact of failure [ARINC 2012] or the utility of a function. An MCS should degrade gracefully, with components of lower criticality (which we will refer to as LOW components) suffering degradation before higher criticality (HIGH) components.

Time sensitivity refers to how important it is for a thread to get access to the processor at a particular time. For best-effort activities, time is fungible, in that only the amount of time allocated is of relevance. In contrast, for a hard RT component, time is largely unfungible, in that the allocation has no value if it occurs after the deadline; soft RT components are in between.

Most OSes only have a single parameter for controlling both attributes, *priority*. For example, Linux aligns priority with timeliness, by running priority-based threads at the highest overall priorities, followed by deadline-scheduled threads with CBS reservations and best-effort below. Similarly, RBED [Brandt et al. 2003], as used by Barrelfish [Peter et al. 2010], prioritises threads based on time sensitivity. In NOVA [Steinberg and Kauer 2010] and Fiasco.OC [Lackorzyński et al. 2012] priority implies criticality.

Finally, *trust* refers to the degree of reliance in the correct behaviour of a component. Untrusted components may fail completely without affecting the core system mission, while a component which must be assumed to operate correctly for achieving the overall mission is trusted. A component is *trustworthy* if it has undergone a process that establishes that it *can be trusted*, the degree of trustworthiness being a reflection of the rigor of this process (testing, certification, formal verification) [Veríssimo et al. 2003].

In practice, criticality and trust are closely aligned, as the most critical parts should be the most trustworthy. However, criticality must be decoupled from time sensitivity in MCS. Referring back to the example in the introduction, interrupts from networks or buses have high time sensitivity, but low criticality (i.e. deadline misses are tolerable), while the opposite is true for the flight control component. Similarly, threads (other than the most critical ones which should have undergone extensive assurance) cannot be trusted to honour their declared WCET.

We need mechanisms that provide enforceable time limits, such that the timeliness of critical threads can be isolated from those of untrusted less critical ones. Reservation-based kernels often allow for a form of over-committing where best-effort threads are run in the slack time left by unused reservations or unreserved CPU. However, this also aligns criticality and time-sensitivity, and enforces a two-level scheduling model (the drawbacks of hierarchical scheduling are explained in Section 2.3).

Static mixed criticality schedulers [Baruah et al. 2011] have emerged in real-time theory, where criticality is orthogonal to priority. Each thread has a criticality level L, and, in addition, there is a system criticality level L_S. The system maps thread priorities onto internal system priorities by adding p_{max} to the priority of all threads with $L \geq L_S$, where p_{max} is the highest thread priority. This puts the priority of critical threads above all less-critical ones.

Under normal operation $L_S = 0$, i.e. all threads get scheduled according to their priorities. If the system becomes critical, i.e. misses deadlines, L_S can be increased, which prioritises threads where $L \geq L_S$, any other threads then only run in slack time left by critical ones. The criticality switch is a get-out-of-jail card for the system, and basically requires dynamic changes of priority.

2.3 Capabilities to time

Capabilities [Dennis and Van Horn 1966] are an established mechanism for fine-grained access control to spatial resources. The seL4 microkernel uses capabilities to provide a memory-management model that delegates all management decisions to userland, including allocation of kernel memory, resulting in a largely policy-free kernel [Elkaduwe et al. 2008; Heiser and Elphinstone 2016]. However, seL4's present capability system does not apply to time.

Prior uses of capabilities for controlling time include KeyKOS [Bomberger et al. 1992], which had *meters*; these granted the holder the right to execute for the unit of time held by the meter. However, the KeyKOS model treats time as fungible, with no guarantee of *when* the time will be provided, making this approach unsuitable for RT use.

Real-time Mach [Mercer et al. 1993] introduced *processor capacity reserves*, which were also implemented in EROS and combined with capabilities [Shapiro et al. 1999]. However, these reserves were optional: a two level scheduler first scheduled the reserves with available capacity, then threads with no or exhausted reserves are scheduled. Like any hierarchical scheduling model, this enforces a policy that reduces flexibility.

Furthermore, hierarchical delegation has the significant disadvantage of algorithmic utilisation loss [Lackorzyński et al. 2012]; this is a direct result of the unfungible nature of time. Consider real estate: like time, it is (arbitrarily) divisible but not fungible: If a block is too small to build a house, then having a second, disconnected block of the same size is of no help (unlike spatial resources in a kernel, which can be mapped side-by-side). The implication is that capabilities for time have a different flavour from those for spatial resources. They cannot support hierarchical delegation without loss, and cannot be recursively virtualised. While delegation is an attractive property of spatial capabilities, this delegation is not their defining characteristic, which is actually *prima facie evidence of access privilege*; in the case of time capabilities, the access is to processor time.

2.4 Resource sharing

As indicated in the introduction, integrity of critical components must be assured even when threads communicate and share resources, e.g. the waypoints in our AAV example. A standard approach is to encapsulate the shared data and the code that accesses and modifies it into a single-threaded *resource server* [Brandenburg 2014]. This is a simple and effective way to achieve the necessary transaction semantics.

If the shared server is accessed by clients of different priority, this creates (temporary) *priority inversion*, where a low-priority thread can lock out a high-priority one. Unless priority-aware locking protocols are used, sharing can also lead to permanent priority inversion, when a thread of intermediate priority monopolises the processor while the low-priority thread holds the critical section.

Standard approaches to mutual exclusion in fixed-priority RT systems [Sha et al. 1990], include non-preemptive critical sections (NCP), the priority inheritance protocol (PIP), and the immediate and original priority ceiling protocols (IPCP[1] and OPCP). For RT systems, an important consideration for mutual exclusion is the bound on priority inversion. Given the safety-critical nature of such systems, implementation complexity (and the resulting potential

[1] IPCP is also known as highest-lockers protocol, stack-based priority-ceiling protocol, and `PRIO_PROTECT` in POSIX.

for bugs) is also of high importance. For efficient systems, the main concern is performance overheads, for secure systems the main concern is the avoidance of covert channels.

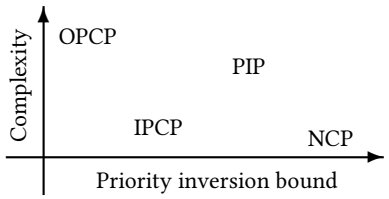


Figure 1: Comparison of RT locking protocols based on implementation complexity and priority inversion bound.

Figure 1 compares the four protocols, none is a silver bullet. PIP comes with high implementation complexity, has potentially high blocking times and can produce deadlock if resource ordering is not used; it is nevertheless popular as it is simple to use. NCP is simplest but has the longest blocking time, while IPCP is also simple but requires the priorities of all lockers to be known *a priori*. OPCP has the shortest blocking time but is even more complex than PIP, and requires global state to be maintained across all locks in the system; this is not acceptable for our purpose, as it introduces covert channels and is incompatible with seL4's decentralised user-level resource management. We will show in Section 3.5 how IPCP can be easily implemented without the kernel requiring knowledge about critical sections.

3 MCS SCHEDULING MODEL

3.1 Requirements

We now propose a model for mixed-criticality scheduling that is suitable for high-assurance systems, such as seL4. The model should satisfy the following requirements:

Capability-controlled enforcement of time limits: Capabilities help to reason about access rights. They allow a seamless integration with the existing capability-based spatial access control of security-oriented systems such as seL4.

Policy freedom: In line with microkernel philosophy [Heiser and Elphinstone 2016], the model should not force systems into a particular resource-management policy. In particular, it should support a wide range of scheduling policies and resource-sharing models, such as locking protocols.

Efficient: The model should add minimal overhead over the best existing implementations. In particular, it should be compatible with fast message-passing (IPC) implementations in high-performance microkernels, such as seL4.

Temporal isolation: The model must allow system designers to create systems where a temporal failure in one component cannot cause a temporal failure in another part of the system, even in the case of shared resources.

The model provides fundamental kernel-level isolation mechanisms, while allowing for complex scheduling policies to be implemented at user level. We target modern, multicore hardware with caches and virtual memory, where WCET estimates are pessimistic by orders of magnitude, allowing for much slack to be exploited in the system.

3.2 High-level concepts

By definition, priority determines what is scheduled next, i.e. the highest-priority runnable thread. In order to support MCS, we do not change the meaning of priority, but what it means for a thread to be *runnable*: We associate each thread with a budget, and make the thread non-runnable if it has exhausted its budget. We retain a static priority model, i.e. the kernel will not adjust priorities on its own (but they can be changed through a system call). We avoid limiting to a specific class of scheduling models by making it possible to implement dynamic priorities at user level (as we will demonstrate in Section 5.5).

3.2.1 Budgets and scheduling contexts. At the core of the model is the **scheduling context** (SC) as the fundamental abstraction for time allocation. An SC is a representation of a reservation in the object-capability system, which means that SCs are first-class objects, like threads, address spaces, or communication endpoints (ports). An SC is represented by a capability to a scheduling context object (scCap).

An scCap represents the privilege to access the processor, it is a capability to time. In order to run, a thread needs an scCap, which represents the maximum CPU bandwidth the thread can consume.

The unfungible nature of time in RT systems requires that the bandwidth limit must be enforced within a certain time window. We achieve this by representing an SC by a *period*, T, and a *budget*, C, where $C \leq T$ is the maximum amount of time the SC allows to be consumed in the period. $U = \frac{C}{T}$ represents the maximum *CPU utilisation* the SC allows. The SC can be viewed as a generalisation of the concept of a time slice that is used on many systems (including present mainline seL4).

In a multicore system, an SC represents the right to access a particular core. Core migration, e.g. for load balancing, is policy that should not be imposed by the kernel but implemented at user level. A thread is migrated by replacing its SC with one tied to a different core.

Setting budgets is admission control and requires appropriate privilege. The total available time on a core is represented in a (virtual) per-core *scheduling-control capability*, `sched_control`, which provides this privilege.[2]

3.2.2 Priorities. Fiasco.OC introduced scheduling contexts into an L4 microkernel [Lackorzyński et al. 2012], They are superficially similar to ours, although Fiasco.OC SCs are not capability-controlled. Furthermore, Fiasco.OC makes priority an SC attribute. In contrast, we retain priority as a thread attribute. The advantage of keeping priority and SCs orthogonal will become evident in Section 3.5.

Like Fiasco.OC, we retain priorities as static attributes (but changeable by user-level through system calls), for a number of reasons. Firstly, fixed-priority scheduling is familiar and widely used in industry (which tends to treat EDF with a high degree of

[2]This is analogous to how seL4 controls the right to receive interrupts: Associating a handler with an IRQ is controlled by the IRQ_control capability. Like time, IRQ sources are non-fungible.

suspicion). It also supports a graceful adaptation of the existing model used in seL4, logically replacing the time-slice attribute with the scCap. Furthermore, it is straightforward to implement EDF on top of this model by using a single priority for EDF's dynamic priorities (and we will demonstrate in Section 5.5 that this can be done with low overhead). The opposite is not true: mapping the dynamic priorities of EDF to a fixed-priority is non-trivial and would come with high overheads. This means that while the kernel provides a particular policy (fixed-priority scheduling), our approach retains the freedom of implementing other policies efficiently.

The final reason to base the approach on fixed priorities is the ability to reason about the behaviour of an overcommitted system. Overcommitting, i.e. the sum of utilisations allowed by all SCs exceeding the schedulability threshold, is important for achieving high actual system utilisation, given the large time buffers required by critical hard real-time threads. It is also core to keeping the kernel policy-free: The degree and nature of overcommitment is a core policy of a particular system. For example, the policy might require that the total utilisation of all HIGH threads is below the RMS schedulability limit of 69%, while LOW threads can overcommit, and the degree of overcommitment may depend on the mix of hard RT, soft RT and best-effort threads. Such policy should be defined and implemented at user level rather than in the kernel.

As indicated in Section 2.1, the result of overcommitting in an EDF-based system is hard to predict, and such a system is hard to analyse. In contrast, with fixed priority the result is easy to understand: If the sum of utilisations of threads at priority $\geq P$ is below the utilisation bound, then all those threads will meet their deadlines, while any thread with priority $< P$ may miss. This allows easy analysis of schedulability.

3.3 Criticality

We do not alter the kernel in order to support a criticality switch as introduced in Section 2.2. This is an action to be performed by a user-level manager (which must hold the `sched_control` capability); the kernel will not change priorities on its own. One way for detecting the need for such a switch is to give critical threads a budget corresponding to an optimistic execution-time bound that is sufficient for normal operation (and supports high utilisation). If, in an exceptional situation, the critical thread exceeds this budget, it is given a larger budget (corresponding to a safe WCET estimate) and criticality is raised to remove interference from high-priority low-criticality threads. The mechanism for detecting deadline misses are timeout exceptions, explained in Section 3.4.2.

3.4 Mechanics

3.4.1 Replenishment. A thread with a *full* SC, where $C = T$, may monopolise whatever CPU bandwidth is left over from higher-priority threads. This is useful for best-effort threads which run in slack time. It can also be used for critical threads that are trusted not to use more than a small share of the CPU except in emergencies. Threads with a full budget incur no inherent overhead other than the preemption rate $1/T$.

Threads with *partial* SCs, where $C < T$, are not runnable once they have used up their budget, until it is replenished. For replenishment, we use the sporadic servers model [Sprunt et al. 1989] with an implementation based on the algorithms presented by Stanovic et al. [2010]. Sporadic servers work by preserving the *sliding window* constraint, meaning that during any time interval not exceeding the period, no more than C can be consumed. This stops a thread from saving budget until near the end of its period, and then running uninterrupted for more than C. It is achieved by tracking any left-over budget when a thread is preempted at time t, and scheduling a replenishment for time $t + T$.

In practice, we cannot track an infinite number of replenishments, so in a real implementation, once the number of queued replenishments exceeds a threshold, any excess budget is discarded. If the threshold is one, the behaviour degrades to polling servers [Sprunt et al. 1989] where any unused budget is lost and the thread cannot run until the start of the next period.

There is an obvious cost to replenishment fragmentation that will arise from preemptions, and polling servers are more efficient in the case of frequent preemption [Li et al. 2014]; an arbitrarily high threshold therefore makes little sense. The optimal value depends on implementation details of the system, as well as the characteristics of the underlying hardware. We therefore make the threshold an attribute of the SC. SCs are variably sized, such that system designers can set this bound per-SC.

If a replenishment is ready at the time the budget expires, the thread is immediately runnable. It is inserted at the end of the ready queue for its priority, meaning that within a priority, scheduling of runnable threads is round-robin.

3.4.2 Budget overrun. Threads may exhaust their budgets for different reasons. A budget may be used to rate-limit a best-effort thread, in which case budget overrun is not different to normal time-slice preemption of best-effort systems. A budget can be used to force an untrusted thread to adhere to its declared WCET. An overrun is then a contract violation, which may be reason to suspend the thread or restart its subsystem. Finally, an overrun by a critical thread can indicate an emergency situation; for example, critical threads may be scheduled with an optimistic budget to provide better service to less critical threads, and overrun may require provision of an emergency budget.

The handling of overrun is a system-specific policy, and the kernel should only provide appropriate mechanisms for implementing the desired policy. Our core mechanism is the *timeout exception*, which is raised when a thread is preempted. To allow the system to handle the exceptions, each thread is optionally associated with a timeout-exception handler, which is the temporal equivalent to a (spatial) protection exception handler. When a thread is preempted, the kernel notifies its handler via an IPC message. The exception is ignored when the thread has no timeout-exception handler.

The handler has the choice of a range of overrun policies, including (i) providing a one-off (emergency) budget to the thread and letting it continue, (ii) permanently increasing the budget in the thread's SC, (iii) changing the system's criticality by deprioritising less critical threats, (iv) killing/suspending the thread and, or (v) abandoning the request and rolling the server back. Obviously, these are all subject to the handler having sufficient authority (e.g. `sched_control` for budget).

3.5 Resource sharing

Access to encapsulated, shared resources requires cross-address-space IPC. For minimising invocation cost, it is essential that scheduler invocations are avoided during IPC. This has historically been done by L4 microkernels [Heiser and Elphinstone 2016]. Not bypassing the scheduler is the main reason why IPC on most other systems is significantly slower than that of L4 kernel, e.g. at least a factor of four in CertiKOS [Gu et al. 2016].

Past L4 kernels avoided the scheduler by time-slice donation, where a server could execute on the client's time slice. While fast, this model is unprincipled and hard, if not impossible, to analyse. For example, the time slice may expire during the server's execution, after which the server will run on its own time slice. The server's execution time is not consistently accounted, and there is no temporal isolation.

Our model supports shared servers, including scheduler bypass, in a principled way through *scheduling-context donation*: A client invoking a server can pass its SC along, so the server executes on the client's SC, until it replies to the request. This ensures that time consumed by the server is billed to the client requesting the work.

Such a shared server is implemented as *passive*, meaning it has no SC of its own, and is therefore not runnable, except on a client's borrowed SC. In contrast, an active server, which has its own SC, executes "for free" from the client's point of view. The model supports both, as not all systems require strict temporal isolation.

Passive servers effectively provide a migrating-thread model [Ford and Lepreau 1994; Gabber et al. 1999], but without requiring the kernel to manage stacks. They also provide a simple, and essentially free, implementation of IPCP (recall Section 2.4): The server is configured with the ceiling priority of its clients, making its execution atomic with respect to all clients. This is possible thanks to decoupling SCs from priorities. The main drawback of IPCP, namely the requirement that all lockers' priorities are known *a priori*, is easy to enforce in a capability-based system: The server can only be accessed through an appropriate invocation capability, and it is up to the system designer to ensure that such a capability can only go to a thread whose priority is known or appropriately controlled.

Figure 2 shows a simplified architecture of the AAV discussed in Section 1. There are a number of passive components, which are essentially encapsulated shared datastrcutureas that require transaction semantics. Most communication is by RPC-type message-passing. Device drivers signal I/O completion via Notifications (dashed arrows), such communication generally uses shared-memory buffers.

3.6 Timeout exceptions

If a passive server exhausts its budget, it and any waiting clients are blocked until the budget is replenished. On its own, this means that a client not only has to trust its server, but all the server's other clients. This would rule out sharing a server between clients of different criticality.

Timeout exceptions are the mechanism for avoiding this need for trust, allowing a server to be shared across criticalities. The server's timeout handler can implement any of the options discussed in Section 3.4.2. A server running out of budget constitutes a protocol violation by the client, and it makes sense to penalise that client by aborting. This option is attractive, as it minimising the amount of budget that must be reserved for overruns.

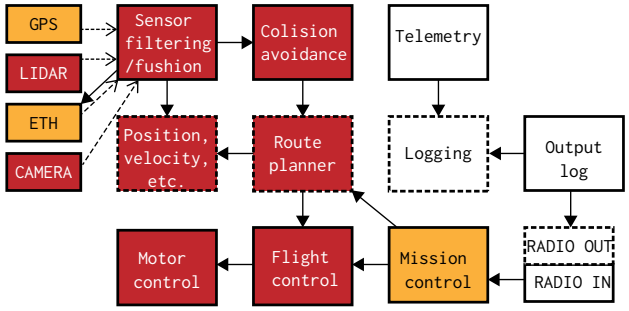


Figure 2: Simplified AAV design, showing main components (seL4 address spaces). Solid frames indicate active components, broken frames passive ones. Solid arrows indicate RPC-style server invocation, dashed arrows are completion signals. For clarity we do not show most invocations of the logging service, this would add a solid arrow from each component to the "Logging" box. Most critical components are shown in red, least-critical ones in white.

In contrast, helping schemes, such as PIP or bandwidth inheritance, as implemented in Fiasco.OC [Lackorzyński et al. 2012], make the waiting client pay for another client's contract violation. This not only weakens temporal isolation, it also implies that the size of the required reserve budget must be server's full WCET. This places a restriction on the server that no client request exceed a blocking time that can be tolerated by all clients, or that all clients must budget for the full server WCET in addition to the time the server needs for serving their own request. Our model provides more flexibility: a server can use a timeout exception to finish a request early (e.g. by aborting), meaning that clients can only be blocked for the largest budget of the other clients, plus the short cleanup time.

Where desired, helping schemes can be implemented on top of our model, e.g. by using a gateway server that adjusts priorities as requests come in.

4 IMPLEMENTATION IN SEL4

4.1 Objects

We add to seL4 a new variably-sized *scheduling context object* (SC) type, whose minimum size is 2^8 bytes, sufficient for holding 8 or 10 replenishments on 64 or 32-bit processors, respectively. This is enough for most uses, but more replenishments can be supported by larger sizes. Sporadic replenishments specify the time at which they become usable, encoded in 16 bytes. An SC contains the budget and period, the ID of the processor core to which it provides access, and some pointers to ensure it will return to its rightful owner across a donation chain (see Section 4.4).

Like any seL4 object, an SC is created by re-typing some free ("Untyped") memory into the SC type, which will create an array

of empty SCs with zero budget. This allows anyone holding a capability to Untyped to create SCs. Setting the budget to non-zero is admission control and requires the `sched_control` capability.

A scheduling context can be associated with a *thread control block* (TCB), which makes the corresponding thread schedulable (provided the SC has budget). An SC can also be associated with a *Notification object*, a small array of binary semaphores. When signalled, such an active Notification will donate its SC to a thread waiting for the Notification. This is useful for allowing passive servers to wait for IPC as well as Notifications, e.g. for handling exceptions or I/O completions, without forcing a multi-threaded implementation. An SC can only be associated with a single TCB or Notification at any time, and TCBs and Notifications can be associated with at most one SC.

Resume objects, modelled after KeyKOS [Bomberger et al. 1992], are a new object type that generalise the "reply capabilities" of baseline seL4. These were capabilities to virtual objects created by the kernel on-the-fly in seL4's RPC-style `call()` operation, which sends a message to an endpoint and blocks on a reply. The receiver of the message (i.e. the server) receives the reply capability in a magic "reply slot" in its capability space. The server replies by invoking that capability. Resume objects remove the magic by explicitly representing the reply channel (and the SC-donation chain). They also provide more efficient support for stateful servers that handle concurrent client sessions, see Section 4.2.

With SC donation we need slightly more bookkeeping to guarantee that a donated SC eventually returns to its rightful owner, even if the server invokes another passive server, or the server operation is long-running and the server handles multiple requests concurrently (as a file server would).

The Resume object consists of three pointers: (i) the active thread who owns the SC, (ii) the immediate caller's Resume object, (iii) the next Resume object in the chain.

In TCB objects we replace the legacy timeslice by the scCap. We add a total of five fields: a timeout-handler capability, a Resume object pointer, an *MCP* value, and some bookkeeping data. As seL4 object sizes need to be powers of two, there was sufficient unused space so the additional data did not grow the TCB size.

The MCP, or *maximum controlled priority*, resurrects a concept from early L4 kernels [Liedtke 1996]. It supports lightweight, limited manipulation of thread priorities, useful e.g. for implementing user-level thread packages. When setting the priority or MCP of a TCB, *A*, the caller must provide the capability to a TCB, *B*, (which could be the caller's TCB). The caller is allowed to set the priority or MCP of *A* up to the value of *B*'s MCP.[3] In a typical system, most threads will run with an MCP of zero and have no access to TCB capabilities with a higher MCP, meaning they cannot raise any thread's priority. The MCP is taken from an explicitly-provided TCB, rather than the caller's, to avoid the confused deputy problem [Hardy 1988].

We add methods operating on TCBs for managing MCP and configuring SCs. There are five methods on SCs, for binding and unbinding objects, as well as an explicit yield which allows threads with access to a SC cap to move that SC to the front of the scheduler queue for its priority, enabling user-level scheduling. All of these methods are shown in Figure 3.

[3]Obviously, this operation also requires a capability to *A*'s TCB.

```
/* Bind a TCB or Notification to this SC. */
SchedContext_Bind(cap_t sc, cap_t obj);
/* Unbind all objects from this SC. */
SchedContext_Unbind(cap_t sc);
/* Unbind a specific object from this SC. */
SchedContext_UnbindObject(cap_t sc, cap_t obj);
/* Return the amount of time since the last
   timeout fault, consumed or yieldTo. */
SchedContext_Consumed(cap_t sc);
/* Place the TCB bound to this SC at the front
   of its prio queue & return consumed. */
SchedContext_YieldTo(cap_t sc);
/* Configure a scheduling context */
SchedControl_Configure(cap_t sc,
    uint64_t budget, uint64_t period,
    word_t extra_refills, word_t badge);
/* Set a TCBs timeout fault handler */
TCB_SetTimeoutEndpoint(cap_t tcb, cap_t ep);
/* Set prio, mcp of tcb deriving authority from
   auth_tcb, and bind sc */
TCB_SetSchedParams(cap_t tcb, cap_t auth_tcb,
    word_t mcp, word_t priority, cap_t sc);
```

Figure 3: Additions to the kernel API.

4.2 System calls

The introduction of Resume objects requires some changes to the IPC system-call API. The client-style `call()` operation is unchanged, but server-side equivalent, `ReplyRecv` (previously `ReplyWait`) replies to a previous request and then blocks on the next one. It now must provide an explicit Resume capability; on the send phase, that capability identifies the client and returns the SC if appropriate, on the receive phase it is populated with new values.

The new API makes stateful server implementation more efficient. In baseline seL4, the server would have to use at least two extra system calls to save the reply cap and later move it back into its magic slot, removing the magic also removes the need for the extra system calls.

There are two new methods for atomically signalling a Notification and then blocking on a message; these are needed to initialise passive servers. New API methods are presented in Figure 4.

4.3 Scheduling algorithm

We convert seL4 from a tick-based to a tickless kernel in order to reduce preemptions and improve scheduling precision. seL4 is non-preemptible (save for explicit preemption points in a few long-running operations) [Klein et al. 2009]. This makes tickless kernel design non-trivial, as preemption interrupts cannot interrupt the kernel itself.

seL4 has a ready queue, which satisfies the invariant that it contains all runnable threads except the one presently executing [Blackham et al. 2012]. It is implemented as a priority-indexed array of queues. A two-level bitfield of occupied priorities ensures O(1) access. We add a *release queue*, where the kernel queues threads that are out of budget. This retains the existing invariant for the

```
/* Wait for a message, when it is
   received store the object badge
   and block the caller on reply */
seL4_Recv(cap_t src, word_t *badge,
          cap_t reply);
/* As above, but invoke reply first */
seL4_ReplyRecv(cap_t src, word_t msg_info,
    word_t *badge, cap_t reply);
/* As above, but invoke dest, not reply. */
seL4_NBSendRecv(cap_t dest, word_t msg_info,
   cap_t src, word_t *badge, cap_t reply);
/* As above, with no reply. Donation not
   possible. */
seL4_NBSendWait(cap_t cap, word_t msg_info,
   cap_t src, word_t *badge);
/* As per recv, but donation not possible */
seL4_Wait(cap_t src, word_t *badge);
```

Figure 4: Additions and modifications to the kernel IPC API.

ready queue, while the release queue is characterised as holding all threads that would be runnable if they had budget. The queue is ordered by the time when the next replenishment is available.

When the kernel schedules a thread, it sets the timer to fire at the thread's SC's budget expiry, or the next wake-up time for the head of the release queue (whichever is first). If an SC switch occurs, because the timer fires or the thread blocks without an SC transfer, the consumed time is subtracted according to the sporadic server algorithm and the accumulated time is updated.

On kernel entry (except on the IPC fastpath, which never leads to an SC change or scheduler invocation) the kernel updates the current timestamp and stores the time since the last entry. It then checks whether the thread has sufficient budget to complete the kernel operation. If not, the kernel pretends the timer has already fired, resets the budget and adds the thread to the release queue.

This adds a new invariant, that any thread in the scheduling queues must have enough budget to exit the kernel. It makes the scheduler precision equal twice the kernel's WCET (which is known [Blackham et al. 2011; Sewell et al. 2017]). This invariant is required, as it simplifies the kernel design and actually reduces the WCET: when a thread runs out of time, it may need to raise a timeout exception, resulting in delivering an IPC to a timeout handler. By requiring that all threads in the ready queue, or any endpoint queue, must have enough budget to wake up, we avoid the need to raise timeout exceptions on many wakeup paths in the kernel.

Threads are only charged when the scheduling context changes, in order to avoid reprogramming the timer, which is expensive on many platforms. If there is no SC change, the timestamp update is rolled back by subtracting the stored consumed value from the timestamp. Figure 5 illustrates the structure of this kernel design.

4.4 Scheduling context donation

Donation occurs where the target thread of an IPC (or Notification wait) does not have a scheduling context. As an SC represents the right to access a core, donation can only happen intra-core. If a

```
/* if possible do fastpath & return */
old_time = kernel_time;
kernel_time = cur_time();
/* calculate amount consumed */
cons = kernel_time - old_time;
cur_SC=cur_TCB->SC;
if (cur_SC->budget - cons > 2*WCET) {
  /* budget is sufficient */
  need_reschedule = slowpath(syscall);
} else {
  /* budget is not sufficient */
  charge(cur_SC);
  need_reschedule = true;
}
if (need_reschedule)
  schedule();
if (cur_SC==cur_TCB->SC) {
    /* rollback time */
    kernel_time -= cons;
} else {
    charge(cur_SC);
    reprogram_timer();
}
```

Figure 5: Kernel accounting logic.

passive server previously executed on a different core, its execution context is migrated to the client's core on invocation. This is overhead which must be included in any schedulability analysis (or avoided by design).

As indicated in Section 4.1, we use Resume capabilities to track scheduling donation chains, Figure 7 shows how this works. A passive server, S, blocks on Endpoint E and accepts a request from client A. As S is passive, SC_A is donated to S during the IPC rendezvous. Additionally, the kernel blocks A on the Resume object provided by S on `recv`. A remains blocked until the server replies on its Resume object, at which time the SC returns to A. There are no limits on SC donation; this process can be nested indefinitely.

While S is running on behalf of A, its scheduling context SC_A points to R_A, which in turn could point to further resume objects in the IPC call chain. Timeout handlers are a thread attribute, so while A has none, S has one, as shown in Figure 7c, where SC_A expires while the server is processing the request.

5 EVALUATION

We evaluate our design and implementation for overheads, achieved isolation through shared and non-shared resources, and user-level implementation of scheduling and criticality. Our evaluation platforms are as follows, with further details in Table 1.

- **ARM:** Cortex A9 system-on-chip on a Freescale i.MX6 SABRE Lite development board,
- **x64:** Haswell i7-4770 machine.

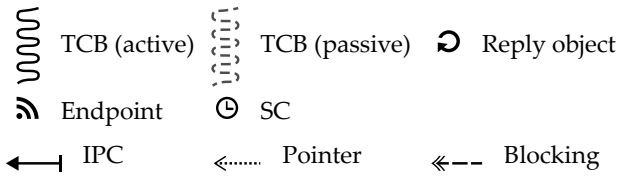
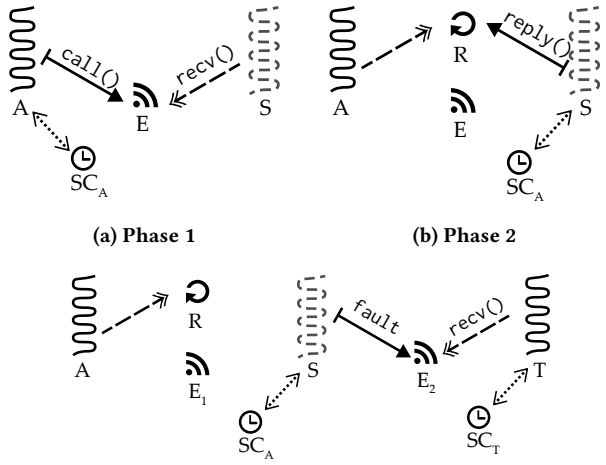


Figure 6: Legend for diagrams in this paper.



(a) Phase 1 **(b) Phase 2**

(c) Timeout exception in server

Figure 7: IPC between an active client and passive server: (a) shows the initial rendezvous, (b) shows the reply. SC_A is donated between client and server. (c) SC_A expires and a fault is sent to the server's timeout exception endpoint E_2. See Figure 6 for the symbols used.

Platform	*Clock* GHz	*L1* *KiB*, ×	*L2* *KiB*, ×	*L3* *KiB*, ×	*TLB* *entr.*, ×
ARM (32-bit)	1.0	32+32, 4	1024, 16	N/A	64, 2
x64 (64-bit)	3.1	32+32, 8	256, 8	8,192, 16	128, 8

Table 1: Hardware platform details, "×" is associativity.

5.1 Overheads

Table 2 shows the cost of the (performance-wise) most important kernel operations of our present implementation (MCS), compared to the baseline seL4 kernel (Base). `call()` is the client-style invocation, which sends a message to an IPC endpoint (port), blocking until receiving a reply. The server-style operation `replyRecv()` replies to a previous message by invoking the Resume object and then blocks, waiting for the next incoming request. The cost of these operations increases by a few percent on both platforms, resulting from extra checks on the fastpath to accommodate scheduling contexts, an extra capability lookup for the Resume object, touching two separate new objects (SC and Resume object) and enforcing priorities on IPC delivery (baseline does FIFO). ARM shows a higher

Arch	*Op.*	*Base*	*MCS*	*O/H*
ARM	call	288 (1)	350 (57)	62 21%
	replyRecv	313 (1)	339 (30)	26 8%
	IRQ lat.	1604 (81)	1813 (113)	209 13%
	signal	138 (*)	144 (*)	6 4%
	schedule	1910 (26*)	2778 (30*)	868 45%
x64	call	448 (0)	455 (1)	7 1%
	replyRecv	431 (1)	450 (2)	19 4%
	IRQ lat.	1144 (54)	1245 (54)	101 8%
	signal	136 (*)	136 (*)	0 0%
	schedule	1512 (0*)	2100 (0*)	589 38%

Table 2: Microbenchmarks (cycles) of seL4-base vs. MCS kernels, standard deviations shown in brackets. "*" indicates an average of 10K runs as single measurements showed high variance due to pipeline effects on our hardware.

IPC overhead, as the larger number of objects accessed is causing conflicts in the two-way associative TLB.

We measure IRQ latency using a thread spinning in a loop, updating a volatile cycle counter, while another, higher priority thread waits for an interrupt. On delivery, the handler thread determines the IRQ latency by subtracting the looped timestamp from the current time. The overhead is higher here, as we must switch scheduling contexts, which requires reprogramming the timer.

The `signal()` operation signals a Notification object (semaphore). This microbenchmark evaluates the cost of signalling a lower priority thread – no scheduling context switch occurs, but the kernel reads the cycle counter before determining whether a budget needs updating.

The `schedule` benchmark measures the cost of a signal to a higher priority thread, which forces a reschedule. Scheduling cost increases noticeably due to the need for first reading and then reprogramming the timer for budget enforcement. Furthermore, the sporadic replenishment logic is far more complicated than the previous tick-based logic, and there is some extra code for dealing with scheduling contexts. Note that seL4 IPC, particularly scheduler-context donation (and its predecessor, the undisciplined timeslice donation), is designed to minimise the need for invoking the scheduler, therefore this increase is unlikely to have a noticeable effect in practice. In fact, the $O(1)$ scheduler is a recent addition to seL4, scheduling used to be far more expensive.

All in all, our overheads are reasonable given the speed of the baseline kernel and the extent of the provided functionality.

5.2 Temporal Isolation

Process isolation. We demonstrate isolation in a single-core setup illustrated in Figure 8a. It consists of a Linux VM, running at high priority with a constrained budget, and a UDP-echo server running at a lower priority, representing a lower-rate HIGH thread. We measure the average and maximum UDP latency reported by the ipbench [Wienand and Macpherson 2004] latency test.

Specifically, the Linux VM interacts with the timer (PIT) and serial device drivers implemented as passive servers outside the

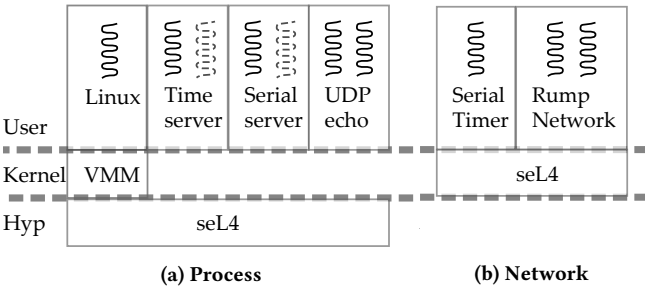


Figure 8: Architecture of isolation benchmarks.

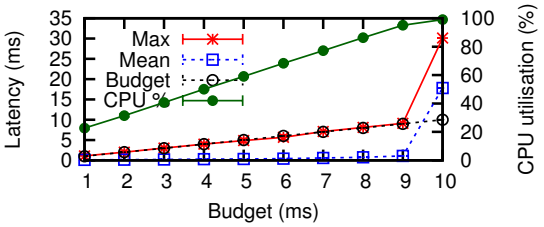


Figure 9: Average and maximum latency of UDP packets with a high-priority CPU hog running with a 10 ms budget.

VM; all three components are at a high priority. In the Linux server we run a program (`yes > /dev/null`) which consumes all available CPU bandwidth. The UDP echo server, completely isolated from the Linux instance, but sharing the serial driver, runs at a low priority.

Two client machines run ipbench daemons to send packets to the UDP-echo server on the target machine (Haswell platform). The control machine, one of the load generators, runs ipbench with a UDP socket at 10 Mbps over a 1 Gb/s Ethernet connection with 100-byte packets. The Linux VM has a 10 ms period, and we vary the budget between 1 ms and 10 ms. Any time not consumed by Linux is available to UDP echo for processing 10,000 packets per second, or 100 packets in the time left over from each of Linux's 10 ms period.

Figure 9 shows the average and maximum UDP latencies for ten runs at each budget setting. We can see that the maximum latencies follow exactly the budget of the Linux server (black line) up to 9 ms. Only when Linux has a full budget (10 ms), and thus is able to monopolise the processor, does the UDP server miss its deadlines, resulting in a latency blowout. This result shows that our sporadic server implementation is effective in bounding interference of a high-priority process.

Network benchmark. We evaluate overheads and again demonstrate temporal isolation by running the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al. 2010]. We run it against a server using the Redis key-value store [RedisLabs 2009] with NetBSD [Foundation 2018] drivers at user-level provided by a single-core Rump library OS [Kantee and Cormack 2014] on x86.

The system consists of Redis/Rump running on three active seL4 threads: two for servicing interrupts (network, timer) and one for Rump, as shown in Figure 8b. Interrupt threads run at the highest priority, followed by Redis, and a low-priority idle thread (not shown) for measuring CPU utilisation; this setup forces frequent invocations of the scheduler and interrupt path. Table 3 shows the achieved throughput of Redis+Rump running bare-metal (BMK), and Redis on the seL4 baseline and as well as the MCS branch, plus Linux and NetBSD for comparison.

System	*IRQ*	*Tput* (k ops/s)	*Utilis.* (%)	*Cost per op.*	*Latency* (ms)
seL4-base	APIC	138.7 (0.4)	100	0.72	1.4
seL4-MCS	APIC	138.5 (0.3)	100	0.72	1.4
seL4-MCS	MSI	127.3 (0.6)	100	0.79	1.6
NetBSD	MSI	134.0 (0.2)	99	0.74	1.5
Linux	MSI	179.4 (0.4)	95	0.52	1.1
Linux	APIC	111.9 (0.4)	100	0.89	1.8
BMK	PIC	144.1 (0.2)	100	0.69	1.4

Table 3: Throughput (k ops/s) achieved by Redis using the YCSB workload A with 2 clients. Latency is average Read and Update, standard deviations in parentheses and omitted where less than the least significant digit shown.

The table indicates the interrupt handling method used, as there is no single method supported across all four scenarios. BMK only supports the legacy programmable interrupt controller (PIC), while NetBSD only supports message-signalled interrupts (MSI). Linux and seL4 both support the advanced PIC (APIC).

The utilisation figures show that the system is fully loaded, except in the Linux case, where there is a small amount of idle time. The cost per operation (utilisation over throughput) is best on Linux, a result of its highly optimised drivers and network stack. Our bare-metal and seL4-based setups use Rump's NetBSD drivers, and achieve performance within a few percent of native NetBSD. This indicates that our MCS support comes with low overhead.

We next run Redis beside a high-priority CPU-hog thread competing for CPU time. All threads have a 5 ms period. We use the budget of the hog to control the amount of time left over for the server configuration. Figure 10 shows the throughput achieved by the YCSB-A workload as a function of the available CPU bandwidth (i.e the complement of the bandwidth granted to the hog thread). All data points are the average of three benchmark runs.

The graph shows that the server is CPU-limited (as indicated by very low idle time) and consequently throughput scales linearly with available CPU bandwidth.

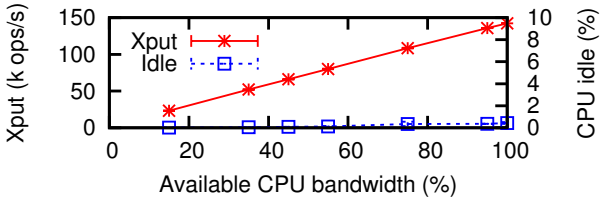


Figure 10: Throughput of Redis YCSB workload A and idle time vs available bandwidth.

Arch	*Op.*	*Cache*	*Min*	*Max*	*Mean*	*σ*
ARM	RB	hot	10.7	13.0	11.8	0.5
		cold	19.4	24.7	23.3	1.2
	Emg.	hot	4.2	5.6	4.7	0.3
		cold	13.0	14.4	13.6	0.3
	Ext.	hot	0.8	3.1	1.9	0.4
		cold	6.8	7.8	7.1	0.3
	Kill	hot	9.9	12.2	11.0	0.5
		cold	21.4	22.7	22.1	0.3
x86	RB	hot	1.47	2.89	1.96	0.39
		cold	4.05	4.96	4.57	0.16
	Emg.	hot	0.97	1.77	1.25	0.18
		cold	2.27	3.04	2.43	0.09
	Ext.	hot	0.16	0.86	0.28	0.13
		cold	0.96	1.46	1.13	0.08
	Kill	hot	1.37	1.49	1.40	0.01
		cold	4.03	4.75	4.36	0.16

Table 4: Cost of timeout handler operations in μs, as measured by timeout exception handler. σ = standard deviation.

Server isolation. As an example of a shared server running out of budget, we implement the scenario of a passive server (see Figure 7) with two clients, A and B. The server is providing an encryption service using AES-256 with a block size of 16 bytes. The server alternates between two buffers, of which one always contains consistent state, the other is dirty during processing. Both clients request 4 MiB of data to be encrypted, and have budget insufficient to complete the request (budgets of 1 ms and 5 ms with a 10 ms period).

As illustrated in Figure 7c, when a server runs out of budget, its timeout exception handler gets invoked. We implement four of the timeout strategies of Section 3.4.2, measuring the latency from the time the handler wakes up until it replies to the server.

Rollback (RB) resets the server to the last consistent state recorded, and replies to the client on behalf of the server with the amount of data successfully encrypted. The client then restarts its request once its budget is replenished. We measure rollback time, from the time the exception handler is invoked, until the server is ready for the next request. Given the small amount of rollback state, this measures the baseline overhead. For schedulability analysis, the actual cost of the rollback would have to be added.

Emergency (Emg.) gives the server a one-off emergency budget to finish the client request, after which the exception handler resets the server to being passive. The benchmark measures the pure handling overhead, the actual request completion time must again be added.

Extend (Ext.) increases the client's budget on a timeout.

Kill destroys the client.

We run the benchmarks with hot caches (primed by some warmup iterations) as well as cold (flushed) caches.

Table 4 shows the results. The maximum cold-cache cost, which is relevant for schedulability analysis, differs by a factor of 3–4 between the different recovery scenarios, indicating that all are about equally feasible. Approaches that restart the server (RB, Kill) are the most expensive, as they must restore the server state from a checkpoint and follow the passive server initialisation protocol (recall Section 3.5). This requires 5 system calls in each case for killing or replying to the client and RPC-ing to the server.

We next demonstrate temporal isolation in the server by using the RB technique and measuring the time taken to encrypt 10 requests of 4 MiB of data. Figure 11 shows the result with both clients having the same period, which we vary between 10 ms (left graphs) and 1 s (right). In each graph we vary the clients' budgets between 0 and the period. The extreme ends are special, as one of the clients has a full budget and keeps invoking the server without ever getting rolled back, thus monopolising the processor. In all other cases, each client processes at most 4 MiB of data per period, and either succeeds (if the budget is sufficient) or is rolled back after processing less than 4 MiB.

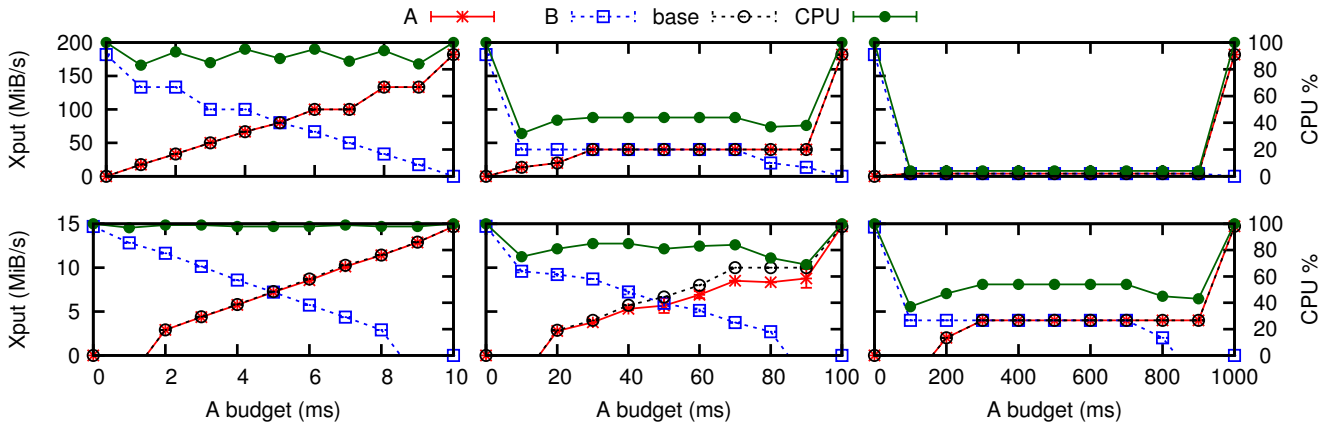


Figure 11: Throughput for clients A and B of a passive AES server processing 10 requests of 4 MiB of data with limited budgets on the x64 (top row) and ARM (bottom row) platforms. The two clients' budgets add up to the period, which is varied between graphs (10, 100, 1000 ms). Clients sleep when they process each 4 MiB, until the next period, except when their budgets are full. Each data point is the average of 10 runs, error bars show the standard deviation.

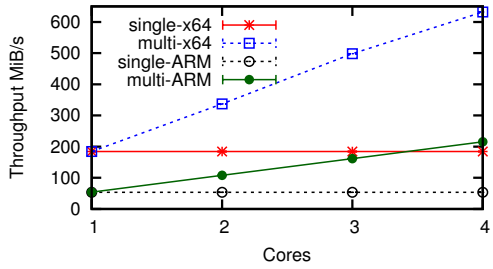


Figure 12: Results of the AES shared-server multicore case study on Sabre and Haswell. Single shows results for a passive server thread migrating between cores to service clients while multi has one passive thread per core. For both series, the number of clients is equal to the number of cores and each client requests 1MiB of data encrypted.

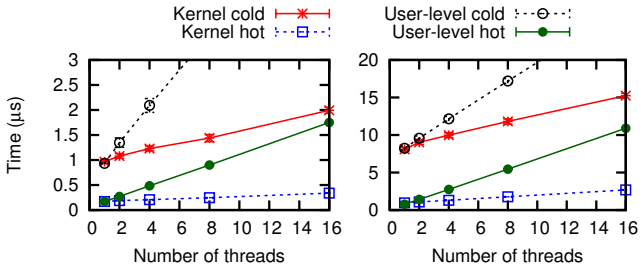


Figure 13: Cost of switching the priority of n threads in kernel and user level, with hot and cold caches, on Haswell (left) and Sabre (right). All data points are the average of 100 runs, with very small standard deviations.

The results show that in the CPU-limited cases (left graphs) we have the expected near perfect proportionality between throughput and budget (with slight wiggles due to the rollbacks), showing isolation between clients. In the cases where there is headspace (central regions of the right graphs), both clients achieve their desired throughput.

5.3 Multicore

We adapt our AES case study to demonstrate how the MCS model applies to multiprocessors. The AES server is configured without a timeout fault handler, and we run two variants. The first is a single, passive server thread which migrates between cores on invocation. This serialises the server; consequently, there is no gain in throughput when further cores are added, as Figure 12 shows. The second scenario has one endpoint and one client per core, and one server thread per core, showing a parallel server workload. Due to minimal bottlenecks in the stateless AES server, this results in near perfect scalability.

5.4 Criticality

We implement a kernel mechanism for changing criticalities of threads in bulk and compare with a user-level approach which simply changes the priority of threads one at a time. The kernel approach tracks a queue of threads for each criticality, in order to quickly access all threads of a specific criticality, and boost their priority. However, given threads are kept in per-priority queues, each thread must be removed and reinserted into a new queue. We evaluate this approach and compare it to changing priority at user-level.

Figure 13 shows the results measured with a primed cache (hot) and flushed cache (cold). As the graph shows, switching is linear in the number of threads being boosted.

In absolute terms, the results show that a criticality switch is fairly fast, the in-kernel implementation remaining under 2 μs on x64 and about 12 μs on ARM for switching 8 threads with a cold cache. Changing the priority at user-level is also linear in the number of threads to be boosted, with a higher overhead due to extra kernel entries for each thread changed. However, most systems will not have more than a few high-criticality threads, and deadlines for critical control loops in cyber-physical systems tend to be in the tens of milliseconds, we conclude that criticality can be implemented at user-level, in line with standard microkernel philosophy.

The higher cost from user-level operation results from multiple switches between kernel and user mode, and the repeated thread-capability look-ups. It could be significantly reduced if seL4 had a way to batch system calls, but to date we have seen no compelling use cases for this.

As a second criticality-switch benchmark, we port three processor-intensive benchmarks from the MiBench [Guthaus et al. 2001] to act as workloads. We use SUSAN, which performs image recognition, JPEG, which does image encoding/decoding, and MAD, which plays an MP3 file. Each benchmark runs in its own Rump process with an in-memory file system, and shares a timer and serial server. We chose these specific benchmarks as they were the easiest to adapt as described below, rather than for comparing systems, so there is no issue of bias from sub-setting.

We alter the benchmarks to run periodically in multiple stages. To obtain execution times long enough, some benchmarks iterate a fixed number of times per stage. Each benchmark process executes its workload and then waits for the next period to start. Deadlines are implicit: if a periodic job finishes before the start of the next period it is considered successful, otherwise the deadline is missed.

We run the benchmarks on both the user-level and in-kernel implementations of static mixed criticality, with 10 runs of each.

SUSAN, the most critical, has three stages: edge detection, smoothing, and corners. The next critical task, JPEG, has two stages: encode, and decode. The least critical task, MAD has only one stage. We run the benchmark for 20 s for each of the stages (repeating the last phase where threads have no new phase), and increment the system criticality level at stage transition. The parameters are arranged such that rate-monotonic priorities are inverse to the criticalities.

Results are shown in Table 5. Only the lowest-priority thread is affected by the criticality switch, with an additional missed deadline due to perturbations in run time due to the user-level versus kernel scheduler. For stage one, the entire workload is schedulable and there are no deadline misses. For stage two, the workload is not schedulable, and the criticality switch boosts the priorities of SUSAN and JPEG, such that they meet their deadlines, but MAD does not. In the final stage, only the most critical task meets all deadlines. This

App. (L)	SUSAN ($L = 2$)					JPEG ($L = 1$)					MAD ($L = 0$)					Util.
L_S	T	C	U	j	m	T	C	U	j	m	T	C	U	j	m	
0	180	25	0.14	111	0	100	15	0.15	200	0	112	28	0.25	179	0	52%
1		51	0.15	111	0		41	0.41	200	0		28	0.25	178	48	86%
2		127	0.54	111	0		41	0.41	155	89		28	0.25	5 (+1)	5 (+1)	100%

Table 5: Results of criticality-switch benchmark, where the system criticality L_S is raised each stage. T = period, C = worst observed execution time (ms), U = allowed utilisation (budget/period), m = deadline misses, j = jobs completed. Standard deviations are less than one significant figure of the result. Observed difference between a user-level and kernel criticality switch in brackets.

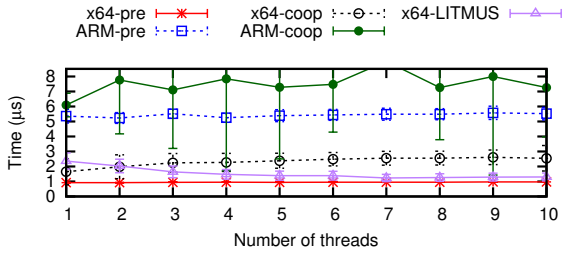


Figure 14: Execution time of seL4 user-mode EDF scheduler compared to kernel scheduler in x64 LITMUS^{RT}.

shows that it is sufficient to implement criticality at user-level, and our mechanisms operate as intended.

5.5 User-level scheduling

Keeping policy out of the kernel is fundamental to the microkernel philosophy, which aims to provide general mechanisms that allow the implementation of arbitrary policies [Heiser and Elphinstone 2016]. On the face of it, our fixed-priority-based model seems to violate this principle. Here we demonstrate that the model is general enough to support the efficient implementation of alternate policies at user level. Specifically, we show that we can efficiently implement EDF, the optimal dynamic-priority policy favoured by much theoretical work (see Section 2.1).

We implement the EDF scheduler as an active server with active clients which run at an seL4 priority below the scheduler. The scheduler waits on an endpoint on which it receives messages from its clients and the timer.

Each client has a period, representing its relative deadline, and a full reservation (equal to the period). Clients either notify the scheduler of completion by an IPC message, or else create a timeout exception on preemption, which is also received by the scheduler. Either is an indication that the next thread should be scheduled.

We use the *randfixedsum* [Emberson et al. 2010] algorithm to generate deadlines between 10 and 1000 ms for a particular number of threads. A set of threads runs until 100 scheduling decisions have been recorded. We repeat this 10 times, resulting in 1,000 scheduler runs for each data point.

We measure the scheduler latency by recording the timestamp when a thread (client or idle) detects a context switch, and process the difference in timestamp pairs offline. We run two schedulers: *preempt* where threads never yield and must incur a timeout exception, and *coop*, where threads use IPC to yield to the scheduler. The latter invokes the user level timer driver more often as the release queue is nearly always full, which involves more kernel invocations to acknowledge the IRQ, in addition to reprogramming the timer.

We compare our latencies to those of LITMUS^{RT} [Calandrino et al. 2006], a widely-used framework for developing real-time schedulers and locking protocols. As it is embedded in Linux, LITMUS^{RT} is not aimed at high-assurance systems.

We use Feather-Trace [Brandenburg and Anderson 2007] to gather data while running the C-EDF scheduler, which is a partitioned (per-core) EDF scheduler, on a single core. We use the same parameters and thread sets, running each set for 10 s. The measured overhead considers the in-kernel scheduler, context-switch and user-level code to return to the user.

Figure 14 shows that our preemptive user-level EDF scheduler implementation is actually faster than the in-kernel EDF scheduler from LITMUS^{RT}, and that the cost of implementing scheduling policy at user level is of the same order as the in-kernel default scheduler. In other words, implementing different policies on top of the base scheduler is quite feasible.

5.6 Summary

Our evaluation demonstrates that our model has low overheads, achieves isolation and allows for efficient user-level scheduling. The implementation adds about 2,000 lines of code (14 %), as measured by SLOC [Wheeler 2001] on the preprocessed kernel code for the Sabre (the verified platform).

6 RELATED WORK

Composite [Parmer and West 2011] removes scheduling decisions from the kernel, invoking a user-level scheduler instead. It reduces the overhead-related capacity loss by configuration buffers, which are shared between user-level and the kernel. Some capacity loss remains, as timer interrupts must be delivered down the scheduling hierarchy. This approach does not suit seL4, as the required reasoning about concurrent access (by kernel and user-level) to those buffers would drastically increase verification effort [Klein et al. 2014]. In our model, the kernel enforces isolation, with the policy tailorable by user-level code, the trusted parts of which can be kept trivially small in most cases.

Composite implements a migrating thread model [Parmer 2010] which requires server components to manage thread pools for performance, and also forces servers to be multi-threaded. Our model of passive servers has the same benefits as migrating threads without

the policy; multi-threaded servers are possible but not mandatory. Recently temporal capabilities were introduced to Composite [Gadepalli et al. 2017], which control individual time slices and can be delegated, with a static limit on delegations. A user-level scheduler is up-called for replenishment once the capability is depleted. Slack re-use is only possible by explicitly delegating the slack. A notion of *time quality* supports delegation across hierarchically-scheduled subsystems without explicitly mapping all local priorities onto a single, global priority space. Composite's time capabilities cannot be revoked unless empty. Efficient support for revocation tends to be one of the core challenges in capability systems and our scCaps, like all seL4 capabilities, can be revoked at any time.

Lackorzyński et al. [2012] first introduced a notion of scheduling contexts into a member of the L4 microkernel family, the Fiasco.OC microkernel. While Fiasco.OC uses capability-based access control, its SCs are not capability-controlled and thus not integrated into the access-control model. The model forces the use of bandwidth inheritance, a policy we aim to keep out of the kernel. Fiasco.OC's SC implementation lives in a branch of the kernel.

Völp et al. [2013] report on mixed-criticality scheduling experiments in Fiasco.OC, but without showing performance data. Their approach associates priorities with scheduling contexts and forces the use of priority inheritance for critical sections. Quest-V [Li et al. 2014] is a multi-criticality separation kernel where criticalities are partitioned across cores. It does not support sharing at all. PikeOS [Kaiser and Wagner 2007], a commercial separation kernel from the L4 family, also assigns criticalities to cores. AUTOBEST [Zuepke et al. 2015] is another separation kernel where the authors demonstrate implementations of AUTOSAR and ARINC653 in separate partitions. Zero-slack scheduling [de Niz et al. 2009, 2012] takes the opposite approach to reservations for mixed-criticality: a timeout is set for the last moment a high criticality task can run and meet its deadline, and earlier in high mode.

Real-time Mach [Mercer et al. 1994, 1993] first introduced scheduling contexts (called processor capacity reserves); their donation over IPC is compulsory, while reserves themselves are optional. Admission is done in the kernel and PIP is used to avoid priority inversion. EROS [Shapiro et al. 1999] follows the Mach scheme.

NOVA [Steinberg and Kauer 2010] provides capabilities to scheduling contexts, however the scheduling context provides access to a timeslice with no period, and thus does not provide temporal isolation. NOVA supports SC donation over IPC but servers can pick the scheduling context on which to run, and implements bandwidth inheritance in this fashion [Steinberg et al. 2010]. Inheritance schemes lead to complex dependency chains and make clients pay for the mistakes of others (as explained in Section 3.5).

Minix 3 [Herder et al. 2006] provides kernel mechanisms for implementing bandwidth servers at user level, however is not targeted at hard real-time systems and has no notion of criticality. Barrelfish [Peter et al. 2010] is capability-based and provides an RBED implementation for scheduling, but aligns criticality with time sensitivity. Nemesis [Leslie et al. 1996] was a single-address-space microkernel designed for multimedia performance, however this architecture is not suitable for mixed-criticality systems as all the code must be at the highest criticality.

Vanga et al. [2017] use EDF in priority bands of the fixed-priority scheduler of PikeOS, in order to run low-latency, low-criticality tasks on the same processor as high-criticality tasks in different partitions with bounded interference. The EDF tasks are bounded by a pluggable reservation algorithm, and run along side system services in the main partition, separate from high-criticality tasks. However, their use case does not involve sharing resources between tasks of different criticality.

7 CONCLUSIONS AND FUTURE WORK

Spatial isolation has long been a core focus of operating systems, culminating in its formal verification in seL4 [Klein et al. 2014]. Mixed-criticality systems, as they are emerging from the cyber-physical systems domain, require equally strong temporal isolation, including a similar level of assurance. The challenge is the non-fungible nature of time, and the need to share critical sections between threads of different criticalities, without making critical threads dependent on less critical ones.

We present simple, yet powerful mechanisms that provide capability-based control over time, supporting strong temporal protection even in the presence of resource sharing. Our implementation of these mechanisms in seL4 show that they can achieve low overheads and policy flexibility. With this new model, we also solve the long-standing issue of a lack of a satisfactory abstraction of time in L4 kernels [Heiser and Elphinstone 2016].

The new kernel is publicly released as open source, presently as a branch of the baseline kernel, and is already being deployed in several critical systems. Its functionality has been formally specified in a branch of seL4's formal verification framework, and verification of the code base is in progress. It will become the mainline kernel once verification is completed. All code is open source and accessible via https://github.com/pingerino/eurosys18.

ACKNOWLEDGMENTS

We thank Mark Miller (Google) for the real-estate analogy and especially for pointing to the confused-deputy problem in our initial MCP model. We further thank Luke Mondy for extremely last-minute help benchmarking LITMUS^{RT}.

This material is based on research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) BAA HSHQDC-14-R-B00016, and the Government of United Kingdom of Great Britain and the Government of Canada via contract number D15PC00223.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security, the U.S. Government, or the Government of United Kingdom of Great Britain or the Government of Canada.

REFERENCES

Luca Abeni and Giorgio Buttazzo. 2004. Resource Reservation in Dynamic Real-Time Systems. *Real-Time Systems* 27, 2 (2004), 123–167.

ARINC 2012. *Avionics Application Software Standard Interface.* ARINC. ARINC Standard 653.

Sanroy K. Baruah, Alan Burns, and Rob I. Davis. 2011. Response-Time Analysis for Mixed Criticality Systems. In *IEEE Real-Time Systems Symposium.* IEEE Computer Society, Vienna, Austria, 34–43.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. 2011. Timing Analysis of a Protected Operating System Kernel. In *IEEE Real-Time Systems Symposium.* IEEE Computer Society, Vienna, Austria, 339–348.

Bernard Blackham, Yao Shi, and Gernot Heiser. 2012. Improving Interrupt Response Time in a Verifiable Protected Microkernel. In *EuroSys Conference*. USENIX Association, Bern, Switzerland, 323–336.

Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. 1992. The KeyKOS Nanokernel Architecture. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*. USENIX Association, Seattle, WA, US, 95–112.

Björn B. Brandenburg. 2014. A Synchronous IPC Protocol for Predictable Access to Shared Resources in Mixed-Criticality Systems. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Rome, IT, 196–206.

Björn B. Brandenburg and James H. Anderson. 2007. Feather-Trace: A Light Weight Event Tracing Toolkit. In *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*. IEEE Computer Society, Pisa, IT, 61–70.

Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. 2003. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Cancun, MX, 396–407.

Manfred Broy, Ingolf H. Krüger, Alexander Pretschner, and Christian Salzman. 2007. Engineering Automotive Software. *Proc. IEEE* 95 (2007), 356–373.

Giorgio C. Buttazzo. 2005. Rate monotonic vs. EDF: judgment day. *Real-Time Systems* 29, 1 (2005), 5–26.

John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Rio de Janiero, Brazil, 111–123.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*. ACM, Indianapolis, IN, US, 143–154.

Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. 2009. On the scheduling of mixed-criticality real-time task sets. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Washington DC, US, 291–300.

Dionisio de Niz, Lutz Wrage, Nathaniel Storer, Anthony Rowe, and Ragunathan Rajkumar. 2012. On resource overbooking in an unmanned aerial vehicle. In *International Conference on Cyber-Physical Systems*. IEEE Computer Society, Washington DC, US, 97–106.

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9 (1966), 143–155.

Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. 2008. Kernel design for isolation and assurance of physical memory. In *1st Workshop on Isolation and Integration in Embedded Systems*. ACM, Glasgow, UK, 35–40.

Paul Emberson, Roger Stafford, and Robert Davis. 2010. Techniques for the Synthesis of Multiprocessor Tasksets. In *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. Euromicro, Brussels, Belgium, 6–11.

Bryan Ford and Jay Lepreau. 1994. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the 1994 Winter USENIX Technical Conference*. USENIX Association, Berkeley, CA, USA, 97–114.

The NetBSD Foundation. 2018. NetBSD. (2018). https://www.netbsd.org

Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. 1999. The Pebble Component-Based Operating System. In *Proceedings of the 1999 USENIX Annual Technical Conference*. USENIX Association, Monterey, CA, USA, 267–282.

Phani Kishore Gadepalli, Robert Gifford, Lucas Baier, Michael Kelly, and Gabriel Parmer. 2017. Temporal capabilities: Access Control for Time. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*. IEEE Computer Society, Paris, France, 56–67.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Savannah, GA, US, 653–669.

Mathhew R Guthaus, Jeffrey S. Reingenberg, Dan Ernst, Todd M. Austing, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*. IEEE Computer Society, Washington, DC, USA, 3–14.

Norm Hardy. 1988. The Confused Deputy (or Why Capabilities Might Have Been Invented). *ACM Operating Systems Review* 22, 4 (1988), 36–38.

Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems* 34, 1 (April 2016), 1:1–1:29.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *ACM Operating Systems Review* 40, 3 (July 2006), 80–89.

Robert Kaiser and Stephen Wagner. 2007. Evolution of the PikeOS Microkernel. In *International Workshop on Microkernels for Embedded Systems*. NICTA, Sydney, AU, 50–57.

Antti Kantee and Justin Cormack. 2014. Rump Kernels: No OS? No Problem! *USENIX ;login:* 29, 5 (Oct. 2014), 11–17.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, et al. 2009. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles*. ACM, Big Sky, MT, US, 207–220.

Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. 2012. Flattening Hierarchical Scheduling. In *International Conference on Embedded Software*. USENIX Association, Tampere, SF, 93–102.

John P. Lehoczky, Lui Sha, and Jay K. Strosnider. 1987. Enhanced Aperiodic Responsiveness in Hard-Real-Time Environments. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, San Jose, 261–270.

Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. 1996. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications* 14 (1996), 1280–1297.

Ye Li, Richard West, Zhuoqun Cheng, and Eric Missimer. 2014. Predictable Communication and Migration in the Quest-V Separation Kernel. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Rome, Italy, 272–283.

Jochen Liedtke. 1996. *L4 Reference Manual*. GMD/IBM.

C. Liu and J. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. ACM* 20 (1973), 46–61.

Cliff Mercer, Raguanthan Rajkumar, and Jim Zelenka. 1994. Temporal Protection in Real-Time Operating Systems. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*. IEEE Computer Society, San Juan, Puerto Rico, 79–83.

Cliff Mercer, Stefan Savage, and Hideyuki Tokuda. 1993. Processor Capacity Reserves: An abstraction for managing processor usage. In *Workshop on Workstation Operating Systems*. IEEE Computer Society, Napa, CA, US, 129–134.

Shuichi Oikawa and Ragunathan Rajkumar. 1998. Linux/RK: A portable resource kernel in Linux. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Madrid, Spain, 111–120.

Gabriel Parmer. 2010. The Case for Thread Migration: Predictable IPC in a Customizable and Reliable OS. In *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*. ACM, Brussels, BE, 91–100.

Gabriel Parmer and Richard West. 2011. HiRes: A System for Predictable Hierarchical Resource Management. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, Washington, DC, US, 180–190.

Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. 2010. Design principles for end-to-end multicore schedulers. In *Workshop on Hot Topics in Parallelism*. USENIX Association, Berkeley, CA, USA, 10–10.

RedisLabs. 2009. Redis. (2009). https://redis.io

Thomas Sewell, Felix Kam, and Gernot Heiser. 2017. High-Assurance Timing Analysis for a High-Assurance Real-Time OS. *Real-Time Systems* 53 (Sept. 2017), 812–853.

Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.* 39, 9 (Sept. 1990), 1175–1185.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *ACM Symposium on Operating Systems Principles*. ACM, Charleston, SC, USA, 170–185.

Brinkley Sprunt, Lui Sha, and John K. Lehoczky. 1989. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems* 1, 1 (June 1989), 27–60.

Mark J. Stanovic, Theodore P. Baker, An-I Wang, and Michael González Harbour. 2010. Defects of the POSIX Sporadic Server and How to Correct Them. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, Stockholm, 35–45.

Udo Steinberg, Alexander Böttcher, and Bernhard Kauer. 2010. Timeslice Donation in Component-Based Systems. In *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT)*. IEEE Computer Society, Brussels, BE, 16–22.

Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th EuroSys Conference*. ACM, Paris, FR, 209–222.

Manohar Vanga, Andrea Bastoni, Henrik Theiling, and Björn B. Brandenburg. 2017. Supporting low-latency, low-criticality tasks in a certified mixed-criticality OS. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS)*. ACM, Grenoble, France, 227–236.

Paulo Esteves Veríssimo, Nuno Ferreira Neves, and Miguel Pupo Correia. 2003. Intrusion-Tolerant Architectures: Concepts and Design. In *Architecting Dependable Systems*. Lecture Notes in Computer Science, Vol. 2677. Springer, Berlin, Heidelberg, 3–36.

Marcus Völp, Adam Lackorzynski, and Hermann Härtig. 2013. On the Expressiveness of Fixed-Priority Scheduling Contexts for Mixed-Criticality Scheduling. In *Proceedings of the 1st Workshop on Mixed Criticality Systems*. IEEE Computer Society, Vancouver, Canada, 13–18.

David A. Wheeler. 2001. SLOCCount. http://www.dwheeler.com/sloccount/. (2001).

Ian Wienand and Luke Macpherson. 2004. ipbench: A Framework for Distributed Network Benchmarking. In *AUUG Winter Conference*. AUUG, Melbourne, Australia,

163–170.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3 (2008), 1–53.

Alexander Zuepke, Marc Bommert, and Daniel Lohmann. 2015. AUTOBEST: a united AUTOSAR-OS and ARINC 653 kernel. In *IEEE Real-Time Systems Symposium*. IEEE Computer Society, San Antonio, Texas, 133–144.