SCHOOL OF ELECTRICAL ENGINEERING
AND TELECOMMUNICATIONS

User-Level Mixed Criticality Systems Scheduling on Multicore Processors

by

Sebastian Holzapfel

Student ID: z5060836

Thesis submitted as a requirement for the degree

Bachelor of Engineering (Electrical Engineering)

ELEC4121 Thesis B

Submitted: October 25, 2018

Supervisors: Gernot Heiser & Alex von Brasch

Abstract

This thesis investigates the feasibility of adapting a modern multicore mixed-criticality scheduling architecture & resource sharing protocol (previously only evaluated on monolithic kernels), to run at user-level on a high-assurance microkernel. Such an architecture has advantages in terms of flexibility and integrity when compared to more conventional approaches, and remains unexplored in literature – likely due to the absence of a microkernel able to facilitate it until recently. To that end, in this work we construct a multicore user-level scheduler & associated mixed criticality resource sharing protocol on a microkernel. We implement tracing infrastructure to evaluate the system, and investigate properties of our scheduler with respect to similar-class monolithic systems. Finally, we propose changes to the kernel API to improve user-level scheduling performance & discover that our scheduler is performance-competitive with a Linux-based monolithic approach.

Abbreviations

OS Operating System

seL4 Secure Embedded L4 - a secure, high performance microkernel

MCS Mixed Criticality System

SRT Soft Real-Time

HRT Hard Real-Time

SMP Symmetric Multiprocessing

WCET Worst-Case Execution Time

IPC Inter-Process Communication

MC-IPC Mixed-Criticality IPC

API Application Programming Interface

EDF Earliest Deadline First - a real time scheduling policy

P-EDF Partitioned EDF - one EDF scheduler per processor core

G-EDF Global EDF - one EDF scheduler shared by all processor cores

C-EDF Clustered EDF - hybrid of P-EDF and G-EDF

SC Scheduling Context

RMPA Rate-Monotonic Priority Assignment

TCB Thread Control Block

IPI Inter Processor Interrupt

Contents

List of Figures

List of Tables

1 | Introduction

Modern cyber-physical systems (such as autonomous vehicles) are normally constructed using physically isolated processors for fault containment. Such an arrangement ensures that low criticality software cannot interfere with high criticality components (with potentially life-threatening failure consequences). Unfortunately, physically isolated hardware is expensive, results in low per-CPU utilisation, and higher power draw.

Recent developments in operating systems have heralded methods for ensuring the required level of isolation between software components on a single processor. Combining high- and low-criticality software components in this fashion forms a *mixed-criticality system* (MCS), which overcomes the disadvantages of traditional hardware isolation. Multicore processors are of particular interest in this context, as they facilitate greater density of compute power per watt (and per unit size) than previously possible with similar-class unicore processors.

State-of-the-art multicore mixed-criticality scheduling & resource sharing techniques are predominantly built on monolithic kernels – extremely complex pieces of software consisting of millions of lines of code (for example, `PREEMPT_RT Linux`) which must be trusted for correctness if isolation enforcement is required. Fortunately, recent work in microkernel design (particularly [Lyons et al., 2018]) has resulted in the construction of operating system kernels that are both high-assurance (through in-progress formal verification), and provide mechanisms for constructing mixed-criticality systems at user-level (see chapter 2 for more information on monolithic & microkernels).

This thesis investigates the feasibility of adapting a modern multicore mixed-criticality scheduling architecture & resource sharing protocol (previously only evaluated on monolithic kernels), to run at user-level on a high-assurance microkernel, seL4. Such an architecture remains unexplored in literature – likely due to the absence of a microkernel able to facilitate it until recently [Lyons et al., 2018].

1.1 Motivation

There are four key reasons microkernel-based user-level scheduling is appropriate for mixed-criticality systems, and worth exploring further:

1. **Flexibility:** There is a significant lag behind MCS scheduling theory of MCS scheduling practice in industry [Mollison and Anderson, 2013]. Customization of scheduling behaviour at user-level allows system designers to more easily use cutting-edge scheduling techniques, and without compromising high-criticality schedulers, by compartmentalising them (or using an in-kernel scheduler).

2. **Performance:** Real-time schedulers in monolithic kernels are generally complex [Calandrino et al., 2006], and thus incur high context-switching overhead. This thesis will investigate whether the comparatively small context-switching cost of fast microkernels outperform monolithic kernels, even if user-level scheduling logic is in-between.

3. **Kernel Investigation:** Multicore MCS schedulers have not yet been evaluated at user-level on microkernels in literature [Burns and Davis, 2018]. As a result, kernel design trade-offs with respect to this use-case remain unexplored.

4. **Assurance:** Since modern microkernels are constructed from small (in rare cases mathematically-verified) codebases, system designers can rely on the trustworthiness of a modern microkernel to enforce user-level isolation properties.

1.2 Thesis Outline

We seek to demonstrate that user-level mixed-criticality schedulers deployed on a multicore, microkernel-based system, are in fact performance competitive with the same class of system constructed with a monolithic kernel. To that end, we construct a clustered multicore user-level scheduler & associated mixed criticality resource sharing protocol on a high-assurance microkernel, and investigate its properties with respect to similar-class monolithic systems.

1.2.1 Contributions

The key contributions of this thesis are as such:

1. An implementation & performance investigation of a microkernel-based multicore hard-realtime user-level scheduler with mixed-criticality support.

2. A tool for tracing of user-level schedulers which allows the collection of performance metrics and correctness testing of a scheduler implementation.

3. Test implementations & proposed changes to said microkernel's API for improving user-level scheduling performance.

1.2.2 Structure

Chapter 2 provides an introduction to real-time and operating systems basics. **Chapter 3** explores different ways of constructing multicore mixed-criticality systems as published in existing literature. **Chapter 4** explains a relevant subset of the seL4 programming model and kernel architecture. **Chapter 5** provides a high-level overview of how we construct & evaluate multicore mixed-criticality systems on seL4. **Chapter 6** explores our implementation in detail. **Chapter 7** covers our results, and our interpretation of them. **Chapter 8 & Chapter 9** provide an outline of possible future work, and our conclusion.

2 | Background

This chapter outlines real-time theory and operating systems concepts as they relate to this thesis.

2.1 Operating Systems & Microkernels

An *operating system* (OS) is a piece of software which interfaces with physical hardware and peripherals to provide services to applications through a common interface. At the core of an OS is the *kernel*, software which has privileged access to the CPU and manages resources at the lowest level.

Many kernels that underpin commodity operating systems (such as Android or Windows) can be broadly described as *monolithic kernels*[1]. In our context *monolithic* indicates that many of the services that the operating system provides to applications exist as part of the kernel (for example device drivers or filesystem implementations). This lies in contrast to the design goals of *microkernels* (such as seL4), which attempt to reduce the kernel size (hence amount of privileged code in a complete system) as much as practically possible. Liedtke [1995] described the microkernel *minimality principle* – a concept which still underpins modern microkernel design [Heiser and Elphinstone, 2016]:

> A concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

Device drivers, memory management, filesystems & schedulers (which are the focus of this thesis) – all conventionally components of monolithic kernels; can exist as user-level components in modern microkernels. See Figure 2.1 for a comparison.

[1] The Windows NT kernel is sometimes denoted a *hybrid* kernel, but it is definitely not a microkernel. Take 'broadly' with a grain of salt

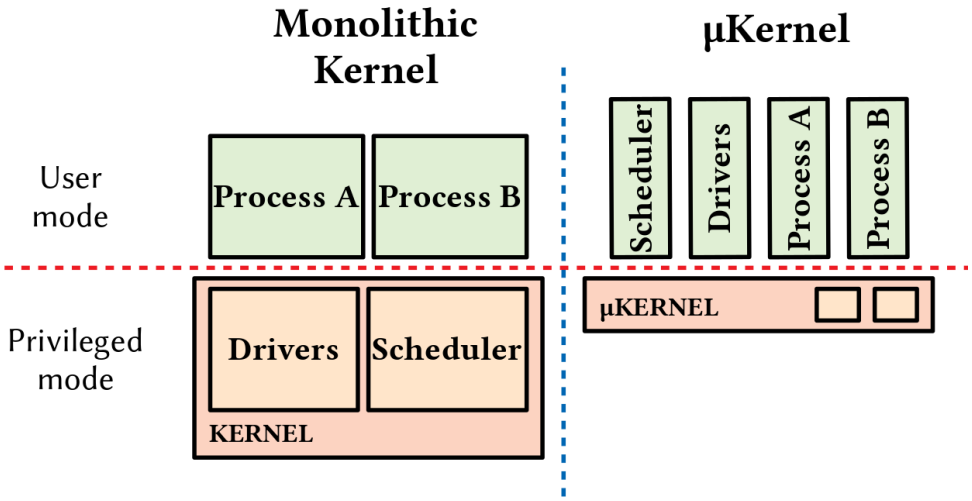

Figure 2.1: Monolithic and microkernel-based systems

Placing key operating system services (such as device drivers) at user-level allows for graceful recovery in the event of faults. In a monolithic kernel, if a security vulnerability in a particular driver is exploited, it may be possible to (at best) crash the kernel or (at worst) gain control of the whole system & steal information. In a well-constructed microkernel-based system, exploitation of vulnerabilities in a particular driver are contained – the driver may crash (in which case the driver could be restarted), or the driver could be otherwise compromised (effecting only the driver and perhaps any other user-level services it has permissions to). For these reasons, smaller kernel size inherently decreases our *trusted computing base* [Rushby, 1981], which is the combination of a kernel and processes whose correctness must be ensured to protect a system from crashes or security vulnerabilities. In the context of high-assurance systems, the reduced trusted computing base and generality of microkernel mechanisms provides us with a strong foundation for building more complex systems.

2.1.1 seL4

seL4 is a high-performance, formally verified microkernel which recently gained support for mixed-criticality mechanisms [Lyons et al., 2018]. A formal, machine-checked proof exists which states that if the kernel is initialized correctly, then the kernel binary will not deviate in behaviour from an abstract formal specification [Klein et al., 2014]. Although such a proof for the recently-added mixed-criticality mechanisms is still in development, it is expected to be completed in the near future. An evaluation of the properties of complex user-level

arrangements which use these MCS primitives is an area unexplored – which is partially why we chose seL4 as an experimental platform upon which to perform our research in later sections. Further details on the specifics of seL4 will be covered in chapter 4, where we describe a subset of the seL4 programming model and kernel architecture that is most relevant to this research.

2.2 Real-time systems

Much of mixed-criticality systems scheduling work stems from 'ordinary' real-time theory. Consequently this section introduces some real-time basics. For a comprehensive review of real-time models, scheduling and locking protocols, the reader is encouraged to explore the work by Brandenburg [2011].

In the context of real-time systems, a *task* denotes a unit of execution that has some temporal constraint. In practical systems, there is often a direct correspondence between tasks and threads. Mok [1983] describes the *sporadic task model*, under which real-time tasks are characterised by 3 parameters – (e_i, p_i, d_i):

- p_i denotes the *minimum inter-arrival time*, i.e the minimum period between successive arrivals of jobs for a particular task.
- e_i denotes the *maximum execution requirement*, often determined by estimating or statically computing the worst-case execution time (WCET) of a job.
- d_i denotes the *deadline* of a job – at least e_i units of execution time must be allocated to a job before its deadline, for the deadline to be met.

Often, the deadline of a task is simply 'before the next job release' – i.e task deadlines and periods match ($p_i = d_i$) – allowing representation of real-time tasks in terms of 2 parameters only (e_i, p_i). This is denoted an *implicit* deadline task (in contrast to *explicit* deadline tasks) (we will later discover that implicit deadline specification is roughly how real-time tasks are denoted in seL4).

In terms of temporal correctness, deadline misses are not tolerated in so-called *hard real-time* (HRT) systems, whereas (bounded) lateness is permitted in *soft real-time* (SRT) systems. Tasks which have no HRT or SRT requirements are often denoted *best-effort* tasks [Brandenburg, 2011].

2.3 Mixed-criticality systems

Mixed-criticality systems extend the real-time models with additional constraints. We will start with a generic mixed criticality system model (based on work by Burns and Davis [2018]), and then briefly look at more detailed models.

A mixed criticality system generally consists of a set of components. Each component has a *criticality level*, and contains a set of sporadic tasks. The component criticality level indicates how important a component is to the system's functional correctness. For example, the stereo software in a car is less relevant to the vehicle's functional correctness than the engine management unit. It is a requirement in mixed-criticality systems that low criticality components *do not interfere* with high-criticality components. The non-interference property is thus asymmetric - high-criticality components may interfere with low-criticality components, but not vice-versa. This means, for example, that components must not use more than their assigned computation time, and resource sharing across criticalities is either carefully controlled or forbidden. To enforce such a constraint, temporal & spatial isolation are required. Systems which rely on the correctness of components (rather than the operating system or user-level scheduler) for isolation require low-criticality tasks to be engineered to the same degree of rigour as high-criticality tasks. This is undesirable – constructing software to meet (for example) high-criticality certification requirements is expensive and requires huge amounts of engineering work.

2.3.1 Criticality modes

Many works on MCS consider *criticality modes* [Burns and Davis, 2018]. In a system with criticality modes, only components with criticality greater than a particular criticality level (indicated by the current mode) receive CPU time. Initially, the system starts at the lowest criticality mode. If any job attempts to execute for longer than e_i, the criticality mode is raised. In some cases, systems only have 2 criticality levels, commonly called LO and HI. In binary-criticality systems with criticality modes, the system is said to be in either LO-criticality or HI-criticality mode, allowing all components, or only high components to execute respectively.

Vestal [2007] observed that it is often desirable for e_i (WCET) to have different values in different criticality modes. For example, if we had a very pessimistic WCET estimate (for safety) and a more realistic WCET estimate, it is desirable to run the component using the realistic

WCET estimate (to allow more time for low priority components to run) when possible. If the realistic WCET estimate is violated, then the system raises the criticality mode and all components are scheduled based on pessimistic WCET estimates.

Now that we have explored the basics of real-time & mixed-criticality systems, we will explore how real-time systems can be scheduled, and later extend this to mixed-criticality systems.

2.4 Schedulers

A *scheduler* is the part of an operating system that determines when execution abstractions (such as processes, threads, or real-time tasks) are allocated physical CPU time.

2.4.1 Uniprocessor real-time schedulers

Real-time schedulers can be classified by how jobs (which are released by tasks) are prioritized over time according to some *prioritization function*. When jobs from a particular task always have the same scheduling priority, this class of schedulers is *fixed priority* – explored further below. There also exist real-time schedulers in which the priority of any one job remains constant, but different jobs from the same tasks may have different priorities – this is called a *job-level fixed priority* scheduler (of which an EDF scheduler is an example, see below). Fully dynamic schedulers also exist, but they can have unpredictable real-time behaviour and suffer from more preemptions than the above two approaches in the general case [Brandenburg, 2011], so we omit them here.

Fixed priority schedulers

Rate monotonic priority assignment (RMPA) is a classic way of allocating priorities for a fixed-priority scheduler. Under RMPA, each task is assigned a fixed priority directly proportional to its frequency (lowest inter-arrival time (period) is of the highest priority). If tasks are independent, then RMPA guarantees that tasks will meet their deadlines if the sum of task utilisations (i.e $\frac{e_i}{p_i}$) is below some upper bound (0.69 in the limit of infinite tasks).

In the context of a mixed-criticality system however, the priority of a task does not necessarily correspond to its criticality level. RMPA only guarantees schedulability if all tasks meet the deadlines specified during system design, which implies trust in low-criticality tasks

if, for example, such a low-criticality task has a high scheduling priority due to RMPA. A simple way to solve this in the fixed-priority case is *period transformation* [Sha, 1986]. Under period transformation, both e_i and p_i of high criticality tasks are split successively into parts of parameters $\frac{e_i}{2}$ and $\frac{p_i}{2}$ (i.e the jobs are interleaved), until the final priority assignment matches criticality levels. The resulting priority assignment ensures that high-criticality tasks will not miss deadlines due to interfering low-criticality tasks, eliminating the need for trust in low-criticality tasks (as long as tasks do not share resources across criticalities). Unfortunately this approach suffers from unnecessarily large context switching overhead, and a developer must figure out how to accurately split up a potentially complex task into separately schedulable subtasks.

Earliest Deadline First (EDF) schedulers

Under uniprocessor EDF, a job with the earliest deadline always possesses the highest scheduling priority. A key advantage of EDF (a job-level fixed priority scheduler) over more classic fixed-priority schedulers (such as RMPA) is higher schedulability. In fact, EDF is *optimal* in the uniprocessor case, meaning it has a utilisation upper bound of 1 (ignoring overheads of the scheduler itself).

Unfortunately, since jobs are prioritized at runtime, it becomes difficult to reason about how the system might behave in a mixed criticality scenario if a job overruns its deadline. Baruah et al. [2011] proposed a solution to this, *EDF-VD* (EDF-Virtual-Deadline), under which high criticality tasks have their EDF deadlines reduced under certain circumstances so that high-criticality tasks are always scheduled when required. Many such modifications to EDF for MCS exist [Burns and Davis, 2018].

2.4.2 Sporadic Servers

A *server* (in the context of real-time) is an algorithm for enforcing an execution upper bound. The *sporadic server model* [Sprunt et al., 1989] is a scheduling policy for budget enforcement that is compatible with other real-time scheduling algorithms (particularly fixed-priority). It is of particular relevance to mixed-criticality systems, as sporadic servers provide a principled way of enforcing an upper bound on thread execution time at high priorities. We will later discover that maximum execution time enforcement in seL4 is based on sporadic servers. Sporadic servers preserve a *sliding window constraint*. Using the parameters e_i and p_i of an

implicit-deadline task, the sliding window constraint is that during any p_i window of time, no more than e_i execution time can be consumed. This constraint is maintained by keeping track of *budget replenishments* – partial budget consumption is tracked, and later restored at p_i units of time in the future. In the POSIX case, a task which overruns its budget under `SCHED_SPORADIC` has its priority reassigned (to a background priority) until its budget is replenished. For more information on sporadic server algorithms, see Stanovic et al. [2010].

2.4.3 Multiprocessor real-time schedulers

Partitioned, global & clustered real-time scheduling

The question remains as to how uniprocessor scheduling algorithms might be extended to multicore scenarios.

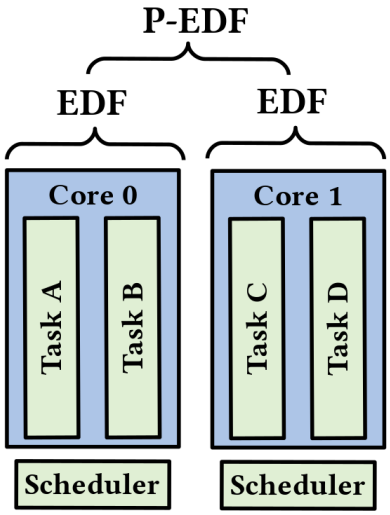

Figure 2.2: A partitioned scheduler (EDF)

The simple case is *partitioned* scheduling, in which each processor is scheduled independently (this has the advantage that we can just use existing uniprocessor analysis on each core). Partitioned scheduling is the scheduling approach most often used in practice, due to ease of analysis, and has been demonstrated preferable in the hard real-time case [Brandenburg, 2011]. The downside of partitioned scheduling is low utilisation – tasks must be statically assigned to partitions at design time such that no processors are overloaded. For example, if we had 3 tasks of utilisation 0.6 (utilisation sum 1.8), this would not be schedulable under partitioned scheduling on 2 cores, despite the utilisation sum being less than 2. Heuristics exist

for computing solutions to this *bin-packing* problem, however we deem partition assignment a system design issue (and out of scope for this thesis).

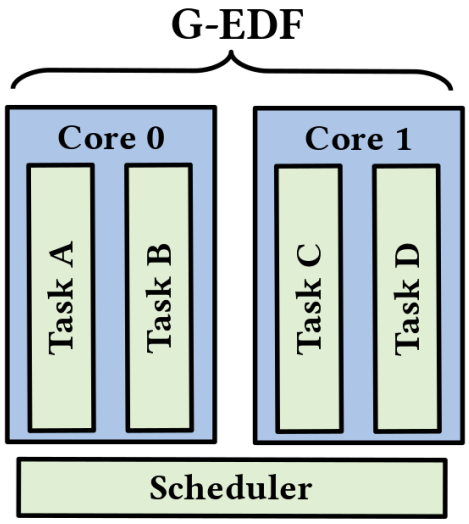

Figure 2.3: A global scheduler (EDF)

Global scheduling takes a different approach, with one (global) scheduler which exercises control over all processors. Since all processors share a single logical ready queue, partitioning is not required and hence global scheduling does not suffer from low utilisation due to bin-packing problems. Unfortunately, the high overheads of frequent cross-core communication (and context migration) associated with global scheduling reduces available scheduling capacity [Brandenburg, 2011].

Clustered scheduling is a generalisation of partitioned and global scheduling, where *clusters* of processors are scheduled by a single logical scheduler per cluster. A clustered scheduler with cluster size 1 is a partitioned scheduler, whereas a clustered scheduler with cluster size equal to the core count is a global scheduler. We will examine some design issues with respect to clustered schedulers in following chapters. Brandenburg [2011] demonstrated that in the general case, the ability for clustered scheduling to trade off between bin-packing partitions and global scheduling communication overhead makes it desirable for soft real-time systems.

Multicore scheduling and cross-core interference

It is possible for tasks on different cores to interfere with each other even if no resources are explicity shared between them. For example, if a low-criticality task has a high cache footprint and shares a cache with a high-criticality task, then the low-criticality task could adversely

influence the WCET of the high-criticality task (a threefold slowdown is demonstrated by Pellizzoni et al. [2010] in this situation). In addition to standard caching mechanisms, modern CPUs contain (in many cases undocumented) shared microarchitectural state, which can cause undesired cross-core interference. A naive solution might be to make task WCETs more pessimistic – unfortunately arriving at an accurate WCET estimate (by accounting for cross-core interference) is an active area of research, and requires deterministic CPU behaviour.

Yun et al. [2012] describe a memory throttling mechanism to mitigate cache-related cross-core interference, by measuring cache evictions and dequeueing offending tasks. Using cache evictions as a scheduling heuristic implies potentially intolerable reponse latency (for example, if a low-criticality task pathologically moves from no cache usage to cache thrashing just before a latency-sensitive critical interrupt arrives for a high-criticality task on another core).

Gang scheduling [G. Feitelson and Rudolph, 1992] is a scheduling approach where only the groups of threads associated with a single process are scheduled at any one time. If CPU state is flushed on every context switch, this reduces the possibility of cross-core interference between different processes (which may be of different criticalities in an MCS). Gang scheduling provides a performance improvement if threads of the same process communicate heavily, but suffers from low utilisation if threads do not saturate cores.

Although this thesis does not directly address the influence of cross-core microarchitectural effects on scheduler design, it is important to be aware of these concerns. Our scheduling architecture should not introduce more cross-core interference opportunities or unintended information sidechannels.

3 | Related Work

To investigate the feasibility of adapting a mixed-criticality scheduling architecture to run at user-level, we require three components – a real-time aware user-level scheduling architecture, a mixed-criticality resource sharing policy and a microkernel with user-level scheduling support. This chapter sequentially examines previous work in relation to each of these elements.

3.1 User-level real-time scheduling

In this section, we will first examine some uniprocessor architectures, move to multicore, and then evaluate whether they are applicable to our MCS use-case. *Scheduler activations* [Anderson et al., 1992] are a kernel construct which facilitates user-level scheduling by having the kernel notify user-level schedulers on a kernel scheduling event. Oikawa and Tokuda [1994] implement a unicore user-level real-time scheduler on the RT-Mach microkernel based on scheduler activations from RT-Mach for user-level scheduling events. Scheduler activations as a feature in monolithic kernels have slowly disappeared over time [Mollison and Anderson, 2013]. Anantaraman et al. [2004] built a user-level EDF scheduler on Linux, but do not address synchronisation or handling of Linux system calls from tasks (which is a key reason for using Linux in the first place). Unlike the above monolithic work, Ruocco [2006] implement a unicore user-level real-time scheduler on the L4::Pistachio microkernel for adaptive scheduling, with user-level interrupt handlers triggering scheduling events - an approach also possible under seL4. Huang et al. [2012] build a user-level fixed-priority mixed-criticality scheduler on Linux based on period transformation, but do not address synchronisation between tasks or multicore. Lyons et al. [2018] demonstrate a microkernel-based user-level EDF scheduler, but it was only evaluated in a unicore situation[1]. RASP [Mollison and Anderson, 2013] is a framework

[1] Lyons et al. [2018]'s scheduler was actually the basis for some of our early uniprocessor EDF prototypes in this thesis – we ended up almost entirely rewriting it. Still, some code remnants remain.

for preemptive multicore real-time user-level scheduling which addresses synchronisation between tasks and supports clustered scheduling. Since the design goals of RASP superficially map closely to this thesis, we shall explore its architecture in detail.

3.1.1 RASP – Real-time Application Scheduling Platform

Mollison and Anderson [2013] describe RASP – an open-source userspace library for multicore real-time scheduling, which uses the PREEMPT_RT Linux kernel as an underlying RTOS (whether it is portable to microkernels will be discussed in this section).

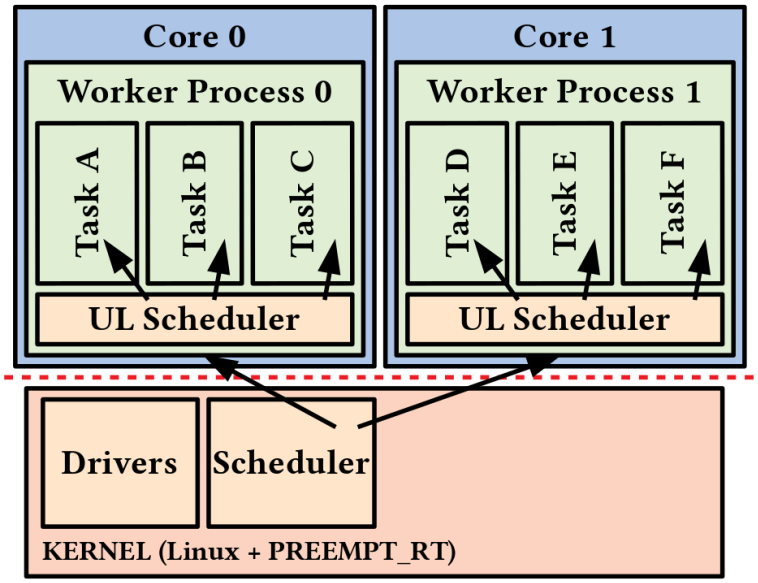

Figure 3.1: RASP architecture

A system based on RASP consists of a single kernel-level thread pinned to each CPU core, and a user-level thread for each real-time task (see Figure 3.1). The kernel-level threads can be conceptualised as 'virtual CPUs', where user-level threads (real-time tasks) execute within the appropriate kernel-level thread. C-EDF is the only scheduling algorithm available in RASP. For preemption, the library makes use of POSIX timers to interrupt real-time tasks, which signal the kernel-level threads to perform a scheduling operation. The kernel-level thread is then able to perform an appropriate scheduling operation, which could include communicating with other kernel-level threads if a cross-core migration is required. For synchronisation between tasks, RASP includes support for Flexible Multiprocessor Locking Protocol [Block et al., 2007], which is similar to ordinary semaphore-based locking protocol, but with some extra analysis and protocol complexity such that it has predictable behaviour on multicore.

PREEMPT_RT Linux is of course a monolithic kernel – a dependency on such a kernel requires a massive trusted computing base. That being said, there exist microkernel-based operating systems which present a POSIX-like interface (such as QNX), to which RASP could be ported. Even if RASP was ported to such an architecture (and we assumed the underlying kernel was correct), enforcing temporal isolation between untrusted tasks (with RASP in its current state) is not possible. RASP's reliance on conventional, lock-based synchronisation protocols means that all tasks must adhere to the locking protocol, else other tasks may starve (for example, one task locks a mutex, never releases it). This places trust in the implementation correctness of (potentially low-priority) real-time tasks, which is unacceptable in our case. From a performance perspective, any implementation of RASP on top of a microkernel requires a POSIX compliance layer – a layer which all RASP preemption/scheduling operations must traverse. Such a layer cannot be without overhead – which we quantify when comparing RASP to our approach in chapter 7. For the reasons described above, we deem RASP's architecture different enough to our research goals such that extending RASP to meet our goals would be impractical.

3.2 Mixed-criticality resource sharing

With respect to sharing resources cross-criticality, there are two dominant approaches [Burns and Davis, 2018] – extending conventional locking protocols, and encapsulating shared resources in resource servers (to be explained shortly). The main problem with conventional locking protocols is that locking implies trust across criticalities. If a low-criticality task locks a resource, a high-criticality task must trust that the lock is eventually released by that task (even if the low-criticality task was moved to the highest possible scheduling priority, we still have to trust its implementation). Resource servers on the other hand, when carefully implemented, do not suffer from this problem. Well-studied in microkernel literature, resource servers are threads which encapsulate shared resources. [Lyons et al., 2018] describe kernel mechanisms to support mixed-criticality resource servers, as well as some examples of how to build systems with resource servers. Essentially, real-time tasks are allocated an upper-bound on CPU bandwidth (using sporadic servers), and must 'donate' their time allocation to a resource server in order for the server to use it. If the bandwidth allocation is depleted (which can be indicated by a 'timeout fault'), the server may be rolled back or exhibit some other behaviour whenever

the next resource request arrives. However, the example systems in [Lyons et al., 2018] do not directly account for some (multicore) edge cases to do with non-reentrant servers and dynamic task numbers. We will explore this further by studying the work of [Brandenburg, 2014], below.

3.2.1 MC-IPC

[Brandenburg, 2014] describes MC-IPC – a multicore, mixed criticality resource sharing protocol based on resource servers and carefully defined IPC semantics. Since MC-IPC forms part of our implementation approach (chapter 5), we will spend some time studying it here. At a high level, MC-IPC employs a 3-level hierarchical queueing structure and scheduling reservations to enforce worst-case resource server invocation times. Although this protocol was initially evaluated as part of a monolithic kernel (LITMUS^{RT} [Calandrino et al., 2006]), Brandenburg denoted implementing MC-IPC in a commodity microkernel as future work. This was acknowledged in later work on mixed-criticality microkernel design [Lyons et al., 2018], however the complexity of MC-IPC made it undesirable to implement as a core microkernel mechanism (given the microkernel goal of policy freedom).

At the heart of MC-IPC's unique isolation properties is an invocation bounding theorem, which we restate here in very condensed form for convenience. Under clustered scheduling, if we take C as the cluster size, K as the number of clusters and L as the server worst-case budget consumption for a single request, the following theorem holds. For a highest-criticality task invoking a shared resource server, the reservation associated with the invoking task will consume at most $(1+2CK)\cdot L$ units of budget. A key observation is that unlike previous work on shared resource servers, the invocation bound presented by Brandenburg is (if an invoking task is of highest or equal-highest criticality):

1. Independent of the number of tasks in the system
2. Independent of the runtime behaviour of any other task in the system

These statements are quite broad, so let us consider some concrete examples by comparing this behaviour to conventional FIFO based and priority-queue (which we will denote PRIO) based L4-style inter-process communication (IPC). If we consider the first property with respect to FIFO-IPC, a swarm of low-criticality tasks could saturate a resource server's FIFO queue and the invocation bound experienced by a high-criticality task would increase. PRIO-IPC

(currently employed by seL4-MCS) does not suffer from this, however it does not completely solve the problem. Instead, considering the second property above with respect to PRIO-IPC, in a situation where a swarm of equally-high-criticality tasks malfunction and swamp the resource server we end up with a similar problem – the invocation bound experienced by a high-criticality task increases (an amount proportional to the number of interfering tasks). MC-IPC does not suffer from this class of problems by employing its 3-level queue structure with specific request pruning rules.

Despite the aforementioned strong invocation bound properties, MC-IPC is not without its limitations. Nested requests for resource servers are common in real systems – for example, if we wanted a resource server which performed sensor fusion, it may use a resource server that reads an accelerometer, which in turn requires another resource server that provides bus-level communication (e.g I^2C). This is a use-case that MC-IPC does not currently support, and systems using MC-IPC must therefore be constructed by putting more functionality in individual servers than would otherwise be necessary. Additionally, the protocol's analysis currently assumes non-reentrant servers – that is, a partially-executed resource request cannot be aborted (by, for example, deleting partial work and reverting the server to a good state). In cases where it makes sense for a server to be able to abort requests that have been partially executed, the worst-case invocation latency under MC-IPC may increase significantly, as this behaviour is not defined by the protocol. In comparison, seL4-MCS defines primitives that support nested server requests and reentrant servers (using scheduling context donation and timeout faults respectively). Unfortunately, a simple application of the mechanisms provided by the kernel in this case (for resource sharing) still suffers from PRIO-IPC invocation latency blowout in the event of malfunctioning high-criticality tasks (as described above). This is the key limitation that MC-IPC is able to address.

With respect to this thesis, MC-IPC remains of interest as it addresses an issue with using PRIO-IPC for resource sharing in multicore, mixed criticality situations. Additionally, a protocol whose performance relies deeply on IPC naturally lends itself to an efficient microkernel implementation – so it is of interest to see how easily it may be implemented using seL4 primitives. For these reasons, a user-level implementation of MC-IPC is presented in later sections as part of our experimental framework.

3.3 Kernels for multicore mixed-criticality scheduling

We established that seL4-MCS will be our target kernel, but have not yet justified the choice. Some isolation kernels with multicore support rely on hard partitioning across cores and eradication of shared resources between partitions (such as PikeOS [Burns and Davis, 2018]). We desire more flexibility in our case. Additionally, we limit our exploration of kernels to those which are open-source.

In the microkernel world, deferring scheduling to user-level allows for implementation of complex scheduling policies. Unlike earlier work on real-time user-level scheduling (such as [Stoess, 2007] in the L4 case), the COMPOSITE microkernel [Gadepalli et al., 2017] provides Temporal Capabilities, which facilitate temporal isolation between tasks. Unfortunately, as of the time of writing, these temporal capabilities have only been evaluated on unicore systems. Additionally, COMPOSITE does not possess a trusted in-kernel scheduler – implying that system designers are required to build a trusted user-level scheduler for high-criticality tasks. In contrast, seL4-MCS has multicore support, a trusted in-kernel scheduler and sophisticated temporal isolation mechanisms (which will be explored in detail in the next chapter).

LITMUS^{RT} [Calandrino et al., 2006] is a monolithic kernel based on Linux designed as a benchmarking framework for multicore scheduling algorithms. It includes (in-kernel) scheduler plugins for multicore real-time scheduling algorithms such as C-EDF, and modern mixed-criticality resource sharing protocols (including MC-IPC). LITMUS^{RT} does not directly support user-level schedulers, but should serve as a realistic monolithic comparison target against our implementation, given its Linux heritage.

Dual-kernel architectures such as Xenomai [Gerum, 2004] and Nizza [Hartig et al.] allow hard real-time scheduling of userspace tasks on a microkernel alongside a monolithic kernel (i.e Linux) on the same system, however modification of the real-time scheduler requires kernel modifications and the approach suffers from multicore scalability issues.

3.4 Summary

In this section we explored existing user-level scheduling approaches with real-time support, a mathematically-backed mixed-criticality resource sharing protocol, existing monolithic architectures with mixed-criticality support, and other kernels of interest. As we have observed, modern mixed-criticality schedulers (in particular multicore-applicable flexibly partitioned

architectures that support synchronisation) remain untested in a user-level context. In the next chapter, we explore the seL4 microkernel in detail as it forms the base of our implementation plan in chapter 4.

4 | seL4-MCS

This chapter introduces the principles of seL4-MCS – our implementation platform – published by Lyons et al. [2018] as an extension to previous work on seL4. We will begin by exploring the basics of resource management & permission control (in the form of seL4's capability system), as well as IPC (used for communication between user-level processes). Following this, the rest of the chapter will cover the seL4 scheduling model, how it can be used to construct user-level schedulers, and how mixed criticality systems can be constructed.

4.1 Capabilities

As seL4 is a capability-based kernel, it is essential to understand seL4's capability system before exploring the rest of the kernel. In the context of seL4, a capability represents an access right to a resource. On system initialisation, the first thread (or 'root task') is given *capabilities* to all available resources by the kernel. This collection of capabilities exist in the thread's *capability space* or `cspace`. In seL4, there is a direct correspondence between the abstract idea of a 'resource' and *kernel objects*. Generally, the concept of a resource is represented in userspace by capabilities to a kernel object, where the kernel object abstracts a physical resource or another type of OS primitive. As a concrete example, there exist capability types associated with mappable physical memory (to frame kernel objects), processor time (to scheduling context objects), and other basic OS primitives. Capabilities can be *invoked* to perform an operation on a resource, by executing a system call with the capability as an argument (more details in section 4.2).

4.1.1 Object types

- **Thread Control Blocks (TCBs)** represent threads in seL4. TCB capabilities can be invoked to start & stop threads, configure scheduling priorities, set up fault handlers

and other operations.

- **Scheduling Contexts (SCs)** represent processing time on a particular CPU in seL4. For each CPU core, the root task receives a `SchedControl` capability that enables the creation of SCs with allocations of CPU time for a particular core. More details on SCs, as well as the way that the kernel scheduler interacts with TCBs & SCs will be explored in section 4.3.

- **Endpoints** are objects which represent a synchronous communication channel between threads (IPC). An endpoint capability can be invoked to `Send()` information (or capabilities) to a second thread, assuming the second thread expects information over the endpoint (i.e `Recv()`). IPC is discussed further in section 4.2

- **Reply objects** serve two purposes. When (for example) a resource server is invoked by a client (i.e. `Call()`), reply objects are used to represent a single-use reply endpoint used to respond to the call by the server. Additionally, when calls such as this are nested, reply objects are used to keep track of scheduling context donation (more in section 4.3).

- **Notification objects** provide a method of asynchronous signalling between threads. They behave like a set of binary semaphores implemented using a bitfield. A notification can be signalled (bits set), polled (the state of the bitfield is checked), or waited on (a call does not return until any bits are set).

- **Interrupt objects** provide threads with the ability to subscribe to hardware interrupts, and acknowledge them after the interrupt is handled. Once an interrupt fires, the kernel will mask future occurrences of the same interrupt. Interrupt objects are usually associated with a notification object, so that the arrival of an interrupt will signal a notification. After an interrupt is handled, a thread may invoke the interrupt object to signal IRQ acknowledgement to the kernel - which unmasks it.

- **Untyped memory objects** represent physical memory that has not yet been converted into kernel objects. On system boot, most of the device's memory is represented as untyped memory objects. Untyped memory objects can be split into smaller untyped objects, and also 'retyped' into mappable frames or other types of kernel objects – this is the fundamental principle behind memory allocation in seL4.

Now that we have a basic understanding of capabilities and kernel objects, we will examine how a system can use these objects for communication between threads.

4.2 IPC & System Calls

seL4 provides a set of IPC system calls that facilitate three primary operations – communication between threads, transfer of capabilities between threads, and invocations on kernel objects using capabilities. IPC between threads is accomplished using capabilities to endpoint objects. If one or more threads hold capabilities to the same endpoint, then that group of threads may communicate or transfer capabilities with rules dictated by the rights associated with the capabilities. IPC in this manner is synchronous – that is, if a `Send()` or `Recv()` is attempted and another thread is not immediately available to service the request, the first thread blocks until the message can be transferred (See Figure 4.1).

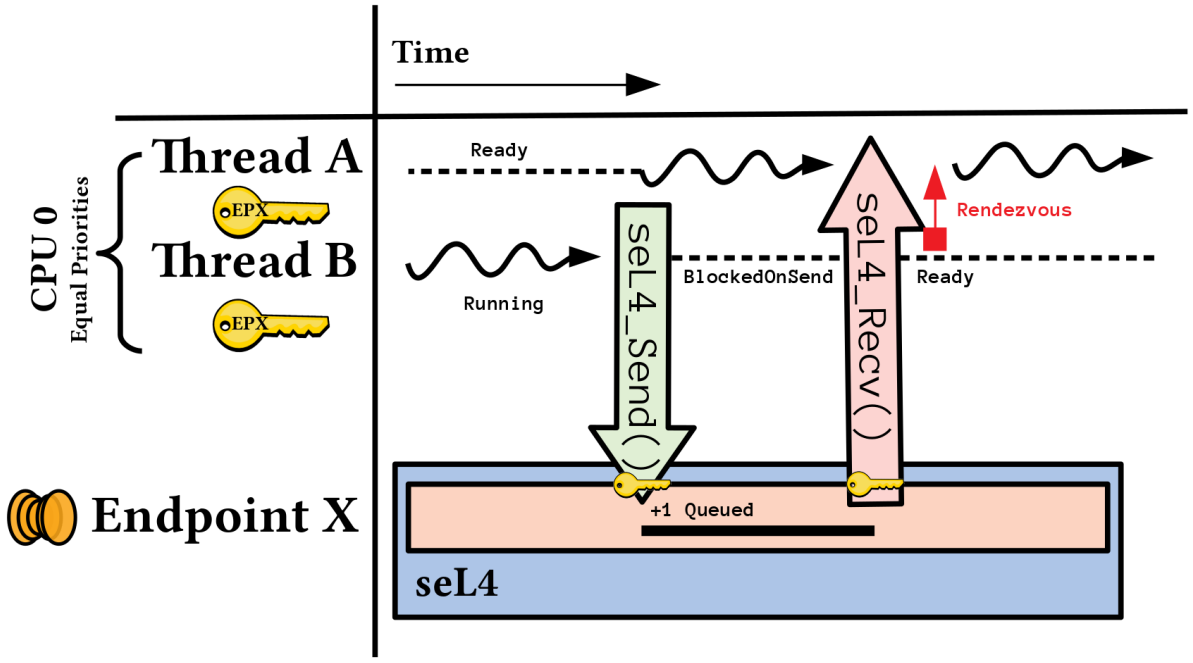

Figure 4.1: Example of an IPC rendezvous

If a thread wishes to perform some operation on a capabilities' underlying resource (for example, to convert some untyped memory into a kernel object), it must invoke the capability by performing an 'IPC' (`Send()` system call) with the target capability as an argument. If the operation is valid, the kernel will perform the operation requested on the thread's behalf, by

manipulating the underlying kernel object (the important distinction here being that the kernel itself is the logical destination of the `Send()`, rather than another thread as in ordinary IPC).

A subset of basic system calls associated with these IPC operations is listed below:

4.2.1 IPC System Calls

- **`seL4_Send()`** sends a message using a capability or performs a kernel invocation. If the capability is to an endpoint and no receivers are blocked on the endpoint, the sending thread blocks, awaiting message transfer.

- **`seL4_Recv()`** is used to receive messages. Similar to `seL4_Send()`, if no message is immediately available on an endpoint then the calling thread will block until a message arrives.

- **`seL4_Call()`** is an atomic combination of `seL4_Send()` and `seL4_Recv()`. A thread which performs this system call will block until a reply is received on the endpoint that the message was sent to. We may wish to have a server that can keep track of multiple clients, but not necessarily possess permanent send capabilities to each client – for this reason, a reply object is populated during the `seL4_Call()`. The server may reply using this single-use capability.

- **`seL4_Wait()`** is similar to `seL4_Recv`, but is designed for endpoints & notifications. Specifically, a reply object is not required (which means that an `seL4_Wait()` cannot be the destination of an `seL4_Call()`.

In addition to the above basic system calls, there are more variations which combine multiple calls into a single system call – as well as non-blocking variants (such as `seL4_NBSend`), which return immediately if a rendezvous between threads cannot happen. It is important to note that the last two system calls above are variants of the first two calls. In fact, the thus-far described list of system calls (if non-blocking and combination variants are included) actually represent *all* (bar one) of the system call classes available in seL4. The as-yet undescribed system call is `Yield()` (and friends), which we will come across in the next section.

4.3 seL4 scheduling model

seL4's top-level in-kernel scheduler is a preemptive fixed-priority scheduler with 256 priority levels[1]. The highest-priority runnable thread will always be selected by the scheduler (scheduling behaviour within a priority level is more complex, and will be discussed later). Superficially this may seem relatively straightforward as far as scheduler architecture goes – but the kernel provides mechanisms to allow for the creation of more complex scheduling behaviour. At the heart of these mechanisms are Scheduling Contexts.

4.3.1 Scheduling Contexts (SCs)

Scheduling Contexts (SCs) are kernel objects which represent CPU execution time. For an SC to represent execution time, a CPU bandwidth upper-bound is indicated by setting the SC *budget* (b) and *period* (p) properties. It is important to note that the b and p properties do not represent a minimum execution time guarantee. The key property that b and p *do* dictate is that the kernel *will not allow* execution on a particular scheduling context for more than b microseconds, every p microseconds. Budget enforcement (when $b < p$) is implemented by the kernel using *sporadic servers* (described in chapter 2, with a finite amount of partial budget replenishments dictated by the SC configuration). This hard temporal budget enforcement property that the kernel provides is essential for constructing user-level MCS & real-time systems.

4.3.2 Active & Passive Threads

As ordinary kernel objects, Scheduling Contexts are decoupled from TCBs (and thread priority) – which allows for some interesting mechanics. For ordinary, *active* threads, an SC can be *bound* to a TCB, which allows the TCB to use the CPU resources represented by the scheduling context. Using scheduling contexts, it is also possible to construct so-called *passive* threads. A passive thread does not possess a scheduling context of its own (except when required during initialisation, after which it is removed). Instead, passive threads represent servers which await *donation* of a scheduling context from an active thread. An example of Scheduling Context donation is illustrated in Figure 4.2, and described below.

[1] seL4 also supports a domain scheduler underneath the FP scheduler, but we will not use it in our work, so omit it here

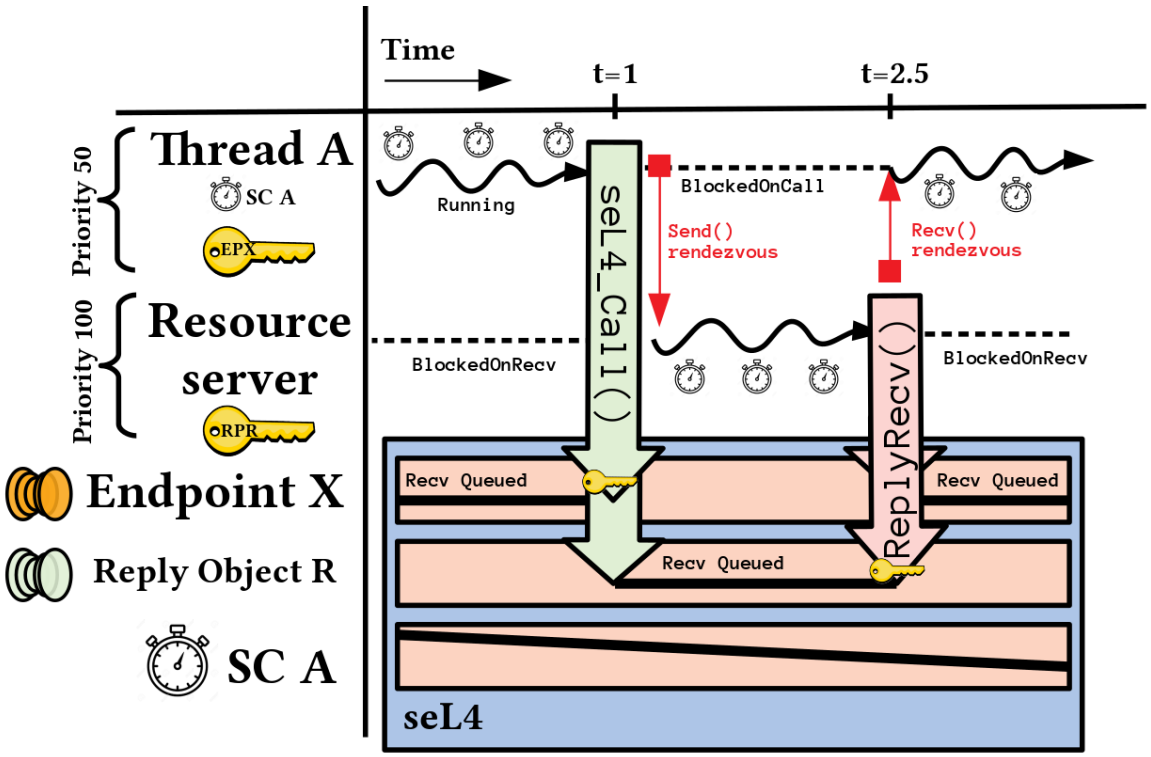

Figure 4.2: Scheduling Context Donation

Initially SC A is bound to Thread A, which also has a capability to Endpoint X. The resource server has a capability to Reply Object R, which is later used as a reply channel.

At t=0: Thread A is running on SC A, and the resource server is waiting for requests on Endpoint X. The scheduling priority of Thread A is 50 – from Thread A's TCB.

At t=1: Thread A issues a request (`Call()`) to the resource server, using its capability to Endpoint X. The resource server is already blocked on Endpoint X, so rendezvous occurs and the message transferred. Thread A simultaneously becomes blocked on Reply Object R (which is assumed previously passed to `Recv()` by the resource server before t=0). Since the resource server has no scheduling context, Thread A's context, SC A is *donated* to the resource server. The resource server continues executing on SC A, at scheduling priority 100 (from the resource server's TCB).

At t=2.5: The resource server replies to Thread A through Reply Object R. SC A is returned to Thread A, and the resource server becomes blocked on Endpoint X once again.

4.3.3 Influencers on scheduling behaviour

In addition to the above properties of Scheduling Contexts, the FP in-kernel scheduler, and IPC, extra system calls exist to facilitate user-level scheduling. Below we summarise key operations which influence how threads are scheduled by the kernel (we have already encountered the first 3):

- **IPC:** IPC system calls can block & wake threads in accordance with the IPC semantics described earlier.
- **Thread Priorities:** Priorities directly influence the in-kernel FP scheduling behaviour. They can be changed at runtime using TCB capability invocations.
- **Scheduling Contexts:** A scheduling context must be bound to a thread for it to be runnable. Additionally, the bound SC must have remaining budget (See subsection 4.3.1). A passive thread may only be scheduled if it is 'donated' an SC over IPC.
- **`Yield()`:** A thread can use a `Yield()` system call to sacrifice it's timeslice (or remaining SC budget) - this can be used to force the next runnable thread at the same priority level to be scheduled.
- **`YieldTo()`:** The `YieldTo()` invocation takes a scheduling context as an argument, and moves the thread bound to that SC to the front of the ready queue (at that priority). This can be employed by a user-level scheduling thread to manipulate the kernel scheduling queues.
- **Timeout Fault:** In the event a thread depletes its SC's budget whilst executing, the thread will be preempted, and its handler (if specified) notified that the thread exceeded its budget. If no handler is specified, the thread will be silently treated non-runnable until the SC is replenished
- **Interrupts/Notifications:** These can additionally change the run state of threads.

4.3.4 seL4 User-level Scheduling Example

By combining the mechanisms we have described thus far, it is possible to construct a simple example of what a user-level-scheduled system on seL4 might look like.

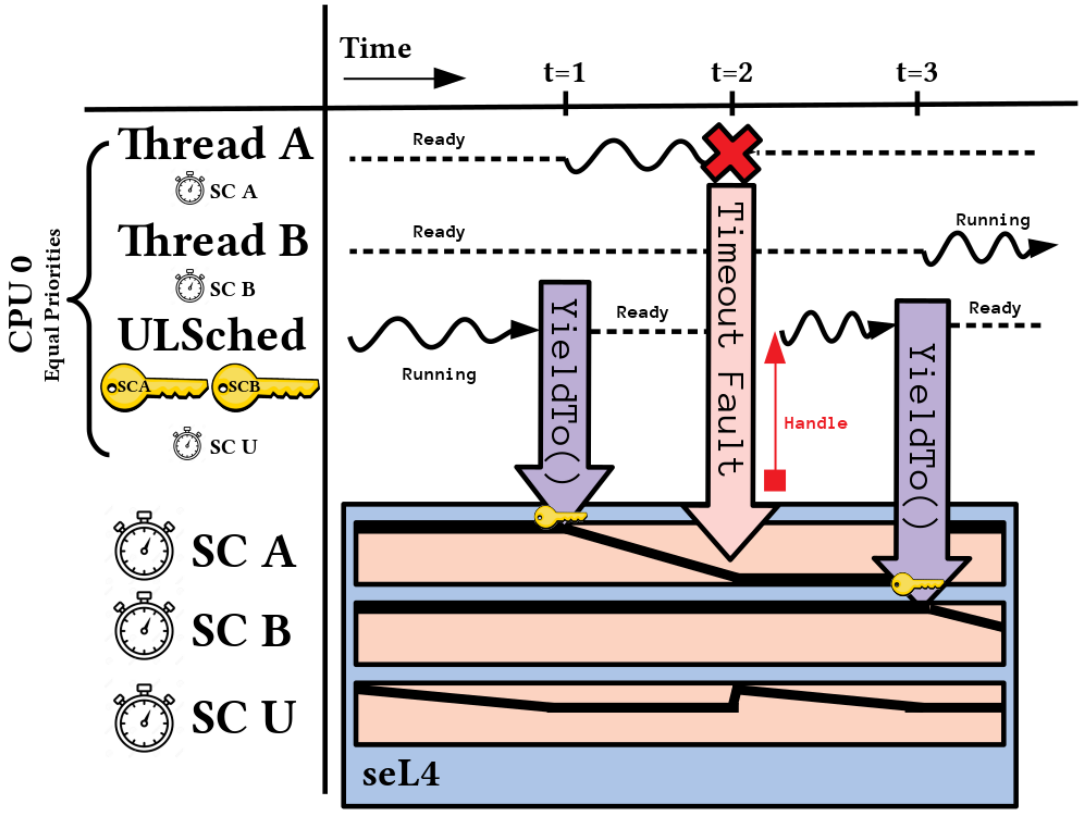

Figure 4.3: Example of user-level scheduling

Observing Figure 4.3, we initially assume that a system consisting of 2 threads and a user-level scheduling thread already exist. The scheduling contexts (SC A, SC B, SC U) are assumed to be already bound to each thread. Additionally, note that the user-level scheduling thread possesses capabilities to the scheduling contexts SC A and SC B so that it can `YieldTo()` them when required.

At t=0: ULSched is running, and computes the first thread that should be selected for execution. ULSched slowly depletes its scheduling context, SC U.

At t=1: ULSched has decided to schedule Thread A. To enact on its decision, ULSched use `YieldTo()` with a capability to Thread A's Scheduling Context. Since Thread A is runnable, it is moved to the front of the scheduling queues and the kernel switches to Thread A.

At t=2: Thread A depletes its entire budget, a timer interrupt fires – invoking the kernel. The kernel preempts Thread A, and acknowledges that it has depleted its budget with a Timeout Fault. ULSched was configured at system initialisation as the fault handler for Thread A, so control returns to ULSched. Note also at this time that SC U happens to receive a budget replenishment.

At t=3: ULSched decides it is Thread B's turn. This time, ULSched performs a `YieldTo()` with the SC B, which causes a switch to Thread B.

In reality, user-level scheduled systems are more complex. For example, in the case of EDF, the scheduler needs to be notified on timeout faults and timer interrupts. A way of implementing this is to move the user-level scheduler thread priority above other threads, and have it wait on timeout faults or interrupts using a single system call.

4.4 Summary

This chapter has outlined some key primitives that enable construction of mixed-criticality, user-level-scheduled systems on seL4 (candidate designs of which will be outlined in the next chapter). The reader is encouraged to explore Lyons et al. [2018] for a deeper exploration of these primitives.

5 | Approach

This chapter outlines our approach for constructing and benchmarking user-level-scheduled mixed-criticality systems on seL4. We shall revisit our research goals and propose an approach for meeting them.

5.1 High-Level Approach

To meet our goal of exploring the feasibility of constructing user-level-scheduled mixed-criticality systems, we must address the following high-level tasks:

1. **Prototype a user-level-scheduled MCS:** We require a realistic mixed-criticality architecture which is portable to (or ideally already available in) other kernels. This will be addressed in section 5.2.

2. **Create tracing & benchmarking infrastructure:** To evaluate the effectiveness of our approach (performance evaluation), and to develop an effective scheduler (validation/correctness), it is necessary to construct tracing & benchmarking tools. We shall address this in section 5.3.

3. **Benchmark against other approaches:** The question of feasibility should be answered by comparison with other approaches. Our MCS prototype and tracing infrastructure will aid this. This will be addressed in section 5.4.

5.2 Approach: User-level MCS on seL4

Constructing a user-level-scheduled MCS on seL4 requires a scheduling architecture, a resource sharing policy, and test task sets. Our evaluation prototype is built on a clustered-EDF (C-EDF)

scheduler and an MC-IPC resource sharing policy – the choices of which will be justified shortly.

C-EDF is arguably the most popular real-time multicore scheduling architecture in current literature (it is the most mature scheduler in both RASP [Mollison and Anderson, 2013] & LITMUS [Calandrino et al., 2006]). Additionally, P-EDF (recall P-EDF is just C-EDF with cluster size 1) has been demonstrated the most desirable approach for multicore hard real-time systems [Brandenburg, 2011], and G-EDF (which is C-EDF with cluster size equal to the core count), is the most desirable approach for increasing CPU utilisation. Because of the real-time optimality and maturity of C-EDF (desirable for benchmarking against existing comparable systems), our user-level MCS is based on a preemptive C-EDF scheduler (we will explore design issues further in subsection 5.2.1).

MC-IPC is unique in its comprehensive analysis – and handles malfunctioning high-criticality tasks, as well as non-reentrant servers, in a time-bounded way. As a protocol intended for use with resource servers on a microkernel-like system, MC-IPC should map well to the mixed-criticality primitives available in seL4. Another motivator for our desire to implement MC-IPC is its complexity – we desire to find shortcomings in the kernel API for investigation. Design issues associated with MC-IPC are explored in subsection 5.2.2.

5.2.1 Scheduler: C-EDF

There are 2 main issues to address in constructing a C-EDF scheduler on seL4 – the design of the scheduler itself, and providing mechanisms to bootstrap the system into the correct cluster-to-cache-heirarchy.

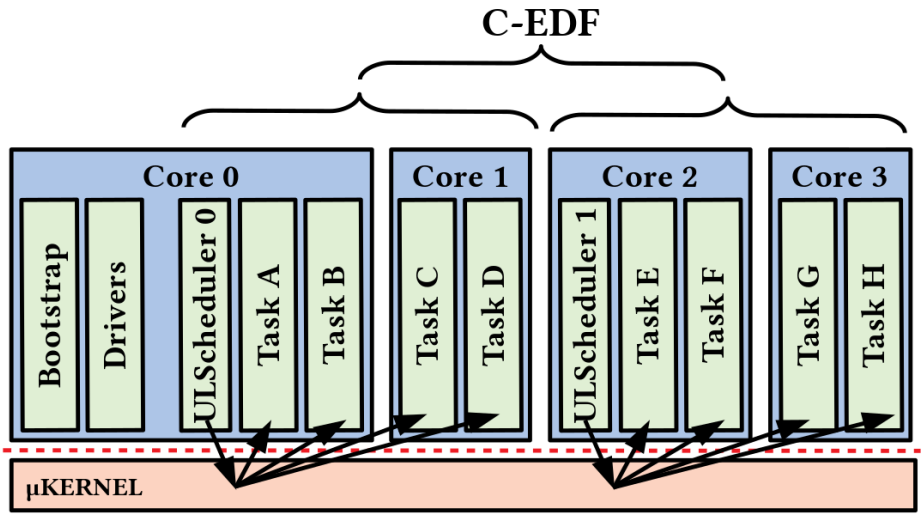

Figure 5.1: Abstract C-EDF architecture on seL4

Figure 5.1 provides an overview of our C-EDF architecture on seL4. The system consists of a simple bootstrap thread, which sets up a system of user-level schedulers assigned tasks according to a task assignment algorithm.

In this example, a single G-EDF scheduler thread is assigned to the root of each cluster. These G-EDF threads together form a C-EDF scheduler. Now, we shall justify why we only place a scheduler thread on the first core in each cluster, rather than distribute them across all cores. For some context, in Linux, core-local datastructures (and logic) are necessary for efficient migrations – the kernel will IPI to itself on another core, which then has responsibility to complete a migration. With this in mind, we can now answer the question – why place our scheduling threads only on the root of each cluster? Our first reason is that under seL4, for migrations, its not necessary to 'distribute' the scheduler thread – migrations can be performed using a single seL4 system call (`seL4_SchedControl_Configure`). Even though the end result is the same (seL4 will IPI to itself on another core to complete a migration) this system call abstracts much of the need for core-local scheduling logic under G-EDF. Our second reason is that a distributed set of schedulers needs to maintain a consistent shared view of the current scheduling state - which effectively means serializing decisions anyway. On the other hand, a potential advantage of having a distributed G-EDF scheduler under our model is if core-local interrupts, and core-local context switches dominate – but this is not easily predictable under G-EDF.

Below, we list some more design trade-offs associated with C-EDF schedulers.

User-level C-EDF scheduler design issues

- **Cache Grouping:** Discounting the degenerate P-EDF case, the 'spread' of each cluster should generally be over sets of cores which share fast caches. For example, if 2 cores share an L2 cache, then the cost of operating on shared data between the cores will be lower than if 2 cores share an L3 cache.

- **Queue structures:** Each EDF node requires priority queues for deadline & release tracking of tasks. There are many candidates for such structures.

- **Timer hardware:** Each EDF node requires access to a timer to preempt the currently scheduled thread if a new job is to be released or a current deadline expires. On some hardware platforms (such as the Freescale i.MX6), there are fewer shared peripheral timers available to userspace than potential EDF nodes. One solution is to employ a separate timer server to multiplex the hardware timer, but there are other solutions – which we cover in chapter 6.

- **Cache Affinity Heuristics:** If a job from a task was previously scheduled on a particular core, it is desirable to assign new jobs from the same task to that core. A simple case is to prefer to schedule such a job on a particular core in the event of an arbitrary tiebreak. This might be rare, and more complex heuristics could be difficult to implement without breaking per-cluster EDF correctness.

- **Integration of criticality modes:** The representative WCET of tasks may change in the event of a mode switch, so it is desirable for the C-EDF scheduler to allow for efficient updating of its scheduling data structures in the event of a mode switch.

- **Interoperation with other schedulers:** One of the key advantages of user-level scheduling is that we may use it 'above' a verified in-kernel scheduler. The C-EDF scheduler should maintain correctness even if higher-priority tasks are running on the system, under a different scheduler.

- **Cooperative yielding:** The scheduler should gracefully handle tasks which do not consume their whole budget.

- **seL4 capability management:** The way in which capabilities managed in the scheduler should be carefully designed. For example, in a P-EDF system, it is undesirable for user-level schedulers to have capabilities to scheduling contexts from other partitions. This will likely be the responsibility of an initial bootstrap thread.

- **Task Assignment:** A system-level problem, the assignment of task sets to clusters is proven to be an NP-hard bin-packing problem [Brandenburg, 2011]. In our test systems, we use static task assignments to avoid tackling this.

Note that cache considerations and more advanced optimisations were gradually introduced as the system was benchmarked for weak points against itself and other systems. We further explore how our implementation addresses the above issues in chapter 6.

5.2.2 Resource Sharing: MC-IPC

There are two key difficulties in implementing MC-IPC on top of seL4 primitives – replicating the reservation model (which is different to that of seL4), and implementing its 3-level queue structure in a way which preserves MC-IPC's mathematically-backed invocation bound. We explore both issues in more detail below.

MC-IPC is based on a different model of reservations to seL4-MCS primitives (budget is consumed even when a task is not scheduled and waiting for a request to be served under MC-IPC – this is called *idling reservations*), so it is necessary to provide user-level logic implementing this type of reservation under seL4. A particular issue is that a different reservation model implies that we need arbitrary timer interrupts in the resource server to prune the request queues (in accordance with the MC-IPC specification).

Next we shall consider a more complex issue – resource server invocations. To implement some protocol complexity on top of ordinary IPC, the first candidate design one might consider is a single guard server per 'ordinary' resource server, with capabilities only handed out to guard server endpoints (not the actual resource). All resource requests would then be proxied through this MC-IPC guard server. Unfortunately, this would not work as cross-core interference from a high-criticality denial-of-service failure can still occur if clients contend on the same priority-ordered endpoint (see MC-IPC section in chapter 3 for an example).

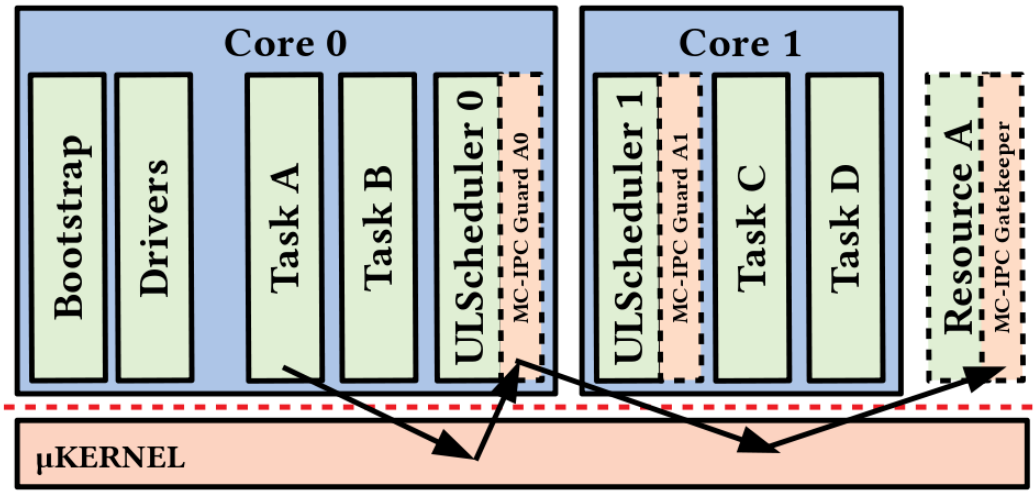

Figure 5.2: Simplified MC-IPC design

A more promising solution, and the approach we implement, is illustrated in Figure 5.2. The (passive) shared resource thread contains an MC-IPC 'gatekeeper' – a library in the same address space as the resource server which proxies requests. For a task in a cluster to IPC to a resource, it first has to travel through its *cluster local* userlevel scheduler, which 'bounces' the request (if it reaches the head of a cluster-local request queue) to the *actual* resource gatekeeper library. The main benefit of travelling through our cluster-local userlevel scheduler is that it reduces the amount of mode-switches that need to be performed – since MC-IPC invocations require bookkeeping in the userlevel scheduler and potential timer reprogramming. A more detailed examination of our MC-IPC implementation and how it forms a correspondence with the protocol as described by Brandenburg [2014] is provided in chapter 6.

5.3 Approach: Tracing & Instrumentation

In order to investigate the performance & correctness of a scheduler implementation, we employ the kernel's tracing features.

For scheduler *correctness* validation, the kernel log buffer infrastructure is employed. The kernel log buffer is a memory area in the kernel that is written to with invocation information whenever the kernel is entered (if logging is enabled). As there is not yet comprehensive log buffer multicore support, this was implemented as part of this thesis. Additionally, since a useful way of parsing the results of a kernel trace after an experiment is not available, a visual

scheduler validation aid has been written as part of this thesis (see `schedplot` in chapter 6).

Any kernel instrumentation is not without overhead. As a result, we detail how our measurement strategies account for measurement overheads (when using the log buffer and trace points) in chapter 7.

5.4 Evaluation Strategy

We seek to demonstrate the following:

1. Tracing infrastructure that is easy to use and accurate.
2. Performance competitiveness with similar-class monolithic systems (i.e LITMUS, RASP).
3. Kernel improvements to multicore user-level scheduling performance.

The accuracy of our tracing infrastructure is evaluated by quantifying the overhead introduced into kernel system calls, and comparing our tracing accuracy with conventional trace points already used for other seL4 benchmarking work over different timespans and workloads. Additionally, we conduct a case study – using our tool for a real-world scheduling trace debugging problem unrelated to this thesis (quadcopter firmware).

To evaluate our scheduler, we measure key properties (such as average scheduler entry time) in G-EDF & P-EDF configurations, in different scenarios, with different time sources, and with different task distributions. For generating simple random multicore real-time task sets, we use the `randfixedsum` algorithm [Emberson et al., 2010], similar to previous work in this space. To compare our scheduler with monolithic systems – we compare 5 key scheduler overhead metrics (i.e release latency, context switch overhead – all are described in detail in chapter 7). These overhead measurements are fed into a schedulability simulator (`SchedCAT` [Brandenburg, 2011]), which allows us to establish which scheduler is advantageous in what situations.

Benchmarking the impact of resource sharing protocols is more difficult – in the case of MC-IPC, an ideal strategy would be to replicate the resource sharing case study used to demonstrate invocation bounds in the MC-IPC paper (it is open source), and compare it with an existing LITMUS-based monolithic implementation. Unfortunately, we were not able to complete a re-implementation of the case study due to time constraints, and the authors of the only existing comparison (monolithic) implementation requested that we not benchmark against it,

so evaluation possibilities with MC-IPC are somewhat limited. Recall that besides performance investigation, one of our key thesis motivations was to demonstrate that building schedulers at user-level requires comparatively low implementation effort. It is still possible for us to evaluate whether the implementation functions, and make a comparison in implementation complexity based on patches made to the similar-class monolithic system. As a result, these aspects will be the focus of our evaluation in chapter 7, and we cover some proposed changes to the seL4 model which would likely make a future evaluation more competitive in chapter 8.

5.5 Summary

This chapter illustrated our approach to implementing a flexible multicore scheduler (C-EDF) & resource management protocol (MC-IPC) as user-level components on the seL4 microkernel. We presented problems that our design will need to solve, a high-level overview of our design, and how we evaluate our implementation. In the next chapter, we delve deeper into the implementation of the high-level approach covered here.

6 | Implementation

This chapter explores our implementation, where we shall dive deeper into the details of each component which was described broadly in the previous chapter. Most of this work is not platform-specific (except for some kernel modifications), however our testing and evaluation was limited to the x86 32-bit architecture and 32-bit ARM architectures. An illustration of the broad areas we shall describe in this chapter is provided in Figure 6.1.

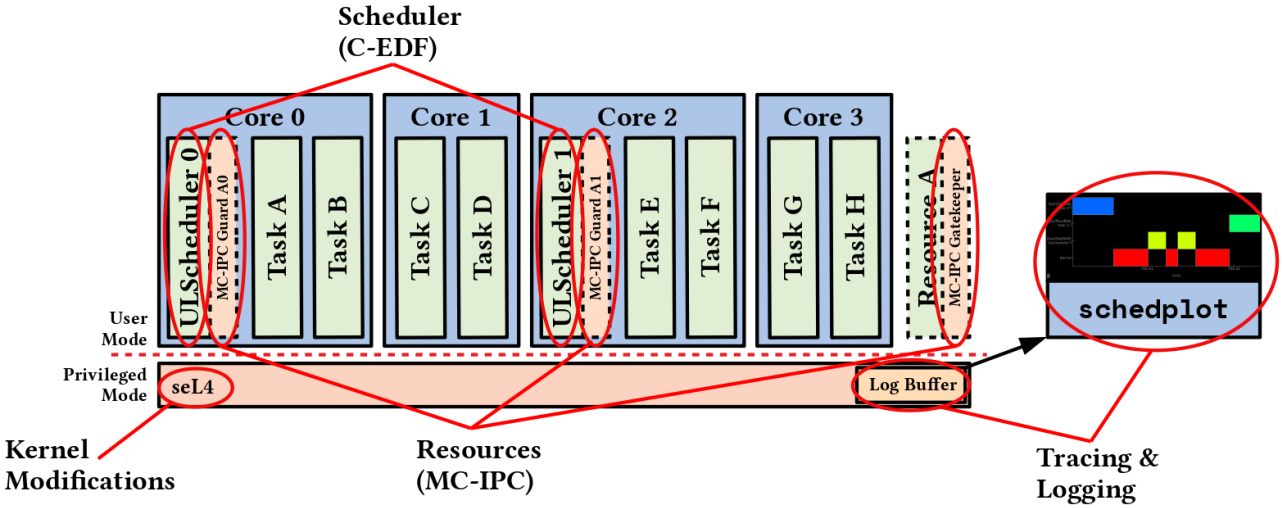

Figure 6.1: Different components of our implementation (circled red)

The following sections will detail each component – namely tracing & logging, our C-EDF scheduler implementation, our (MC-IPC) resource sharing implementation, and some kernel modifications.

6.1 Tracing & Benchmarking Infrastructure

As this thesis required evaluating the correctness & performance of our scheduler implementation, improving tracing and benchmarking infrastructure associated with seL4 was essential.

This involved:

- Optimising & adding features to the seL4 kernel log buffer infrastructure, including multicore support.
- Writing a visualisation tool – `schedplot` (in Python) for displaying & analysing logbuffer results.
- Creating test code to evaluate that the above work functions correctly, as well as measure the overheads introduced so that it can be accounted for in later analysis.

6.1.1 Kernel Log Buffer Changes

In order to make the kernel log buffer useable for our application, we added some extra switches to the kernel configuration which alter the contents of log buffer entries. Additionally, since the current implementation maps a single buffer into the kernel's address space which is then used by kernel entries on all cores, we investigated adding multicore support through core-local mappings.

Changes to Buffer Contents

With none of these new features enabled, the 'original' contents of a single log buffer entry is pictured in blue in Figure 6.2. At first glance, it is easy to see why changes were necessary. There is no way to determine which CPU the entry corresponds to, or whether it describes a switch between tasks – both critical elements required to trace a user-level scheduler. Consequently, we implemented two additional log buffer trace modes shown in Figure 6.2 and Figure 6.3.

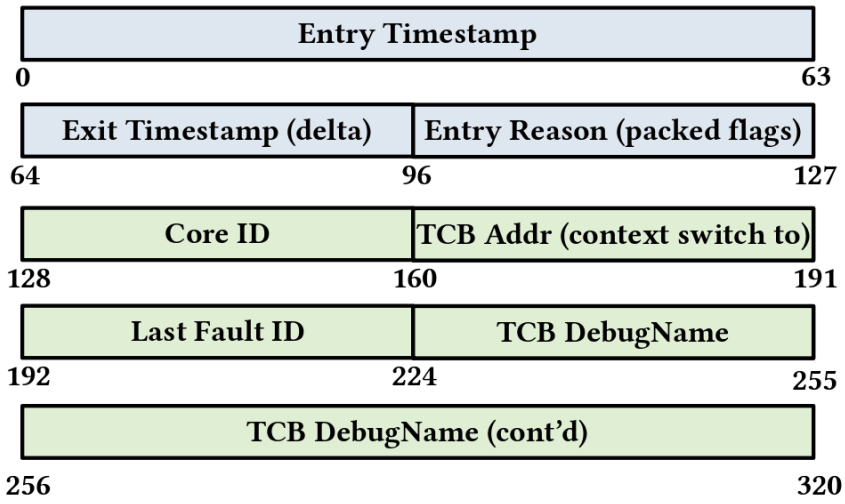

Figure 6.2: Contents of a log buffer entry (original in blue, debug extensions in green)

In debug trace mode (Figure 6.2), we added a number of properties. Critically, the core ID (determined by reading the core-local kernel stack address) and TCB address (which indicates the target of a context switch by the kernel) properties allow offline analysis to determine what the execution trace of userspace is. The additional properties facilitate ease of debugging – for example, fault ID allows a user to determine whether a timeout exception was cause for a thread to be descheduled. Of course, in this mode, each log buffer entry is quite large (40 bytes), and creating such entries has significant overhead (which we shall quantify later). To work around this problem, we implemented another mode which we deem 'lite' tracing mode, whereby each log buffer entry contains the bare minimum amount of information required to reconstruct a user-level scheduling trace (see Figure 6.3).

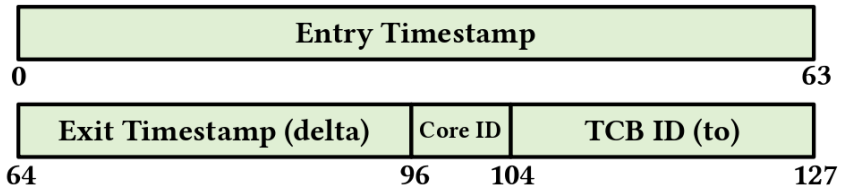

Figure 6.3: Contents of a log buffer entry (lite/trace mode)

In this mode, each log buffer entry is 16 bytes long. We experimented with reducing the size of the entry timestamp fields further, however this had negligible performance impact.

Core-Local Mappings

The existing logging infrastructure operates as follows. A single 4-megabyte page is allocated and mapped into userspace by a benchmarking application. The capability to this page is then provided to the kernel using an `seL4_BenchmarkSetLogBuffer` invocation, after which the kernel maps it write-through (to minimize the cache impact) and then uses this page to log kernel entries. The existing kernel behaviour on multicore is relatively simple – a monotonically increasing, atomically incremented up-counter prevents different cores from writing to the same part of the log buffer[1].

During an early evaluation phase, we discovered that transitioning the kernel log buffer from mapping log buffers globally to instead use a per-core mapping counter-intuitively *increased* the log-buffer overhead in our tests. Since these changes were therefore not used in our final evaluation, they are described in Appendix A.

6.1.2 `schedplot` visualisation application

A significant part of this thesis was developing `schedplot`, which is a visual tool to test the correctness of user-level schedulers by analysing log buffer results.

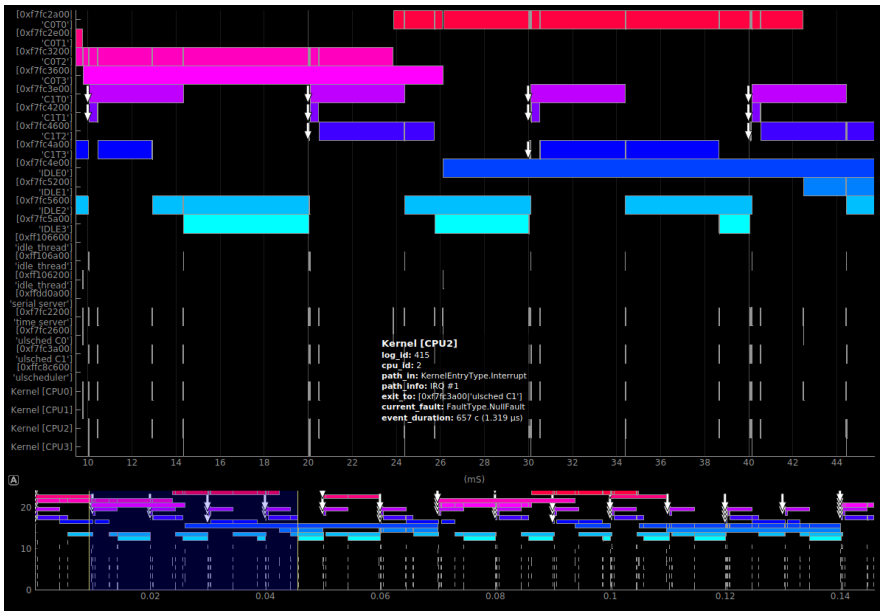

Figure 6.4: `schedplot` GUI screenshot

[1]The global kernel lock (also known as the Big Kernel Lock – see Peters et al. [2015]) prevents similar race conditions, but it is not locked on every kernel entry

A screenshot of the tool interface is in Figure 6.4, and has the following main features:

- A visual interface for navigating scheduler activity (with zoom capability from second-timescales down to nanosecond-timescales).
- Visualisation of sporadic task parameters, which can be displayed on top of a scheduling log to verify real-time correctness.
- Generation of log statistics – such as average kernel entry times, context switch frequencies, core idle times and thread utilisations.
- Detailed tool-tips (when mouse is hovered over events) which describe kernel entry reasons, thread system call types, event durations and fault occurrences.
- A 'minimap' element that allows the user to view current log position when zoomed in to small timescales.
- Command-line options that allow a user to filter out specific types of logs – for example for specific cores or specific threads.
- A feature which allows for estimation of real-world event times, by applying measured log buffer overheads and modeswitch overheads to displayed statistics.

The tool is cross-platform, based on pyqtgraph and Qt for real-time graphical operations. Note that there are other open-source tools called `schedplot` (for completely different purposes) – it may be necessary for us to rename this project at some point, but we have decided to keep the name as-is due to current lack of inspiration.

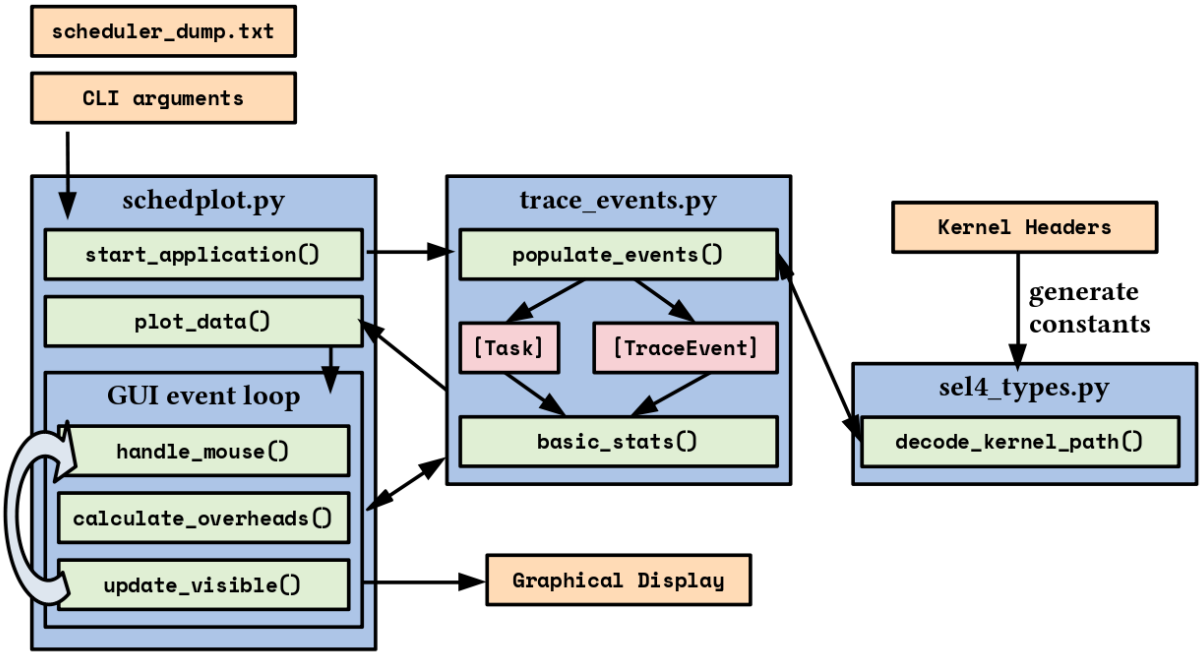

Figure 6.5: `schedplot` high-level architecture

A high-level functional overview of `schedplot` is provided in Figure 6.5. The application can be broadly divided into 3 components, each of which corresponds directly to an implementation file. Broadly, the `schedplot.py` component manages all aspects of the GUI, and the `trace_events.py` component together with the `sel4_types.py` component translate & sanitise a provided scheduler dump into in-memory objects which can be displayed. We shall now work through how this application operates in more detail, both externally and internally.

The first step to working with schedplot is providing the scheduling trace to the application. Generally (as is the case with all the scheduler measurements made in this thesis), the log buffer is dumped in ASCII format after an experiment is completed (over a serial port), by the host seL4 application. The `schedplot` application is then executed by providing this scheduler dump, as well as some command line arguments specifying display options, filtering operations or overhead measurements. Some command-line options provided are illustrated in Listing 6.1.

Before the application is started (as part of our build process), the seL4 kernel headers are used to generate constants which are included by `sel4_types.py`. These constants describe architecture-specific (and kernel build flag-specific) values which are used by the kernel internally to represent capability types, fault identifiers, and system call identifiers. Once the application is started, `schedplot.py` will load the scheduler dump file and call into `trace_events.py` which will attempt to translate the dump into 2 lists of in-memory objects –

```
--isolate_core N         Only display readings from provided core
--ignore_threads [S]     Don't create thread events with provided TCB names
--keep_threads [S]       Only create thread events with these TCB names
--label_putchar          Display DebugPutChar calls inline with scheduling trace
--show_deadlines         Display sporadic task model deadlines on top of traces
--modeswitch_overhead N  Measured modeswitch overhead (in + out) in cycles
--logbuf_overhead N      Measured logbuf overhead in cycles
```

Listing 6.1: Some `schedplot` CLI options.

representing sporadic task parameters (`Task`) and '`TraceEvents`'. `TraceEvents` are essentially cleaned-up, more informative versions of the scheduling trace. These are created by querying `sel4_types.py` with raw values to obtain objects describing what a particular entry in the log buffer actually means. The results of this operation are used, for example, to determine whether execution of a task (or the idle thread) happens between 2 logbuffer entries on the same core. There are many more `TraceEvent` objects created than entries in the scheduling dump, since a single `TraceEvent` can represent a duration of task execution or a kernel invocation, but not both (recall that the scheduling dump only records kernel invocations).

After their creation, `TraceEvents` form the core datastructure in the application. They are used by `schedplot.py` to plot the visual display, query mouse events, check sporadic task parameters, and perform overhead accounting (see subsection 7.2.1). This datastructure is also used by `trace_events.py` to compute some basic statistics (such as cumulative entry times or utilisation), in addition to some sanity checks (for example, if the scheduling trace represents a CPU usage less than $100\% \times N_{CPUs}$ and idle thread entries are recorded; then there is obviously a problem with the scheduling dump).

As for how `schedplot` supports overhead accounting, we will cover this in subsection 7.2.1.

6.2 C-EDF Scheduler Implementation

6.2.1 Overview

In this section we will discuss how our C-EDF scheduler is implemented. Each cluster in a C-EDF arrangement has its tasks scheduled by a cluster-local user-level scheduler, which is an active thread placed on the first core in the cluster. In our implementation, the cluster arrangement can be modified at compile-time to match the target cache hierarchy, or specified as blanket P-EDF or G-EDF.

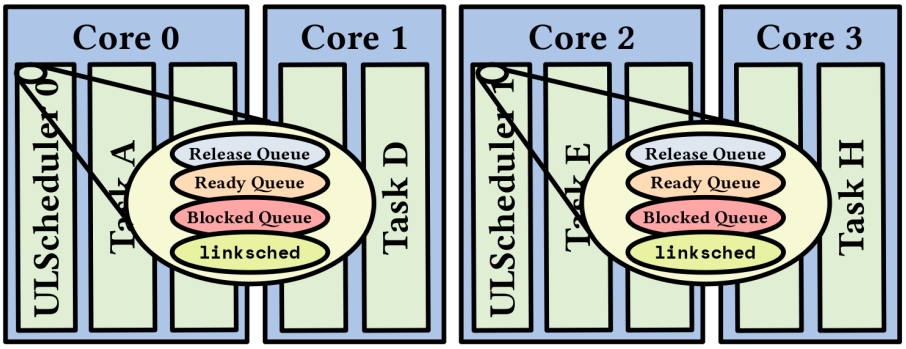

Figure 6.6: Every G-EDF cluster has its own set of queues and CPU links

Each cluster can be thought of as an independently scheduled G-EDF node (ignoring cross-cluster resource sharing, which we cover in more detail in section 6.3). As a result, every cluster contains its own set of scheduling queues and job-to-CPU associations – as illustrated in Figure 6.6. Throughout the lifetime of a system, tasks will move between a number of states within their cluster.

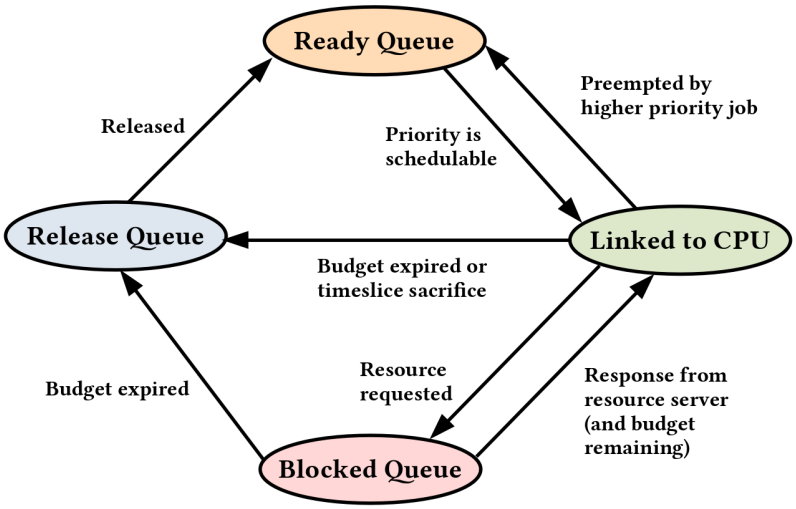

Figure 6.7: States which tasks may move between

These states are illustrated in Figure 6.7. A task enters the system by being inserted into the user-level scheduler's release queue. Each task has a priority attribute, which corresponds to one of 2 properties. When a task is in the release queue, its priority corresponds to the time at which it should be released (lowest priority is assigned to the next task to release). Once this time elapses, a job is created and this task moves into the ready queue. When a task

enters the ready queue, its priority is updated to correspond to its relative deadline (such that earliest deadline job has the lowest priority). The release, ready, and blocked queues are all implemented as minimum-priority queues (see subsection 6.2.2). As a result, the lowest priority N_{CPUs} elements of the ready queue are the tasks which are schedulable under G-EDF. To map schedulable jobs to CPUs efficiently, we employ 'link-based scheduling' which is described in detail in subsection 6.2.6. Once the budget of a scheduled job expires, it is moved to the release queue with a release time set based on the task period. Essentially, periodic tasks with no resource sharing will simply move between these 3 states (release, ready, linked) until the system is shut down. The semantics of tasks with respect to the blocked queue will be explored in more detail in section 6.3.

6.2.2 Priority Queues

In our scheduler, there are two selectable priority queue implementations – binomial heaps, and red-black trees. Both are based on open-source implementations of the datastructures – we use the implementation of red-black trees from Vittek et al. [2006], and binomial heaps from Brandenburg [2012]. Heaps are the 'classical' answer to priority queues – the binomial heap implementation we employ has constant-time insertion and extract-min operations (the minimum is cached, and find-min occurs more often than any other operation). Additionally, binomial heaps are the datastructure used by LITMUS^{RT} for its ready/release queues. We also chose to include red-black trees because our implementation sometimes queries queues for presence of a job ($O(N)$ for heaps, $O(\log N)$ for a red-black tree), in order to verify some runtime invariants. In testing, we found binomial heaps to be faster in general – but this will be explored further in chapter 7.

6.2.3 Bootstrap & Capability Distribution

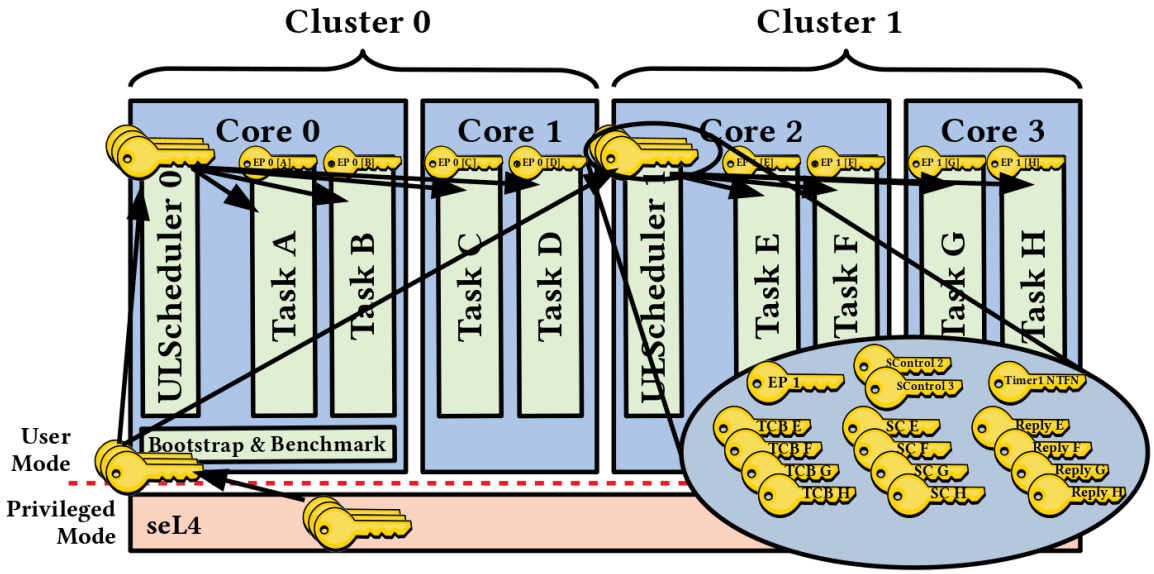

Figure 6.8: How capabilities are distributed under C-EDF

When a test system is run, a bootstrap thread creates a set of tasks according to pre-generated test task set, creates a user-level scheduler for each cluster, and then adds each task to the pre-assigned cluster-local schedulers. The distribution of capabilities under this arrangement is illustrated in Figure 6.8. Each scheduler is provided with `SchedControl` capabilities so that it may migrate scheduling contexts for tasks under its control between cores in the cluster. Additionally, each scheduler is provided with a timer notification which is bound to the scheduler's TCB, so that a scheduler may await events from registered timeouts or requests from scheduled tasks. Each task requires access to an endpoint to the cluster's user-level scheduler (so that it can sacrifice its timeslice or request resources). When a task is added to a user-level scheduler, the scheduler takes ownership over the task's TCB and scheduling context, creates a reply object for each task (reply object management is covered further in subsection 6.2.7), and supplies a badged capability to its communication endpoint. In this fashion, due to the way capabilities are distributed it is impossible for user-level schedulers to interact with tasks or cores which they do not have jurisdiction over (ignoring e.g. resource sharing, or timer device sharing).

6.2.4 Tour of the Scheduling Loop

```
1  while(true) {
2    /* Release jobs, get next release time */
3    uint64_t next_release_delta = release_jobs(data, time);
4
5    /* Compute next budget timeout by walking all scheduled jobs */
6    uint64_t next_budget_delta = next_budget_timeout_relative(data);
7
8    /* Figure out next desired interrupt */
9    uint64_t next_timeout_delta =
10     MIN(next_release_delta, next_budget_delta);
11
12   /* We need to be woken up at this relative timeout or sooner */
13   set_timeout_for(data, next_timeout_delta);
14
15   /* Wait for an interrupt (or timeslice sacrifice) from tasks */
16   info = seL4_Recv(endpoint.cptr, &badge, reply);
17
18   /* Update time & deltas */
19   uint64_t time_delta = get_time() - time;
20   uint64_t time += time_delta;
21
22   /* Why did we wake up? IRQ or coop yield? */
23   if(was_irq(badge)) {
24     sel4platsupport_handle_timer_irq(data->env_timer, badge);
25   } else if(was_timeslice_sacrifice(badge)) {
26     deschedule_job_with_badge(badge);
27   }
28
29   for(cpu = 0; cpu != N_CPUS; ++cpu) {
30     job_t *job = job_scheduled_on_cpu(cpu);
31     job->budget_consumed += time_delta;
32     conditional_complete_job(job, time);
33   }
34 }
```

Listing 6.2: (simplified) main G-EDF scheduling loop.

A simplified overview of the main scheduling loop of a single G-EDF cluster (resource sharing & idling logic omitted) is shown in Listing 6.2. On line 3, `release_jobs()` checks the release queue for any jobs to be released, moves them to the ready queue and returns the next job release time. `release_jobs()` additionally notifies the link scheduler (subsection 6.2.6) for each released job, which ensures that any newly schedulable jobs will displace those currently being scheduled, if their deadline is near. Note that the reason we have to set timeouts for budget expiry (line 6) is because we implement our own reservation model for MC-IPC – this additional step is not necessary if scheduling contexts are used for handling budget expiry.

A scheduler on a particular cluster will need to make a new scheduling decision at the sooner event of:

1. The next job needs to be released
2. A currently scheduled job's budget expires
3. A currently scheduled job sacrifices its timeslice

These first two items require use of a timer device – so we will now explore how `set_timeout_for()` (line 13 of Listing 6.2) operates.

6.2.5 Sources of Time

Our C-EDF implementation supports different time sources as a compile-time switch. Namely, the supported ways in which `set_timout_for()` can be implemented are:

1. Through an *(unshared) userspace timer driver*. This is the simplest option. Note that some platforms do not have more than one timer device available to userspace, so this method is only allowed in the single-cluster G-EDF case.
2. Through a *(shared) userspace timer driver*. In this mode, an active thread on the first cluster serves as a time-server, and schedulers in each cluster are given endpoints and notifications to communicate with the time-server. This requires many mode switches on timer reprogramming, and impacts schedulability on the core which runs the timer server (although it is possible to put the timer server on its own isolated core).
3. Through *timeout exceptions*. In this mode, the scheduling context of the job with the earliest deadline is programmed such that the context's budget expires when the scheduler requires an interrupt. An advantage is that this approach exploits the core-local timers used by the kernel, and needs less mode switches. The main limitation is that there is no way to have such a scheduler exist alongside other schedulers, as it assumes that the scheduler has complete control over the cluster (special idle threads are required to use up all time not spent in tasks).
4. Through a *new system call*, `YieldToTimeout`. This mode is similar to timeout exceptions, but without the disadvantages outlined above. The time provided to `YieldToTimeout` is actually an absolute time (rather than a budget), so this approach can be used alongside other schedulers. `YieldToTimeout` is described in more detail below.

Each of these have advantages and limitations as described above. We will explore the performance implications of each in chapter 7.

The `YieldToTimeout` System Call

We implemented this system call as a basic extension of seL4's existing `YieldTo` system call, which moves the provided scheduling context to the front of the kernel scheduling queues. `YieldToTimeout` additionally allows user-level to specify an absolute timeout, and alters the existing timeout exception semantics. If this absolute timeout elapses, a timeout exception is generated, originating from the yielded-to context. When a user-level scheduler `Recv`s this timeout exception, the exception data indicates whether it came from a budget expiry or an absolute timeout, as well as the current kernel time.

This essentially allows a user-level scheduler to schedule a thread, set a timeout, wake up, and get the new time in a single `YieldToTimeout` & `Recv` combination. These semantics are similar to those used by COMPOSITE [Gadepalli et al., 2017] for user-level scheduling.

6.2.6 Link-Based Scheduler

One of the key design issues with constructing G-EDF schedulers is determining how to map jobs to CPUs. Essentially, we want a policy which places the same job on the same CPU as long as it remains schedulable (to avoid unnecessary preemptions). A solution which is employed in modern G-EDF implementations is known as a *link-based scheduler* [Brandenburg, 2011]. The key property of a link scheduler is that jobs are *linked* to CPUs, but a link does not necessarily correspond to that job actually being scheduled. That is, a job which is ready (& schedulable) can be in a *linked* or *unlinked* state.

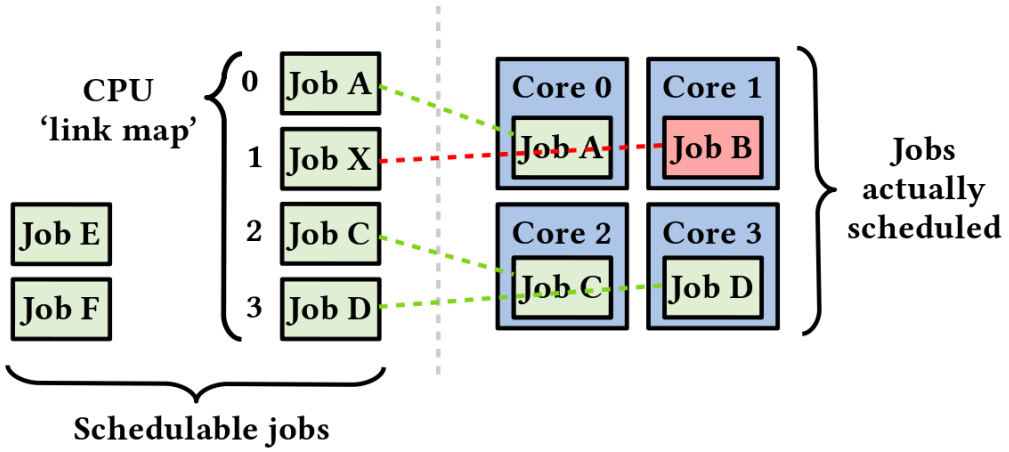

Figure 6.9: Link scheduler CPU associations

An illustration of this is provided in Figure 6.9. In our implementation, only jobs which are currently in the ready queue may be linked to a CPU. Once a job is linked, it is removed from the ready queue, and will preempt the currently scheduled job on that CPU at the next possible opportunity. In Figure 6.9, jobs A, X, C & D have been removed from the ready queue and are linked to CPUs, but jobs E & F remain in the ready queue. Additionally, job B is in a non-preemptable state – so even though job X is linked to CPU 1, it cannot be scheduled until job B becomes preemptable. This approach has a few key advantages – outlined below:

- Since the link map is stored using a priority queue, if a job arrives, it is a cheap to check whether this job is higher priority than the lowest priority currently scheduled job, and replace the link associated with the lowest priority job. Thus, the ordering is reversed compared to our other queue structures.
- It is simple to reshuffle priorities (in the event of an MCS mode-switch, for example) by unlinking all jobs of altered priority and then attempt a re-link.
- The link scheduler provides a clear abstraction separating the high-level scheduling decisions from how these decisions are actually executed (allowing for optimizations such as reduced IPIs).
- The approach is easily extended to facilitate non-preemptable-critical-sections (not a feature that we use in our MC-IPC implementation, however it may prove useful in further work extending this scheduler).

We will now detail how our implementation achieves the above. Note that in this section the 'link scheduler' or `linksched` represents a *library* we have written which is used *by* a per-cluster user-level scheduler (referred to simply as the 'scheduler'). Also note that our work in the following sections draws heavily from link-scheduler algorithms as outlined by Brandenburg [2011] – the reader is encouraged to explore that work for a more rigorous analysis.

CPU Map & Job Arrival

In our implementation, the CPU map is implemented as a priority queue as described in subsection 6.2.2. Observing Listing 6.3, there are 2 cases which may occur when a job arrives. In the first case there is a free CPU, so we simply link the job to a CPU. If there are no unlinked CPUs, we must check whether the job is of higher priority than the lowest priority linked job. Our code implements a simple optimization to keep track of the number of CPUs currently linked, such that the common case of all CPUs being occupied can be determined in constant time (this is checked by `get_any_unlinked_cpu()`).

Mode Switches & Priority Changes

In a mixed-criticality system, if one wants to enforce criticality modes under, for example, EDF-VD (which we covered in section 2.4.1), we need to be able to change priorities at runtime. Under link-based scheduling, this is simply a matter of unlinking the job undergoing the priority change, and attempting a re-link, as illustrated in Listing 6.4.

Providing an Interface

As pointed out earlier, it is important that the link scheduler provide a clear abstraction separating the high-level scheduling decisions from how they are actually executed. Essentially, `linksched` needs to be able to query & modify the scheduler ready queue, and act on scheduling decisions (i.e manipulate what is actually running on CPUs) – function pointers provided by the user-level scheduler.

On the other hand, the interface that the link scheduler provides *to* a user-level scheduler is shown in Listing 6.5. We cover how the C-EDF scheduler interacts with the link scheduler in subsection 6.2.4.

```c
/* To be called when a job arrives (i.e enters the ready queue) */
void on_job_arrival(linksched_cluster_t *cluster, job_t *job)
{
  cpu_t *unlinked_cpu = get_any_unlinked_cpu(cluster);
  if(unlinked_cpu) {
    perform_link(cluster, unlinked_cpu, job);
    try_to_schedule(cluster, unlinked_cpu);
  } else {
    job_t *lowest_prio_linked_job = cpu_map_last(cluster->cpu_map);
    /* If the 'attempt to link' job is of higher priority than the lowest linked job */
    if(job_compare(job, lowest_prio_linked_job)) {
      /* Unlink the old job, link the new job and try to schedule it */
      cpu_t *cpu_to_update = lowest_prio_linked_job->cpu_linked;
      perform_unlink(cluster, lowest_prio_linked_job, true);
      perform_link(cluster, cpu_to_update, job);
      try_to_schedule(cluster, cpu_to_update);
    }
  }
}
```

Listing 6.3: Algorithm for handling job arrival in `linksched`.

```c
void
linksched_on_priority_change(linksched_cluster_t *cluster, job_t *job)
{
  if(job->cpu_linked) {
      /* Return to ready queue */
      perform_unlink(cluster, job, true);
  }
  /* Job may be re-linked straight away (or not) */
  link_next(cluster);
}
```

Listing 6.4: Function to handle job priority changes in `linksched`

An example of a simple optimization to reduce IPIs can be found in `on_job_departure()`, as seen in Listing 6.6. If the core which the old job was departed from still has a (stale) job on it after `link_next()`, then a new job wasn't scheduled. Only in that case we want to go idle on this core. This avoids immediately forcing a CPU to go idle when a scheduled job departs, in case a new job becomes schedulable and it is possible to perform a single IPI (i.e call to `schedule_job()`, which can be expensive for remote CPUs).

Link Scheduler Limitations

If a job from a task was previously scheduled on a particular core, it is desirable to assign new jobs from the same task to that core for cache affinity reasons. Under G-EDF, the only way this

```c
/* Called by scheduler when job becomes ready */
void on_job_arrival(linksched_cluster_t *cluster, job_t *job);
/* Called by scheduler when job budget expires or is suspended */
void on_job_departure(linksched_cluster_t *cluster, job_t *job);
/* Called by scheduler on e.g a criticality mode switch */
void on_effective_priority_change(linksched_cluster_t *cluster, job_t *job);
```

Listing 6.5: Functions supplied by `linksched` to a scheduler.

```c
linksched_on_job_departure(linksched_cluster_t *cluster, job_t *job)
{
  cpu_t *cpu_before_departure = job->cpu_scheduled;

  if(job->cpu_linked) {
      /* Don't return to ready queue! */
      perform_unlink(cluster, job, false);
  }
  link_next(cluster);

  /* Check for idle last to avoid unnecessary IPIs */
  if(cpu_before_departure->job_scheduled == job &&
     cpu_before_departure->job_linked == NULL) {
    /* Go idle on that core */
    schedule_job(cluster, cpu_before_departure, NULL);
  }
}
```

Listing 6.6: Algorithm for handling job departure in `linksched`.

heuristic does not compromise scheduler correctness is in the event of an arbitrary tiebreak – something that would only occur frequently if task periods are harmonic (i.e their periods are multiples). In general, our test task sets are not harmonic (they are randomly generated). Additionally, under C-EDF we choose clusters to share L2/L3 caches where possible – which reduces the adverse effects of having no cache affinity heuristic within a cluster. As a result, although we do place jobs on the same core as the user-level scheduler if *all* cores are idle, we do not implement a cache affinity heuristic in the event of tiebreaks for clusters.

Another limitation of our link scheduler is that there are some situations where it does not prevent unnecessary IPIs. For example – if a job is linked with almost zero budget remaining on a remote core this may cause an `on_job_arrival()` call to be quickly followed by an `on_job_departure` (so quickly that the job makes no progress). Thus, `linksched` relies on the user-level scheduler to avoid such cases.

6.2.7 Scheduling a Job

Now we will describe how scheduling decisions are acted upon, and how reply objects are managed. Although the link scheduler module (subsection 6.2.6) assigns jobs to CPUs, it utilizes a callback from the user-level scheduler to act on its decisions.

Initially, every task has a TCB and associated scheduling context – managed by the user-level scheduler. When a job is scheduled by the link scheduler, the user-level scheduler will `seL4_SchedControl_Configure` the associated scheduling context such that the job is on the correct CPU, before executing a `YieldTo()` (or `YieldToTimeout`). In our implementation, we desired (in the `YieldToTimeout` case) that rescheduling callbacks were all executed in the same place – so that multiple (comparatively expensive) `YieldToTimeout` calls did not have to be made. This is an option which we implemented also, by having the scheduling callback populate a CPU map of 'scheduling contexts to change', and inserting a new `actually_schedule_jobs()` call (logically between lines 13 and 16 of Listing 6.2).

With respect to line 16 of Listing 6.2, one might ask how reply objects are managed. There are 2 situations where a reply object will be populated – when a task makes a resource request, and when a timeout exception arrives. Unfortunately, it is not possible to make a simple 1:1 association between reply objects and tasks on system initialization, because it is impossible for a user-level scheduler to know ahead of time which task will populate the reply object (when a cluster consists of >1 CPU). To solve this, the user-level scheduler is allocated $N_{tasks} + 1$ reply objects. Every task is associated with one of these objects, and the final reply object kept for the user-level scheduler. When the user-level scheduler makes its scheduling decision and blocks on an event, it will `seL4_Recv` on the reply object associated with the user-level scheduler. Once an event arrives, the badge of the response indicates which (if any) task the event came from. If the event did come from a task, then the reply capability pointer associated with the user-level scheduler is swapped with that of the task from which the event came (which will be empty). As a result, the rest of the scheduling logic is able to make the assumption that the reply capability pointers associated with tasks correspond to the correct reply object capabilities.

6.3 MC-IPC Implementation

This section explores how we augmented our C-EDF user-level scheduler to support MC-IPC. We previously covered MC-IPC basics in chapter 3 and subsection 5.2.2. Our MC-IPC implementation modifies the C-EDF scheduler (and other components) to include the following features:

- **Blocking semantics** – tasks must be able to IPC to user-level schedulers in order to make resource requests, and have the scheduler respond appropriately (by, for example, scheduling an alternative task until a request is complete). This was achieved by adding an additional task state (as seen in Figure 6.7).
- **Idling reservations** – MC-IPC (as part of its proof) requires that reservations associated with tasks consume budget even *when that task is blocked*. Since scheduling contexts do not currently support such a model, we have implemented our own reservation model at user-level which augments scheduling contexts. Essentially, this involves incrementing task budgets even when they are in the blocked state, and recycling them from the blocked to release queue with correct abort semantics on an expiry.
- **Bandwidth Inheritance** – servers must be able to inherit reservations from tasks which make MC-IPC requests. Since we are using a custom reservation model, we also had to implement our own form of bandwidth inheritance.
- **MC-IPC queueing semantics** – to maintain correspondence with the MC-IPC specification, we implemented the same queueing structures, invocation semantics and pruning rules (which rely on the correctness of the above 3 features).

6.3.1 Overview

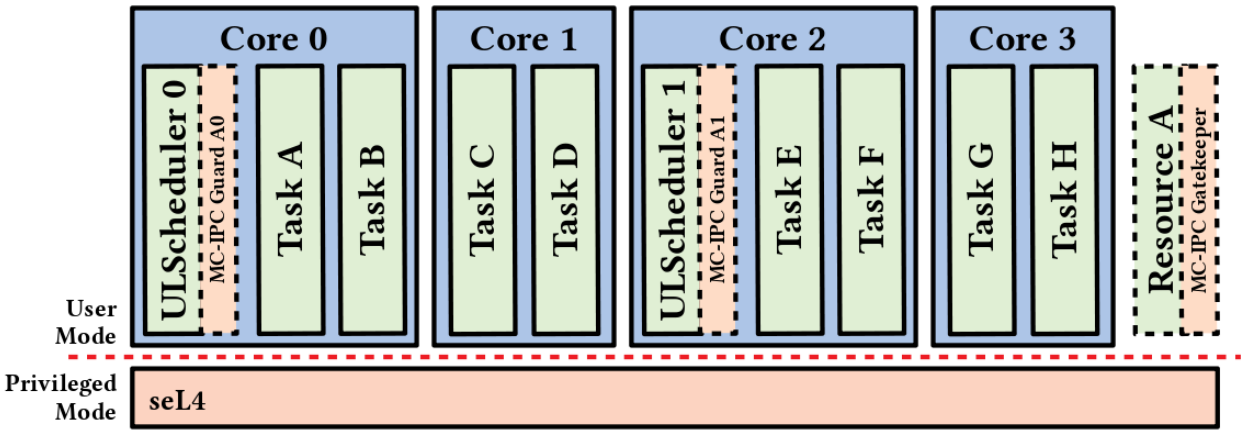

Figure 6.10: Basic 2-cluster MC-IPC structure

Our implementation maps directly to the general approach outlined in subsection 5.2.2 – which is also pictured in Figure 6.10. To make an MC-IPC request, a task will IPC (ordinarily) to its cluster-local user-level scheduler. The user-level scheduler interacts with an MC-IPC *library* (in its address space), which modifies some cluster-local queues in accordance with the MC-IPC semantics. Then, the user-level scheduler is responsible for modifying the global MC-IPC queue (if there is space), and setting up bandwidth inheritance for the server (if appropriate). Communication of an MC-IPC request from user-level schedulers to an MC-IPC server is implemented using shared memory and semaphores (see subsection 6.3.2); since it is not possible to arbitrarily prune ordinary IPC requests as required by MC-IPC (we also considered high-priority 'cancellation' messages or using `seL4_CancelBadgedSends`, but ordinary synchronisation primitives were simpler). Once the server completes an MC-IPC request, it communicates the result back to the relevant user-level scheduler, which in turn reverts bandwidth inheritance (if appropriate), and then proxies the MC-IPC response back to the waiting task. This process will be explored more rigorously in subsection 6.3.3, however we will first look at the high-level datastructures in more detail.

6.3.2 Per-Cluster & Per-Server Properties

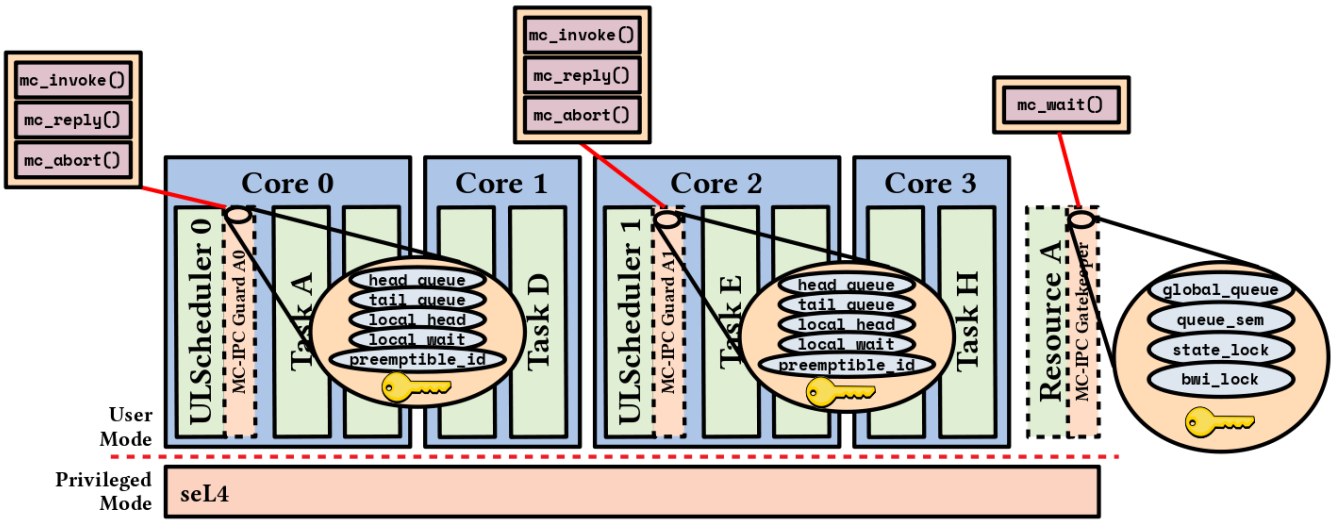

Figure 6.11: Per-cluster and per-server MC-IPC properties

An illustration of attributes which are associated with user-level schedulers and resource servers in our implementation is provided in Figure 6.11. The blue ovals associated with Resource A represent variables in shared memory which can be manipulated by Resource A or user-level schedulers. The blue ovals associated with user-level schedulers represent scheduler-local state. The `mc_...` operations displayed above the figure represent which MC-IPC operations are performed by which components. These correspond directly to those described by Brandenburg [2014], but we provide an extremely condensed summary here for convenience:

- **`mc_invoke()`** – add a request to either the local queues (`head_queue`, `tail_queue`, `local_head`) or the global queue (`global_queue`), and (if appropriate) set up bandwidth inheritance.
- **`mc_reply()`** – remove a completed request from the global queue, and (if enough space) move a request from the local queues to the global queue. Additionally, revoke bandwidth inheritance (if appropriate).
- **`mc_abort()`** – attempt to prune a request from the local or global queues, and revoke bandwidth inheritance (if appropriate). If a request could not be pruned (it is currently being processed), create 'back-pressure' by setting the `local_wait` flag (which is checked by `mc_invoke()` and `mc_reply()`).

- **`mc_wait()`** – wait on the global queue (through a semaphore – `queue_sem`), to get a request for the resource server.

Examining Figure 6.11 further, we use a spinlock (`state_lock`) to prevent race conditions between user-level schedulers accessing global state (we chose a spinlock since every mutation on global state is extremely short in our implementation). Thus far, in addressing Figure 6.11, we have not described the purpose of `preemptible_id` or `bwi_lock`. The purpose of these will become clear shortly.

6.3.3 Walkthrough of an MC-IPC Request

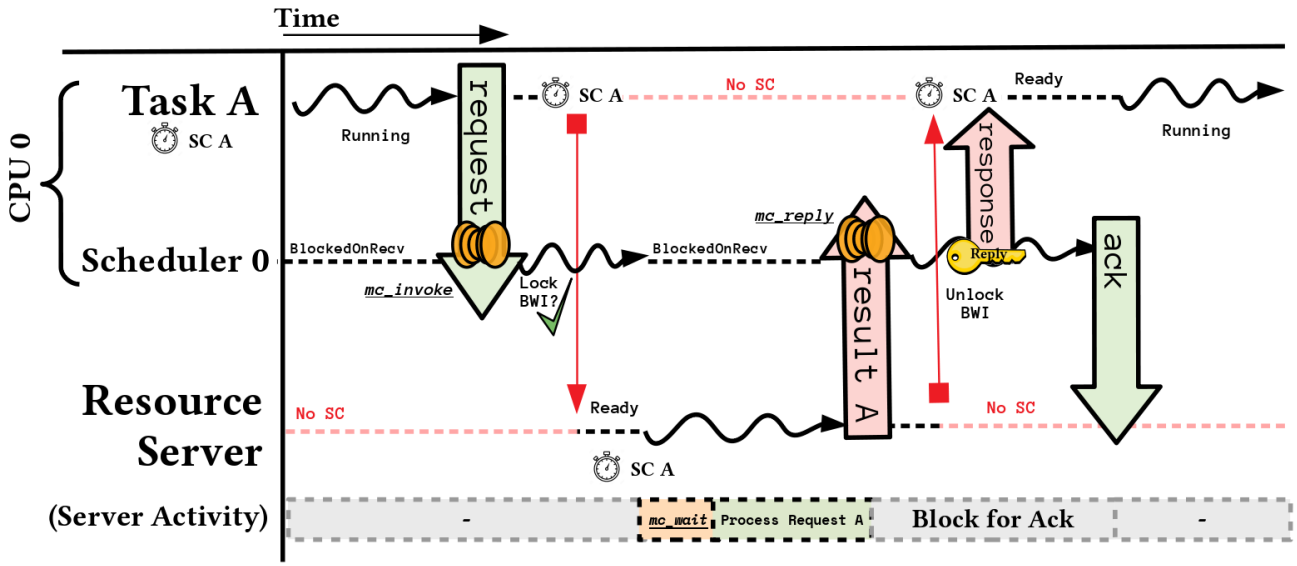

Figure 6.12: A single uninterrupted MC-IPC request

An illustration of what happens during an MC-IPC request in our implementation is provided in Figure 6.12. This example illustrates a simple single-core scenario for the sake of clarity. Initially, Task A has been selected by Scheduler 0 (and is running on its SC), the resource server has no clients, and all the MC-IPC queues are empty. Thereafter, the following events occur:

1. Task A makes a resource request, by invoking an `seL4_Call` on its badged copy of the user-level scheduler's endpoint.

2. Scheduler 0 receives the request, determines that it is an MC-IPC request, and saves the reply object for Task A.

3. Scheduler 0 performs an `mc_invoke`, which (since there is no competition) adds the request straight to the global queues.

4. Scheduler 0 performs a non-blocking lock on a global `bwi_lock` associated with the server. It succeeds, which indicates that this cluster is permitted to schedule the resource server (i.e no other clusters are currently scheduling the resource server through bandwidth inheritance).

5. Scheduler 0 unbinds the scheduling context from Task A, gives it to the resource server, and yields to Task A's SC (which causes the resource server to run).

6. The resource server invokes `mc_wait` and pops Task A's request off the global queue.

7. The resource server processes Task A's request, and IPCs the result of the request back to Scheduler 0 (note that *responding* over IPC does not break the MC-IPC assumptions, since the requests have all already been correctly serialized and pruned). Additionally, the resource server blocks for an acknowledgement from the scheduler, since it requires that the server perform an `mc_reply` before the next time it wakes up (such that the queues maintain their invariants).

8. Scheduler 0 performs an `mc_reply` (to update the MC-IPC queues), gives Task A's SC back, IPCs the response to Task A, unlocks `bwi_lock` (so that other schedulers can now schedule the resource server), and sends an acknowledgement back to the resource server. Since no other schedulers are waiting to schedule the resource server, Scheduler 0 does not have to send messages to any other schedulers notifying them that the resource server has become free.

9. Scheduler 0 yields to Task A's SC, scheduling task A.

10. Task A receives the MC-IPC server response as an IPC and continues executing.

As we have seen, our implementation requires there be considerable protocol complexity involved with even a simple MC-IPC request. We argue that this complexity is required to support clusters on multiple cores – something we will cover in the next section.

6.3.4 Multi-cluster Bandwidth Inheritance & Idling Reservations

In this section, we address how our design facilitates bandwidth inheritance and idling reservations across multiple clusters. The simplest way to understand how our design operates is, again, through example – however we will first provide a summary of one of the main issues concerning bandwidth inheritance.

A key problem we had to solve is figuring out how to notify other clusters who *would* have scheduled a resource server if it were available, once the server's reservation on a local cluster expires. In our implementation, this is the purpose of an atomically-incremented per-cluster variable – `decision_id`. Whenever a new scheduling decision is made on a cluster, that cluster's `decision_id` is incremented. Importantly, if *any* task of higher priority than the currently scheduled set is blocked on a resource server, a globally-visible `preemptible_id` variable is also set to the current cluster's `decision_id`. Every resource server has a `preemptable_id` associated with every cluster. This allows clusters (who are reverting bandwidth inheritance) to:

1. First, check if a local task is able to donate bandwidth (as this is cheaper than notifying remote cluster)

2. If this fails, walk all the remote cluster's `preemptible_id` and `decision_id` pairs. If the two are equal, we have found a cluster which *would* have scheduled a resource server if it were available. Wake it up by sending it a 'server idle' message.

3. If no clusters wish to donate bandwidth to the server, do nothing.

Note that this approach is similar to that used in LITMUS^{RT} for the same purpose, however our system of course consists of communicating servers rather than a single monolithic kernel using synchronisation primitives for communication. There are some obvious race conditions involved here – for example, what happens if a preemptable cluster is found, but by the time our message arrives it is no longer preemptable? In that specific case, our implementation will simply propagate the idle message to the next preemptable cluster.

To further solidify some of these concepts, let us consider an example of multi-cluster bandwidth inheritance.

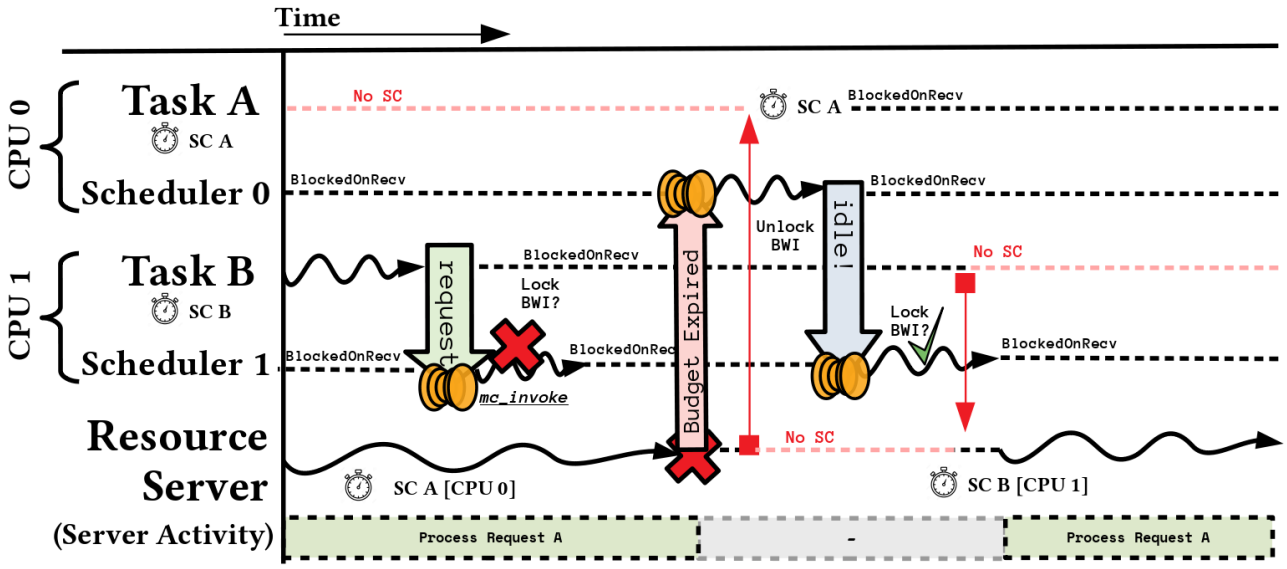

Figure 6.13: Bandwidth inheritance example

A simple example of bandwidth inheritance with multiple clusters in our implementation is shown in Figure 6.13. Initially, it is assumed that Task A previously made an (unpruned) MC-IPC request, Scheduler 0 has scheduled Task A's SC (such that the resource server is running), and Scheduler 1 is currently executing Task B. Thereafter, the following events occur:

1. Task B makes an MC-IPC request

2. Scheduler 1 performs an `mc_invoke`, adding Task B's request to the global request queues (there is no competition in cluster 1).

3. Scheduler 1 performs a non-blocking lock on `bwi_lock`, and fails. Thus, Task B is moved to the 'blocked' queue, and budget will continue to be charged from Task B even though it is not scheduled. Additionally, the local `decision_id` is incremented, and `preemptable_id` set to the same (for the correct resource server). Since no other tasks are schedulable on CPU 1, the CPU goes idle. (note that if another task was schedulable, Scheduler 1 would have now scheduled that in place of Task B)

4. Budget expires on the reservation associated with the resource server, which belongs to Task A on CPU 0.

5. Scheduler 0 returns the SC, unlocks `bwi_lock`, and checks for other bandwidth inheritance candidates in its own cluster. It finds none, so sends an idle message to the next

preemptable cluster for the resource server it just abandoned – which is Scheduler 1. CPU 0 goes idle as there is nothing left for Scheduler 0 to schedule.

6. Scheduler 1 receives the idle message, attempts to lock `bwi_lock` and succeeds. It then re-binds Task B's SC to the resource server, and yields to it.

7. The resource server continues processing Task A's request. At some point in the future, Task A's request will complete, and the server will then begin processing Task B's request. (recall that MC-IPC servers are assumed to be non-reentrant).

6.3.5 MC-IPC Limitations

Our implementation inherits some limitations of MC-IPC, namely that server requests cannot be nested, and that servers are assumed to be non-reentrant. Additionally, since a large amount of MC-IPC state is contained in the user-level scheduler, high-criticality tasks have a dependency on the correctness of user-level schedulers (both local and remote) for their invocation bound to be adhered to. Perhaps the most crucial limitation of our implementation, however, is that this implementation (due to time constraints) is constructed purely on existing kernel primitives (excluding the C-EDF scheduler and logging infrastructure). In an ideal world, the MC-IPC implementation we present would have made heavier use of seL4-MCS constructs such as scheduling context donation and ordinary priority-ordered IPC – however the slight difference between MC-IPCs assumptions and the primitives supplied by seL4-MCS made this impossible in our case. Fortunately, our work on implementing MC-IPC in the manner described here provided us insight into some changes to the seL4 model which would make a future implementation much simpler. These modifications and a new, candidate MC-IPC architecture that would prove interesting future work is described in chapter 8.

6.4 Summary

This chapter detailed each component of our implementation – namely tracing & logging, our C-EDF scheduler implementation, and our resource sharing implementation. We explored how each of these components interact, examples of how they achieve their aims, and limitations of our approach. In the next chapter, we will evaluate & discuss how well our implementation performs in practice.

7 | Evaluation & Analysis

This chapter details our experiments and how we interpret them. Our overall goal is to determine whether the scheduling architecture we have constructed is performance-competitive with a similar-class monolithic system. To that end, we start by describing how our tracing infrastructure accounts for overheads, whether it is accurate, and the applicability of our tracing tool to other domains. Then, we shall explore the properties of our C-EDF scheduler – the impact of different time sources, scalability concerns, criticality modes and overhead sources. We perform a schedulability analysis on our implementation to compare it with other (monolithic) systems. Finally, we conclude this chapter by performing functional tests on our MC-IPC implementation.

7.1 Evaluation Hardware

Platform	**CPU Family**	**N CPUs Clock**	**L1** (KiB) **Assoc.**	**L2** (KiB) **Assoc.**	**L3** (KiB) **Assoc.**
x86 (Optiplex 990)	i7-2600	**4**	32I+32D	256	8192
32-bit	Sandy Bridge	3.4GHz	8-way	8-way	16-way
ARMv7 (Sabre)	Cortex-A9	**4**	32I+32D	1024	✗
32-bit	NXP i.MX6	1.0GHz	4-way	16-way	✗

Table 7.1: Hardware platform basic details.

Throughout this chapter, we perform experiments on the hardware platforms shown in Table 7.1. On both platforms, last-level caches are shared, other levels core-local. On x86, hyperthreading & dynamic voltage frequency scaling were disabled.

7.2 Tracing Infrastructure

7.2.1 Log Overhead Accounting

Our evaluation generally makes use of conventional trace points & timestamps, and log buffer traces for longer timespans or for events which are difficult to trace directly. As using the log buffer has a performance impact, it is important to be able to quantify and account for overheads in any `schedplot` measurements. A visual overview of what a recorded timestamp by the kernel logging infrastructure actually measures is provided in Figure 7.1.

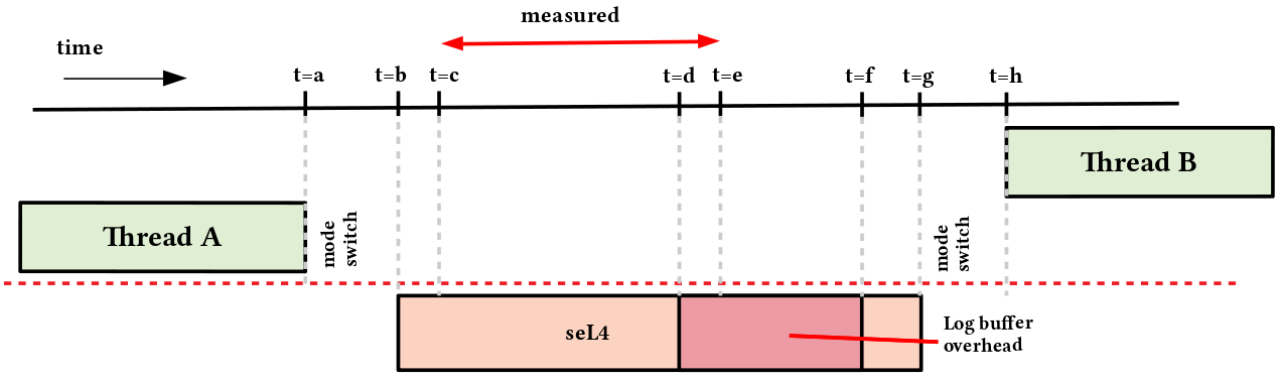

Figure 7.1: Measured event duration by kernel logging infrastructure (not to scale)

This figure illustrates a simple context switch between 2 user-level threads, where Δt_{ab} and Δt_{gh} represent mode switches into and out of seL4. Δt_{bc} represents time between when the mode switch completes, and the kernel entry timestamp is read. Δt_{cd} represents time from this first timer read until the log buffer code begins executing. Δt_{de} is the time between when the log buffer code starts, and the'exit' timestamp is read. Δt_{ef} represents the overhead of creating and writing a log entry. Δt_{fg} is the time that passes between when the log buffer code finishes executing, and the mode switch is actually initiated.

In particular, note that Δt_{ce} is the in-kernel time recorded by the log buffer, and Δt_{bg} is the real kernel entry time. `schedplot` uses an approximation to allow the user to make timespan measurements by counting kernel invocations between 2 points and subtracting the log buffer overheads for each invocation. This allows a user to establish an estimate as to what the length of such a timespan *would* have been if logging infrastructure were not present.

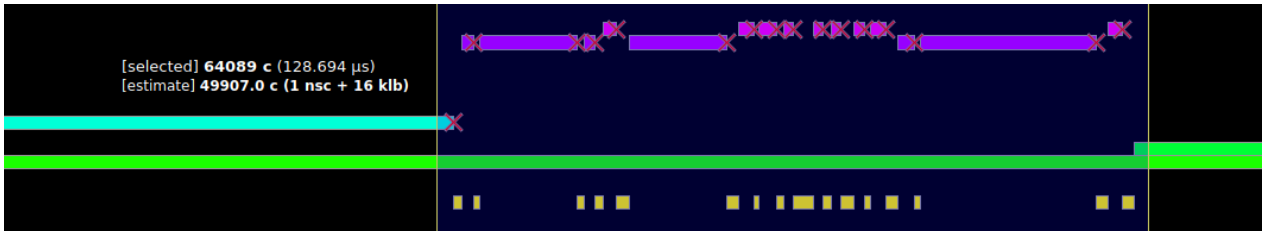

Figure 7.2: multi-task system, kernel invocations in yellow (`schedplot`)

Consider a simple scenario (pictured in Figure 7.2) - with a timespan t_{meas} where both endpoints are in userspace tasks, and assuming a single core situation and no interrupts occur within the timespan. If there are N kernel entries (i.e $2N$ mode switches) between the endpoints, then the same execution trace *without* kernel logging would have taken $t_{meas} - N\Delta t_{df}$ units of time. It becomes more difficult to provide an estimate once endpoints approach the edges of kernel entries, since we have to account for mode switch times and the values of Δt_{bc}, Δt_{de} and Δt_{fg} (but only in the kernel entries closest to the endpoints). `schedplot` accounts for mode switch times, but assume these other factors are negligible. We argue this is a valid approximation, as the feature is intended for a user to make timespan measurements over periods of time spanning tens to hundreds of kernel entries, rather than for microbenchmarks. The worst-case accuracy of this approximation will be explored further in subsection 7.2.3.

We reiterate that `schedplot` corrected-timespan measurements are only valid in certain situations – for example, they are only permissible when the first entry (or no entries at all) come from an interrupt, because subtracting overheads does not make sense if any invocation was held up by a temporal event. Additionally, if invocations on multiple cores are involved, one cannot simply subtract log buffer overheads since kernel instances may IPI to each other. Generally, in our experiments, we use conventional tracepoints & the cycle counter where possible, and an overhead-accounted scheduling dump as a last resort. The precise method depends on the experiment and will hence be described alongside each one.

7.2.2 Logging Overheads

To quantify the impact of our log buffer changes, and establish baseline values Δt_{df} (the overhead of the log buffer), we compared the cost of performing a null system call, with & without logging enabled, in various modes.

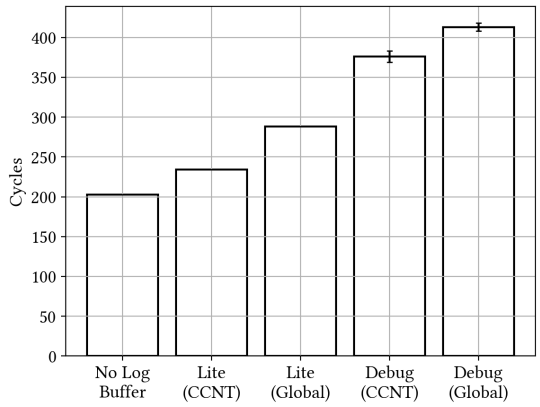

Figure 7.3: ARM log buffer overheads

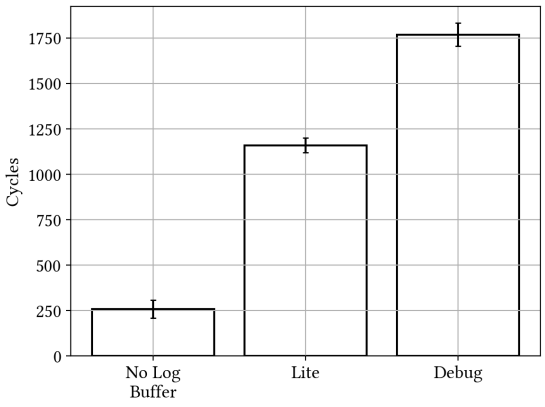

Figure 7.4: x86 log buffer overheads

Our results are shown in Figure 7.3 and Figure 7.4. Each result indicates the number of cycles (measured from userspace) it takes to perform a null system call, by taking a timestamp before and after the call, and subtracting the measurement overhead. The results are formed from an average of 1000 tests (after warmup), and errors bars represent 1 standard deviation.

On ARM, the overhead added by the fastest logging mode is only 34 cycles, however on x86 it is much higher at 902 cycles (we will explain why shortly). On both x86 and ARM, the debug trace mode has a much higher overhead than the minimal tracing mode – which isnt a surprise given how much more information is copied in debug mode.

On ARM, we trialled 2 methods of getting timestamps in the kernel for the log buffer. In Figure 7.3, 'CCNT' indicates the 32-bit ARM cycle counter, and 'Global' indicates the 64-bit ARM global timer. Importantly, even though the global timer has a higher overhead, it is just as deterministic as the cycle counter ($\sigma < 0.5$). As a result, we use the global timer in our experiments on ARM which use the log buffer - since it is not prone to overflow half-way through an experiment.

One may wonder why the logging overhead even in minimal mode is so high on x86. To investigate this, we re-ran the same benchmark as above, disabling different parts of the logging infrastructure.

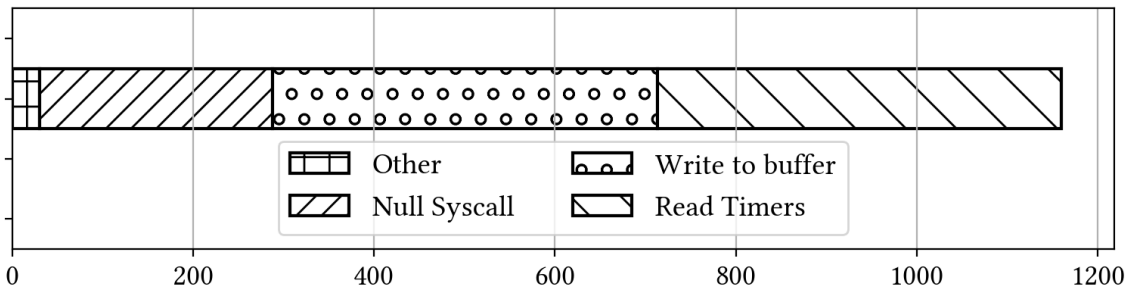

Figure 7.5: Cycle breakdown of logging overhead on x86

As shown in Figure 7.5, essentially all our overhead is attributable simply to reading the timer and the store instructions which write to the kernel log buffer (write-through mapping is responsible for this). Extra branches, CPU ID calculation, and fetching SMP state were all found to have next to zero overhead compared to these operations (and are thus encompassed under 'Other' in Figure 7.5). Although the overhead is high, what really matters is overhead-aware prediction accuracy – which we will covered next.

7.2.3 `schedplot` accounting accuracy

To evaluate how well our `schedplot`-corrected timespan approximations represent reality in the worst-case, we constructed another microbenchmark.

Platform	**System Call**	**Baseline** (cycles)	**Measured** (cycles)	**Adjusted** (cycles)	Δ_μ (cycles)	Δ_μ (%)
ARM	`Null`	199 ($\sigma = 0$)	243 ($\sigma = 0$)	206 ($\sigma = 0$)	7	3.5
	`Yield`	1076 ($\sigma = 0$)	1111 ($\sigma = 0$)	1076 ($\sigma = 0$)	0	0.0
x86	`Null`	204 ($\sigma = 50$)	1099 ($\sigma = 40$)	198 ($\sigma = 64$)	6 $\pm$64	2.9
	`Yield`	714 ($\sigma = 45$)	1621 ($\sigma = 39$)	776 ($\sigma = 60$)	62 $\pm$60	8.7

Table 7.2: Worst-case accuracy of `schedplot` log-buffer overhead adjustments, per call.

In this test, we execute a thread which makes 100 system calls one after the other, and time how long the *total* operation takes (before and after *all* the calls) using 2 different methods. Each cycle result in the table is normalized by the number of calls made by the thread, so represents a 'per call' value (not excluding the loop overhead). In Table 7.2, 'Baseline'

represents the measured length of time with all logging disabled, 'Measured' represents the same measurement with logging enabled, and 'Adjusted' shows the baseline approximation which `schedplot` made based on the log buffer contents. Δ_μ represents the difference between the means of the actual baseline and `schedplot`-approximated baseline.

Since we measure the log buffer overheads used for `schedplot` accounting using null system calls, we also tested a more complex system call, `Yield`. The behaviour of ARM was much more deterministic than x86 (we display $\sigma < 0.5$ cycles as 0), and `schedplot` was able to predict a baseline within a margin of several cycles. On x86, the `Yield` offset error is attributable to the larger values of Δt_{bc}, Δt_{de} and Δt_{fg} than on ARM – which is also why the cycle offsets are all positive.

It is important to note that in the context of the intention of this feature, the error percentages actually represent *worst-case* errors – since this benchmark timespan is completely dominated by in-kernel time. When measuring user-level schedulers, a much greater proportion of timespans is dominated by userspace threads, and so the timespan error reduces accordingly. Of course, if kernel entries are less frequent and userspace is cache-heavy, then kernel state is less likely to remain in the caches. This means that `schedplot` measurements are more accurate when the working set of userspace threads is relatively small.

To emulate what a user of the tool would do, we measured the total invocation time of a simple user-level EDF scheduler by using conventional tracepoints and through UI-selected `schedplot` measurements (30x random samples) on our x86 platform.

Baseline (cycles)	**Measured** (cycles)	**Adjusted** (cycles)	Δ_μ (cycles)	Δ_μ (%)
12231 ($\sigma = 1419$)	18879 ($\sigma = 3601$)	11829 ($\sigma = 2844$)	402	-3.3

Table 7.3: Error of x86 `schedplot` log-buffer overhead adjustments, simple EDF scheduler.

The results are shown in Table 7.3. The scheduler we measured makes between 7 and 10 kernel invocations depending on the event causing it to be invoked, and spends approximately 40% of its time in userspace. Note that our measured standard deviations are quite high because our measured error also includes the 'human error' of making an accurate time selection in the tool, in addition to the uncertainties introduced by subtracting log buffer overheads (which have their own standard deviations).

To summarise - the worst-case accuracy of a `schedplot` approximation is tolerable, but the real-world discrepancy will be smaller. In a situation where timespan measurements are made over periods of time spanning tens to hundreds of kernel entries, with most of that time spent in userspace – `schedplot` will provide a more-than-sufficient approximation.

7.2.4 `schedplot` debugging case study

We used `schedplot` throughout this thesis to debug & measure scheduler performance, however were curious whether the tool might be applicable to other domains.

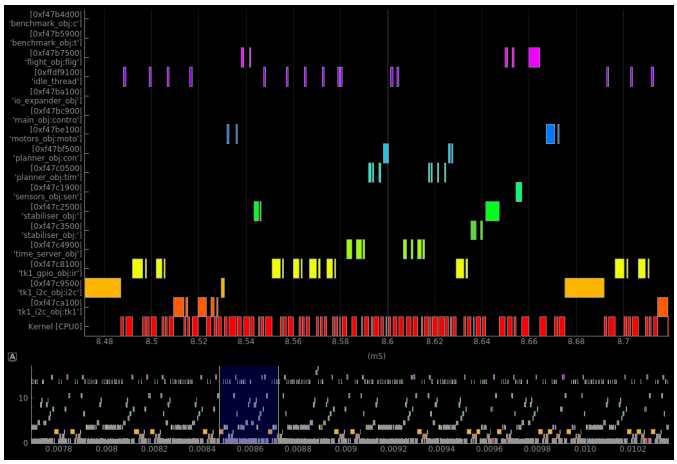

Figure 7.6: Quadcopter case study `schedplot` trace

As part of another project within the research group, a quadcopter flight-control loop was in development. We were attempting to diagnose a timing issue which spuriously caused the flight control loop to run at a lower loop rate than normal (150Hz instead of the designed 200Hz). The flight control code was based on *CAmkES*, which is a statically-configured, component-based architecture for connecting different software components in an embedded system on top of a microkernel. To help diagnose the issue, we applied our log-buffer changes to the kernel used by the flight control code, and additionally wrote a 'benchmarking' CAmkES component, which allowed the results of a scheduling trace to be emitted and analyzed by our `schedplot` tool. An example of such a trace is shown in Figure 7.6. By analyzing the results of our scheduling trace, we were able to quickly diagnose that the problem was caused by a locking race condition in the I^2C driver component, thus causing that part of the software to block the rest of the system longer than it should.

7.3 C-EDF Scheduler

In this section we will explore how our C-EDF scheduler performs. We will first look at what the main overheads of the scheduler are, discuss scalability issues, measure the key properties of our scheduler, and finally compare it with existing schedulers (The C-EDF schedulers in LITMUS^{RT} and RASP). Our work in this section focuses predominantly on the x86 platform, since our comparison scheduler only runs on this platform – however we supply some similar ARM measurements in Appendix C. It is important to note that on both x86 and ARM, last-level caches are shared, other levels core-local. This means that the 4x1 C-EDF (i.e P-EDF) and 1x4 (i.e G-EDF) cluster arrangements are most appropriate from a memory hierarchy standpoint, although the *correctness* of our 2x2 C-EDF implementation can still be tested on these platforms.

7.3.1 Dissecting the Total Invocation Time

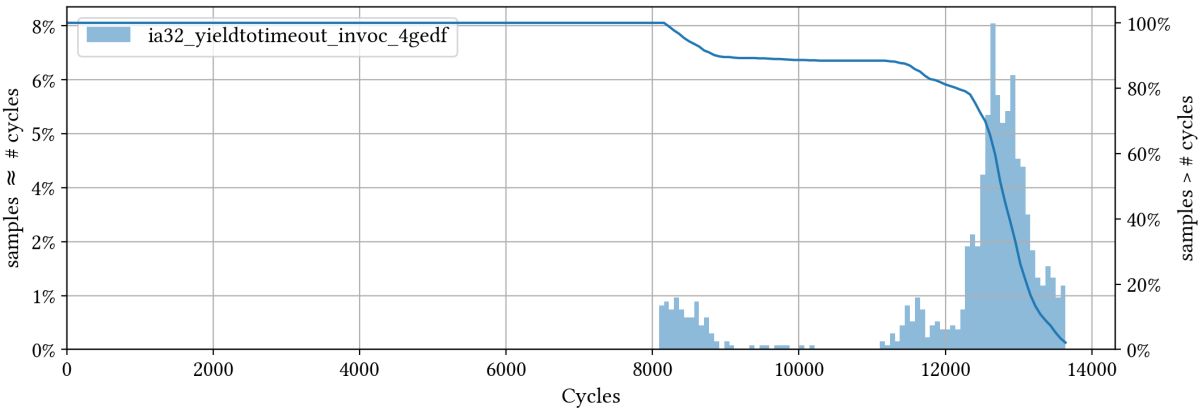

Figure 7.7: x86 4-GEDF scheduler invocation time

Many of the measurements we make in the following sections take the form of probability histograms, as shown in Figure 7.7. This histogram shows the total invocation time of a user-level G-EDF scheduler, as measured using tracepoints. The scale on the left, which is associated with the histogram plot, represent how probable it is that the scheduler takes a particular amount of time to be invoked. The scale on the right, in addition to the line plot, represents the *cumulative* probability of invocations which took *longer* than the number of cycles on the x-axis. In all invocation traces, we trace the time elapsed between when seL4 returns control to the user-level scheduler (i.e just after the `seL4_Recv` in Listing 6.2), and just before the scheduler returns control to userspace (i.e just before the `seL4_Recv`), and

subtract the measurement overhead. Hence, this histogram excludes the 2 mode switches and kernel invocations which occur on entry and exit of the user-level scheduler. For the sake of brevity, all of our histograms are taken on a `randfixedsum`-generated 10-task system (10 total for G-EDF, 10 per core in P-EDF case), with 1-1000ms random periods, over a period of 30 seconds unless otherwise specified. The tasks themselves are CPU-bound busy loops and require preemption once they consume their timeslice – this models the behaviour of `rtspin` as provided by LITMUS^{RT}.

From the trimodality of Figure 7.7, it is possible to directly see the impact of different code-paths on the scheduler invocation time. In this example, the left-most peak represents the codepath which is executed when a job is released, but is still of lower priority than any scheduled job – which results in no changes to the currently scheduled task set. In that case, the user-level scheduler only has to update its scheduling queues, read the current time, and reprogram the timer for the next budget expiry or job release. The two right-most peaks indicate the extra time required when the scheduler must change the currently-scheduled task set (i.e on release of a short-deadline job). There is an observable difference between changing the scheduled task set on local cores, versus changing it on remote cores, as seen by the separation of the 2 peaks.

7.3.2 Overheads of Time Sources

We found that one of the most significant overheads encountered by our user-level scheduler in both the P-EDF and G-EDF cases was reading & reprogramming the timer. As a result, we implemented a number of time sources (these were previously covered in subsection 6.2.5).

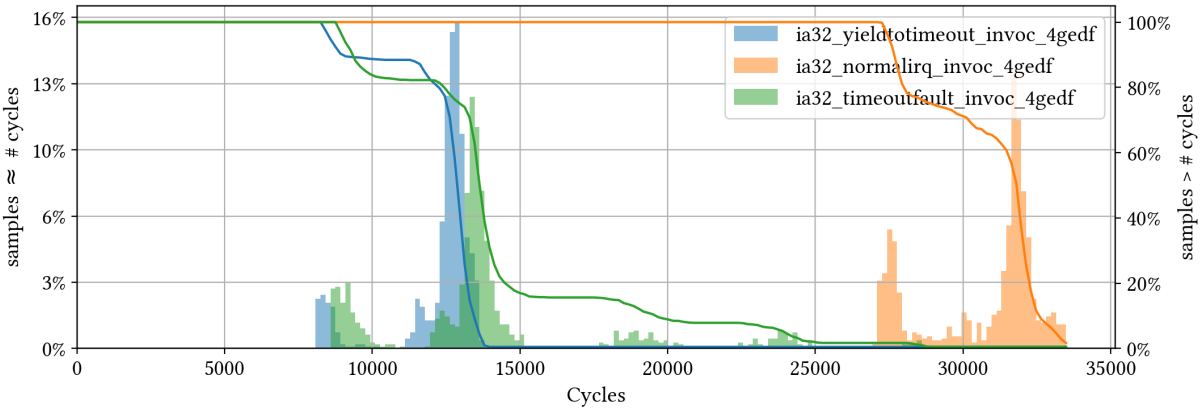

Figure 7.8: x86 4-GEDF invocation times with different time sources

A comparison of 4-GEDF time sources is shown in Figure 7.8. We observe that the cost of a user-space timer driver, labelled 'normalirq' (specifically, the High Precision Event Timer (HPET) on x86) is quite high compared with other time sources. One key reason is that an explicit kernel invocation is required to acknowledge the IRQ, which is not required by the other time sources shown. The other reason is that timer operations on the HPET are generally slower than the same operations within the kernel, as the kernel employs the Time Stamp Counter (TSC) for generating interrupts – which has a lower overhead. Virtualizing the timer driver such that userspace may indirectly exploit the TSC would have somewhat mitigated the slowdown, but we did not implement this it would still require more kernel invocations than the other time sources we describe here, and would have been much more work to implement.

Again, observing Figure 7.8, recall (from subsection 6.2.5) that `YieldToTimeout` allows a user-level scheduler to schedule a thread, set a timeout, wake up, and get the new time in a single `YieldToTimeout` & `Recv` combination. The speedup resulting from this change is quite dramatic, as minimal kernel invocations are required, and the user-level scheduler indirectly makes use of the TSC. Interestingly the `YieldToTimeout` approach has lower overhead than the timeout fault approach. The extra cost associated with the timeout fault approach is likely attributable to the extra reprogramming of a scheduling context deadline that this required by that approach.

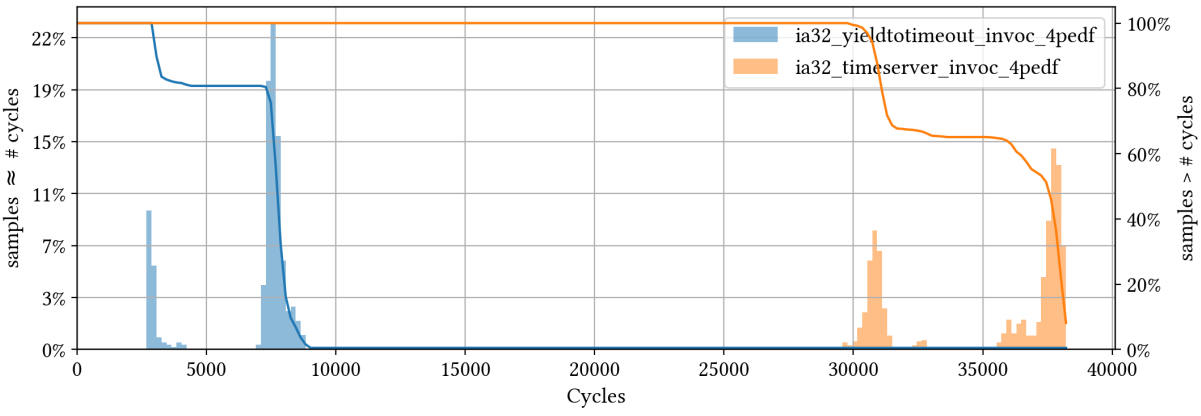

Figure 7.9: x86 4-PEDF invocation times with different time sources

A comparison of the 2 available 4-PEDF time sources is shown in Figure 7.9, measurements made on all clusters in the system. One can observe that in this P-EDF case, the average

invocation latency in the `YieldToTimeout` case is lower than that in the G-EDF case – because the scheduler no longer has to migrate tasks between cores. In the time-server case, note that the time-server thread itself is on the first cluster, so the first cluster will suffer capacity loss whenever other schedulers use the time-server. In a real system, it would be possible to mitigate the unpredictable capacity loss in the first cluster by simply placing the time-server on its own core, but this of course comes at the cost of sacrificing an entire core. Additionally, whenever a cluster uses the time-server but makes a request at the same time as another cluster, time-server invocation time may increase significantly. To investigate this time-server behaviour further, we ran another test.

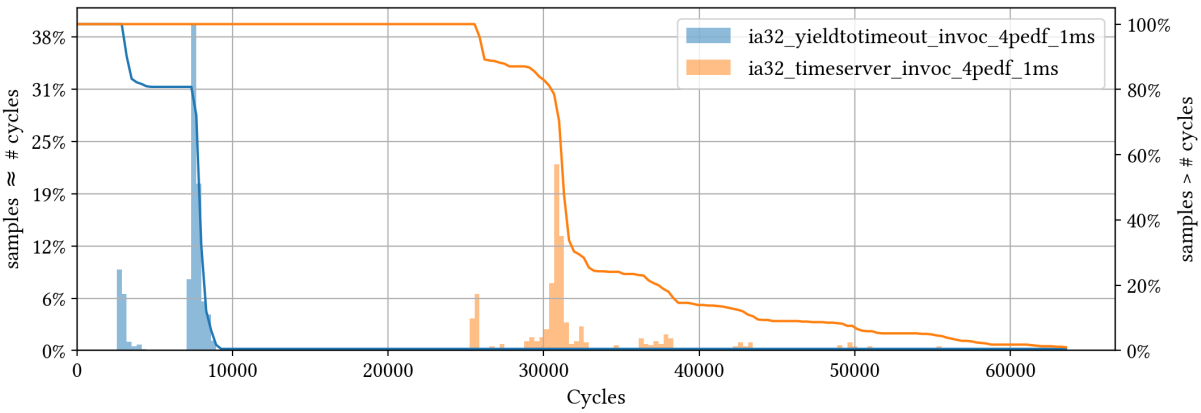

Figure 7.10: x86 4-PEDF time sources with 1ms average task periods

We can clearly see the effect of timeserver conflicts when comparing the effect of using a timeserver as shown in Figure 7.10, when we reduce average task period from 100ms to 1ms. There is a large probability tail when using the timeserver approach – which is caused by these conflicts. In the `YieldToTimeout` case, we see no such change.

To summarise, the `YieldToTimeout` approach has been measured to be faster than the other time sources which we prototyped. As a result, we use this approach in the following sections.

7.3.3 Scheduler Scalability

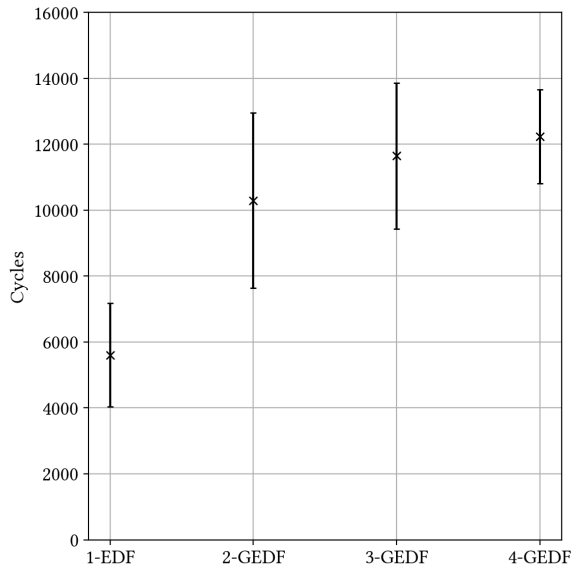

Figure 7.11: x86 G-EDF core scalability

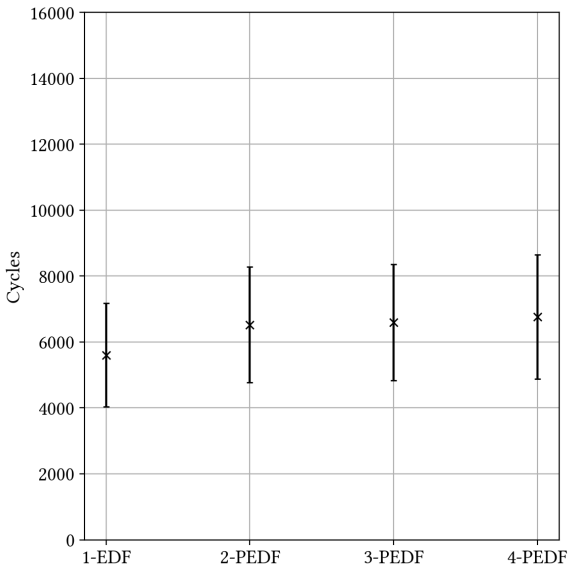

Figure 7.12: x86 P-EDF core scalability

First, we evaluate how our scheduler scales in both P-EDF and G-EDF modes with different core counts. In both Figure 7.11 and Figure 7.12, we evaluate on 16-task systems of average task period 100ms, and measure how the scheduler invocation time changes with an increase in core count.

The G-EDF case trends toward a limit. The main reason for this limit is that as we increase the core count under G-EDF, the proportion of task migrations to or from remote cores increases – i.e approximately 0% for 1-EDF, 50% for 2-GEDF, 66% for 3-GEDF, 75% for 4-GEDF. Under 4-GEDF, the majority of scheduling operations involve remote cores – which explains why the error bars become smaller (one code path is executed more often than others). Under the hood, these remote rescheduling operations involve a scheduling context invocation, after which the kernel will IPI to the relevant core and complete the migration – so there is not much opportunity for optimisation beyond what already exists (unless, of course, the remote migrations were removed altogether – but that violates G-EDF correctness).

In the P-EDF case, there is statistically insignificant change in scheduler invocation time over core count, except for an initial small jump in invocation time between the 1-EDF and 2-PEDF cases. We observed no change in the probability histogram shape between 1-EDF and 2-PEDF, only the average offset. To investigate this further, we also tested whether this change

was caused by cache conflicts by enabling & disabling cache alignment for our datastructures – which made no statistically significant difference. Since this small jump has no effect on our main schedulability analysis, we refrained from investigating it further. That being said, we believe that the most likely culprit is that the kernel codepaths increase in execution time when compiled in SMP mode – the 1-EDF case is compiled against a non-SMP kernel, and this would explain the constant offset.

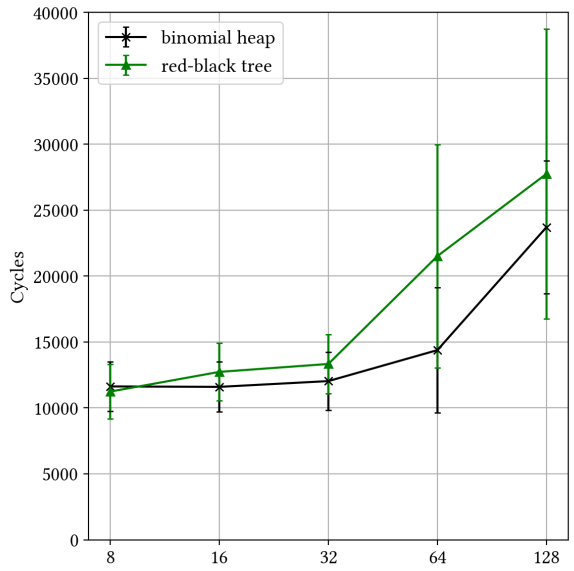

Figure 7.13: x86 4-GEDF task scalability

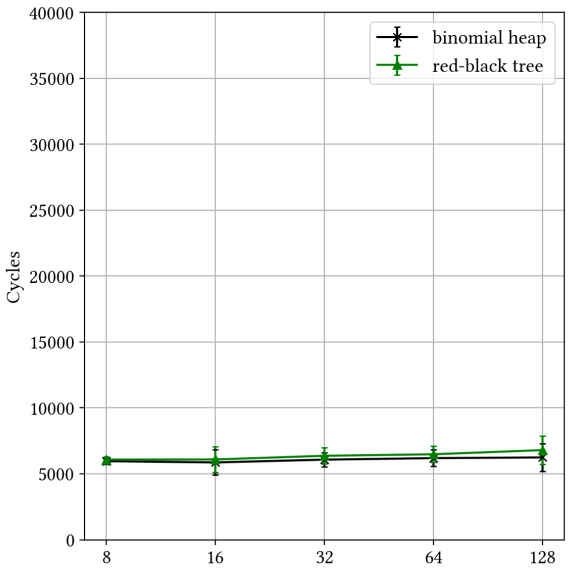

Figure 7.14: x86 4-PEDF task scalability

To test the cost on scheduler invocation time when large amounts of tasks[1] are present in the system, we evaluate the scheduler invocation time under 4-GEDF and 4-PEDF with different total numbers of tasks, and different ready/release datastructures – as seen in Figure 7.13 and Figure 7.14. Both graphs indicate the *total* number of tasks in the system on the x-axis (i.e under 4-PEDF, 128 tasks is 32 tasks per core). We test using ready/release queues backed by both binomial heaps and red-black trees, to see if there is any measurable difference between the two. On average, binomial heaps are faster than red-black trees - likely due to their constant-time minimum lookup. The G-EDF case is the most interesting, as there is an obvious performance penalty when the system contains more than 32 tasks. Once the number of tasks exceeds this threshold, the granularity of timer interrupts starts to have an effect on our results – we will now explain why. Maintaining a certain utilization bound (in our case 0.7) implies reducing the

[1]We refrain from testing more than 128 tasks simply because higher task counts caused the seL4 benchmarking infrastructure to crash, and we did not have time to debug the memory allocator.

costs of each task accordingly. Once the task costs are short enough (i.e > 32 tasks), a G-EDF scheduler may have to perform more than one context switch in a single scheduler invocation – this is why our variances increase so much. We cap our later schedulability analysis at < 32 tasks such that we measure the difference across scheduling architectures rather than the queue properties or timer granularity properties directly – the scheduling architecture itself is what we are more interested in.

7.3.4 Criticality Mode Switch

To establish the overheads involved with a criticality mode-switch, we measured the worst-case penalty on our 4-GEDF scheduler in 2 cases, under a 16-task system, as shown in Figure 7.15.

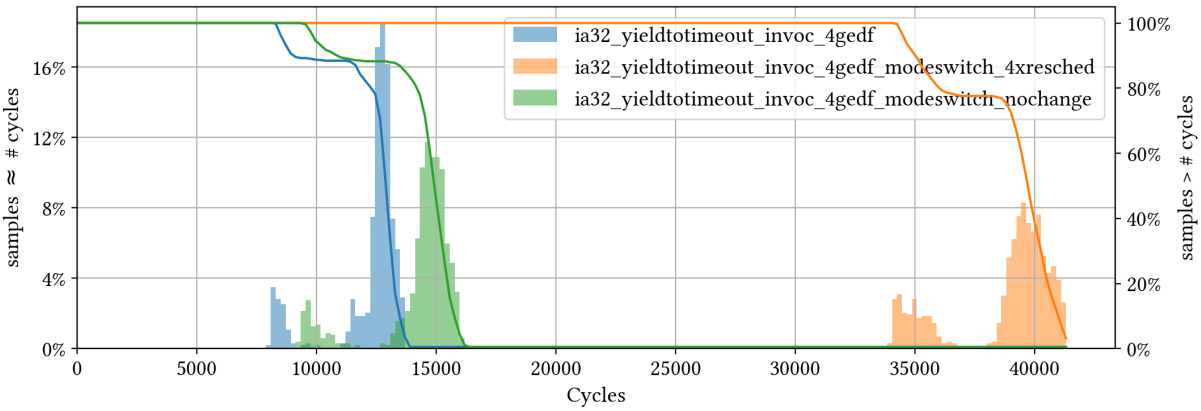

Figure 7.15: x86 4-GEDF invocation times with criticality mode switch

In both cases, our emulated mode-switch involves un-linking and re-linking all scheduled jobs using the link scheduler interface, as would happen under EDF-VD. The 'nochange' case represents when a mode switch occurs and all currently scheduled jobs remain scheduled. The '4xresched' case represents when all 4 scheduled jobs are displaced by high-criticality jobs. As expected, there is quite a large penalty on the scheduler invocation time when all jobs are displaced, as local & remote reschedules are expensive in our scheduler.

7.3.5 Overheads & Schedulability Analysis

In this section, we compare the scheduling overheads of our C-EDF scheduler with those of LITMUS^{RT} (linux-based, has an in-kernel C-EDF scheduler), and RASP (also linux-based, but

has a user-level G-EDF scheduler) – the architectures of which we briefly covered in chapter 3. Essentially, our goal is to discover whether there is any practical difference in the amount of CPU time available to real-time tasks under comparison C-EDF schedulers. To this end, we perform overhead-aware schedulability simulations, based on the overhead analyses described by Brandenburg [2011] and by utilizing their `SchedCAT` tool (which performs schedulability simulations). The overhead model of `SchedCAT` is coupled to the architecture of LITMUS, so it is first necessary for us to form a correspondence between our measurements and the `SchedCAT` model – we shall explore this now.

Scheduler Overheads

Before moving onto simulations, we will discuss which overheads we actually consider, how we measure them, and why it is valid to compare these overheads across schedulers. The overheads considered in our simulations, as required by `SchedCAT` are described below, with their LITMUS names in brackets (we will later use the short names to refer to them):

- **Context Switch Overhead (CXS):** The overhead incurred when a scheduler makes a switch between tasks, on a local or remote core *excluding* communication latency (i.e IPIs) between cores. In LITMUS, this is the length of execution of `context_switch` within the Linux kernel. In RASP, this is cost of a user-level context switch to another thread in the same address space. In our implementation, it is the length of execution of our function which switches between tasks (i.e scheduling context invocations). Note that our approach performs multicore rescheduling differently to LITMUS – we explain how this is compensated for shortly.

- **Scheduling Overhead (SCHEDULE):** The time taken to make a scheduling decision (i.e the top-level scheduling function), excluding the cost incurred when making a context switch.

- **Release Overhead (RELEASE):** When a job is released, this incurs some overhead. In all cases, this is the length of execution of the release handler, which moves the job between queues, checks whether cores need rescheduling, and performs a reschedule if required. Hence, this also includes scheduling and context switch overhead.

- **Release Latency (RELEASE-LATENCY):** The difference in time between when a job *should* have been released (i.e what time a release interrupt was programmed for), and

the timestamp when it is *actually* released. In all cases, this is measured as the difference between the last release timer setting, and the current time just before the relevant task is removed from the release queue.

- **Remote Rescheduling Overhead (SEND-RESCHED):** (Does not apply to P-EDF as there is no inter-core communication) This is the latency between when one core requests a remote reschedule, and the remote core actually receives the request. In LITMUS, this is the latency of an IPI between cores including the IRQ handler. In RASP, this corresponds to 'request overhead' in their paper – the time taken to request preemptions on remote processors through their userspace mechanism. In our implementation, we measure the latency between when a scheduling context invocation returns (as that is accounted for in context switch overheads), and when the operation takes effect on the remote processor.

A key difficulty with making comparisons between schedulers, particularly in our case; is that we have a slightly different model of multicore rescheduling than LITMUS or RASP. In LITMUS, if a release IRQ arrives on core 0, and this requires a reschedule on core 1, the kernel will send an IPI to core 1, causing a context switch to happen on core 1. In our implementation, if a release IRQ arrives on core 0 and this requires a reschedule on core 1, the user-level scheduler will perform a scheduling context invocation – which causes seL4 to IPI to itself on another core and perform the requested action (which is has a different cost to an ordinary context switch). To compensate for this, we absorb the cost of seL4 performing the remote context switch in SEND-RESCHED. Since SEND-RESCHED is directly added to the task cost in the relevant part of the overhead accounting code in `SchedCAT`, we deem this as valid. Note that the context switch cost is not 'double-charged' – the only difference is that a remote reschedule in LITMUS is an IPI (SEND-RESCHED) followed by a context switch, whereas for us, it is a context switch followed by SEND-RESCHED (where the SEND-RESCHED absorbs the IPI and cost of seL4 performing the context switch).

Another interesting question one might ask, is where are the mode-switch overheads accounted for (i.e into & out of the kernel)? As far as we were able to tell, this overhead is actually not measured by LITMUS' tracing tools, and is likely assumed to be negligible – not an unreasonable assumption for the 200-cycle overhead of a 2-way mode-switch compared to the total execution time of the monolithic kernel with scheduler (which we later measure to be in the tens of thousands of cycles). In our microkernel case, this overhead is however *not*

negligible – there are 4 mode switches and 2 kernel invocations which must be added to the user-level scheduler invocation time. We account for this by adding the cost of the entry & exit mode switches and kernel invocations to the SCHEDULE overhead.

Overhead Measurements

We perform our overhead measurements on LITMUS (using Feather-Trace), RASP (using their provided tracing scripts) and on our implementation (using sel4bench tracepoints). More details on the precise nature of our tracepoints under seL4 are provided in Appendix B. We run RASP on `PREEMPT_RT` Linux v4.9.115, use the v4.9.30 Linux rebase of LITMUS^{RT}, and apply our modifications to seL4-MCS v10.0.0. Overheads were traced on 4-GEDF and 4-PEDF systems on LITMUS & our approach, but only 4-GEDF under RASP (there is no PEDF configuration in their source distribution). In all cases, tasks are CPU-bound spin loops with as little cache footprint as possible. We use the same sequence of 1ms-1000ms period randomly-generated 16-task sets in all configurations, with 1 minute of testing per task set, and combine the results over multiple runs. Recalling the results of our task-scalability tests from Figure 7.13, we chose sets of size 16 so that our overheads are dominated by the scheduling architecture costs (rather than the overhead associated with ready/release queues), and so that a single interrupt corresponds to a single reschedule in the G-EDF common case (we aren't fighting with timer granularity). Additionally, this task count offered a good tradeoff between being able to visualize capacity loss, and reasonable simulation times (when we perform simulations later on).

All overhead figures show the mean & standard deviation of each overhead, as well as the worst measured overhead. Note that standard deviations are omitted from RASP overheads, simply because their tracing infrastructure does not provide it.

There are a few interesting things to observe from our results, as shown in Figure 7.16, Figure 7.17 and Figure 7.18 (4-GEDF) – and Figure 7.19, Figure 7.20 (4-PEDF). Firstly, the results from seL4-MCS are much more deterministic than the other approaches - illustrated by the distance between worst-case and mean. Although LITMUS & `PREEMPT_RT` are both real-time kernels, our smaller code paths, kernel cache footprint (and seL4's small worst-case execution time) are likely significant drivers of this.

In the average case, the context-switch cost under RASP is much lower than that under LITMUS or our approach – since their context switch is a relatively cheap user-space operation, and does not involve an address space switch. Our context switch cost is slightly worse than

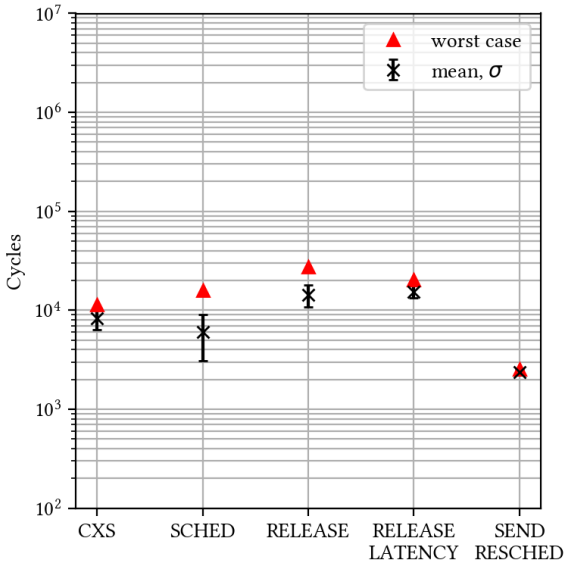

Figure 7.16: Our approach, 4-GEDF

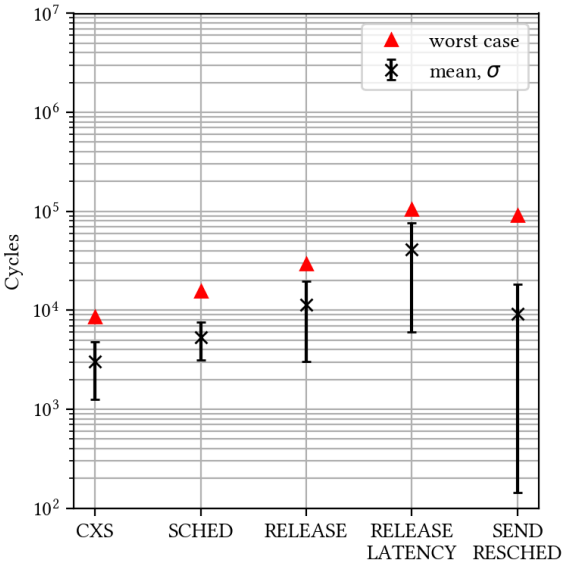

Figure 7.17: LITMUS^{RT}, 4-GEDF

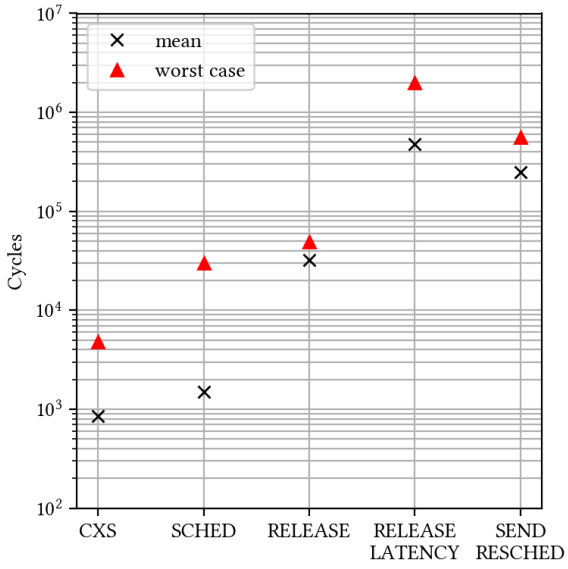

Figure 7.18: RASP, 4-GEDF

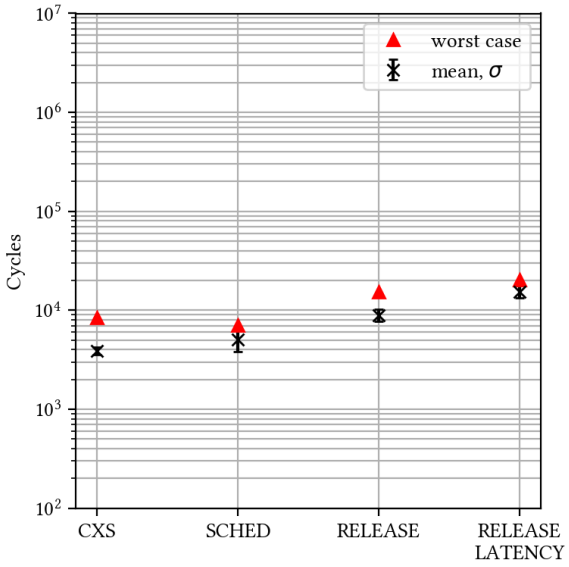

Figure 7.19: Our approach, 4-PEDF

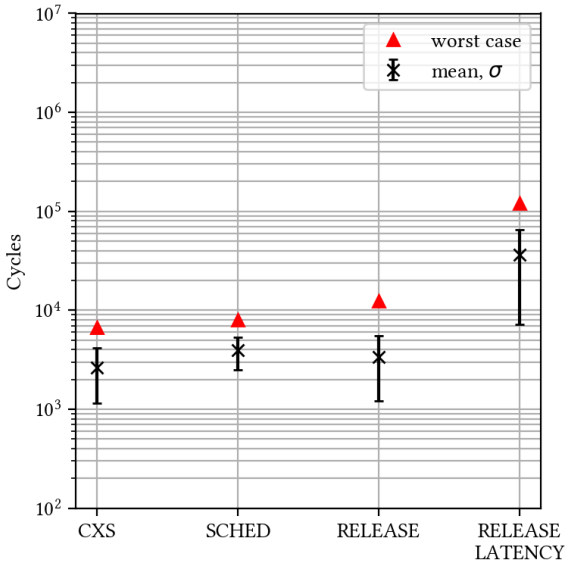

Figure 7.20: LITMUS^{RT}, 4-PEDF

that of LITMUS, because it incorporates some remote rescheduling overhead (as described earlier), and requires mode switches to make seL4 invocations.

The scheduling overhead under LITMUS is quite comparable to our approach. Under RASP, the average-case scheduling overhead is a fair amount smaller – this is likely because they perform their POSIX timer reprogramming in the release handler where possible (which may explain why the release overhead is so high under RASP). Under our approach and in LITMUS, the release overhead is dominated by the scheduling and context switch costs.

Under RASP, the release latency and remote rescheduling overhead is much higher than that experienced by LITMUS or our approach. This is because both of these operations require multiple Linux system calls under RASP. For example, sending a preemption signal under RASP involves repeatedly invoking `pthread_kill` for remote processors.

Schedulability Analysis

Using these measured overheads, we can perform schedulability simulations. In each of these tests, randomly-generated 16-task systems are used, with uniformly distributed costs & periods. Each data-point in the schedulability test represents the percentage of all task sets with CPU utilisation sum (x-axis), out of 500 random task sets, which were schedulable (y-axis). In each of these plots, a 'good' result is represented by a skew to the right, as this means more task sets were schedulable. Strictly, in terms of real-time schedulability, worst-case overheads should be

the inputs to our schedulability analysis. We, however, decided to also run simulations based on mean overheads – this gives an unfair advantage to the monolithic systems, but may be relevant in a soft real-time scenario.

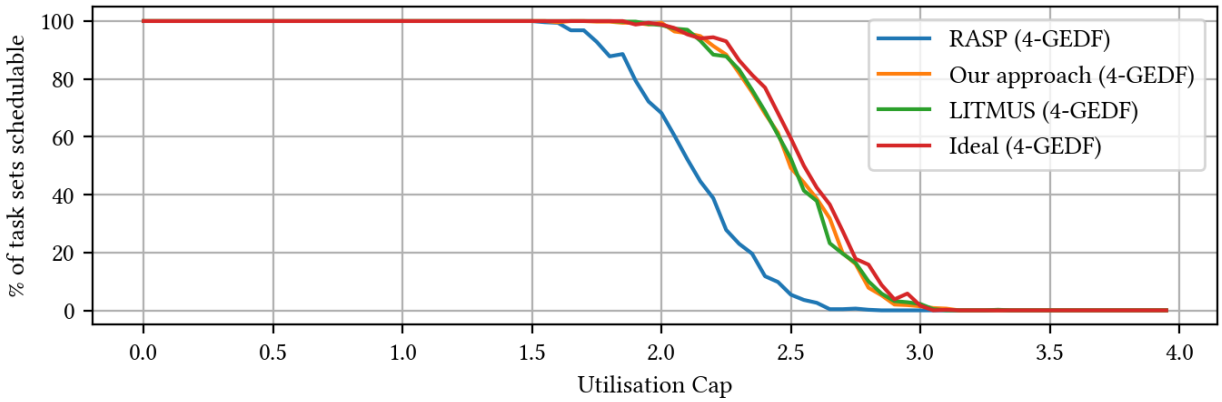

Figure 7.21: (mean) 4-GEDF schedulability, 1mS to 10mS task period distribution

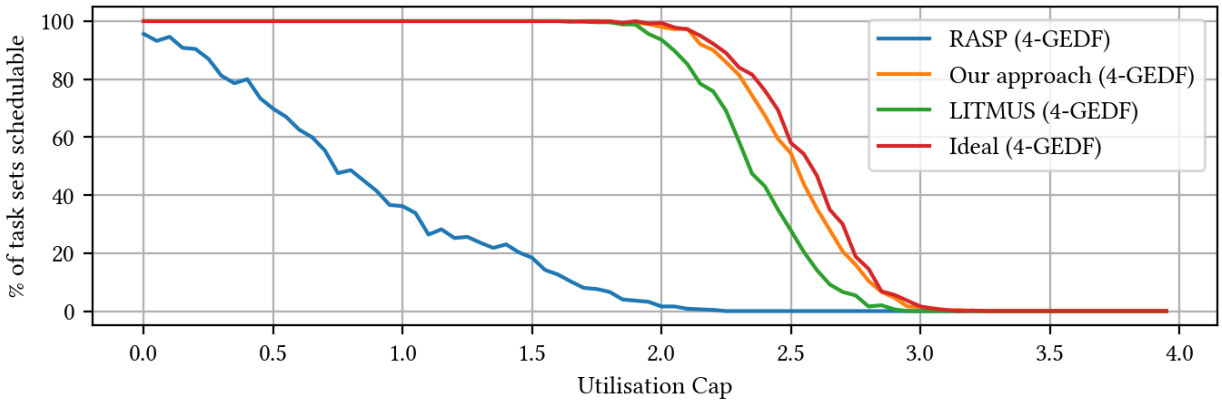

Figure 7.22: (worst) 4-GEDF schedulability, 1mS to 10mS task period distribution

First, we consider 4-GEDF schedulability with long task periods in Figure 7.21 and Figure 7.22. In both cases, LITMUS & our approach are very near the ideal utilisation limit – our approach posessing a slightly higher utilisation cap when worst-case overheads are considered. The schedulability of RASP suffers in the worst-case, due to its repeated system calls into Linux. We omit RASP from the next series of short-period tests, as its schedulability cap is obviously lower than for LITMUS or our approach.

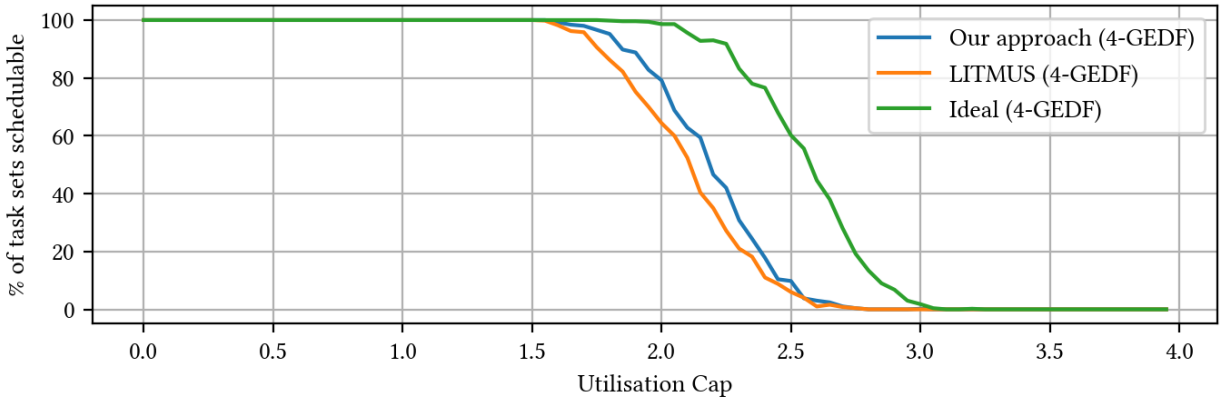

Figure 7.23: (mean) 4-GEDF schedulability, 100μS to 1mS task period distribution

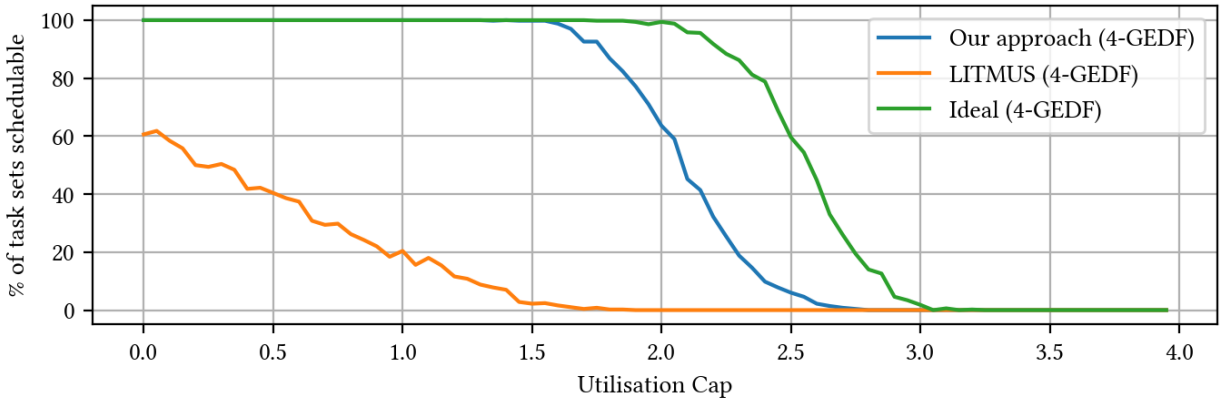

Figure 7.24: (worst) 4-GEDF schedulability, 100μS to 1mS task period distribution

By shortening the task periods under 4-GEDF as shown in Figure 7.23 and Figure 7.24, the difference in utilisation cap between LITMUS and our approach becomes more dramatic. In the mean overhead case, we obtain a *slightly* higher utilisation cap than LITMUS, but as soon as we use the worst-case overheads, the higher variation in LITMUS code-path overheads means that its schedulability suffers. Remote rescheduling overhead is a significant factor in this test – which is much smaller under our approach. Note that 100μS is quite a short task period – perhaps unusually short for an actual real-time system, we include it as it makes capacity limits more obvious.

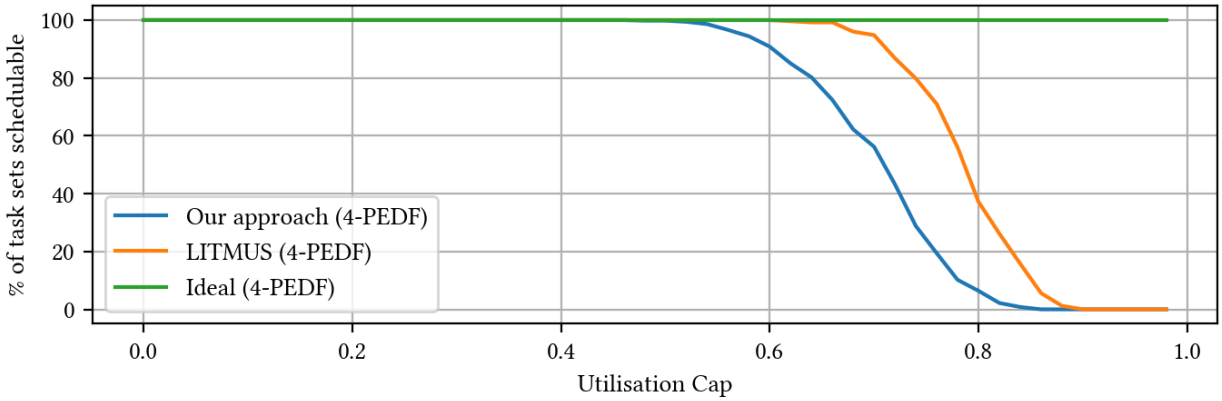

Figure 7.25: (mean) 4-PEDF schedulability, 100μS to 1mS task period distribution

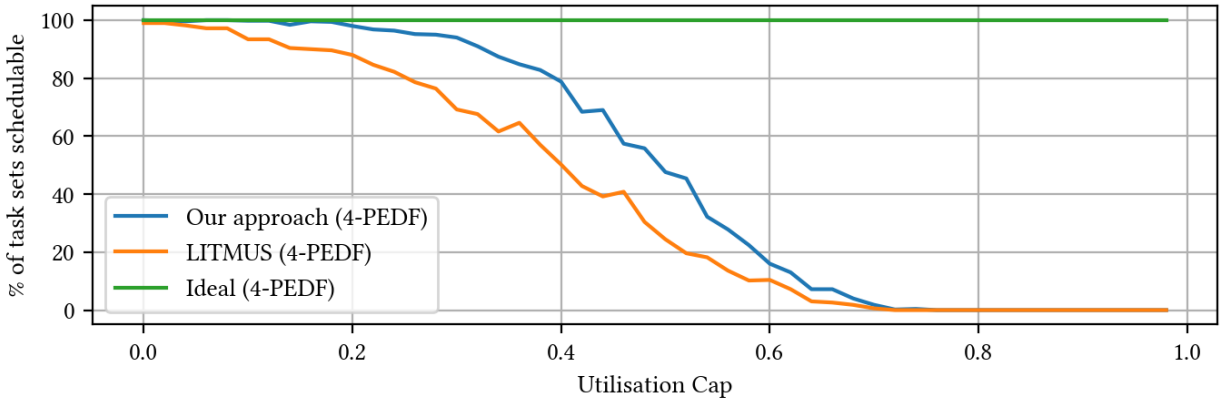

Figure 7.26: (worst) 4-PEDF schedulability, 100μS to 1mS task period distribution

For 4-PEDF, we omit the 1mS-10mS tests as there was no significant schedulability difference between our approach & LITMUS (they are located in Appendix D). The short-period cases are shown in Figure 7.25 and Figure 7.26. In the mean-overhead case, LITMUS has a higher schedulability than our approach, likely because it has slightly lower average context switching and release overheads. In the worst-overhead case however, our approach has higher schedulability due to the lower variance in overheads.

To summarise, the C-EDF scheduler we have constructed on seL4 is competitive with that provided by LITMUS, particularly when worst-case overheads are taken into account. In the 1mS-10mS case, both schedulers achieve almost the same schedulability cap, with RASP falling slightly behind.

7.4 MC-IPC Evaluation

In this section, we describe how our MC-IPC implementation was evaluated. Whilst we did construct a functioning MC-IPC prototype as described in section 6.3, our evaluation of the prototype is unfortunately rather limited – both due to time constraints, and for the reasons we described previously in section 5.4:

As a result, we limit our MC-IPC evaluation to a `schedplot` trace demonstrating properties we outlined in section 6.3, a simple invocation latency test, an observation of the implementation complexity, and an outline of why we believe model changes should be made before making any judgement as to how competitive an MC-IPC implementation might be on a microkernel.

7.4.1 `schedplot` trace

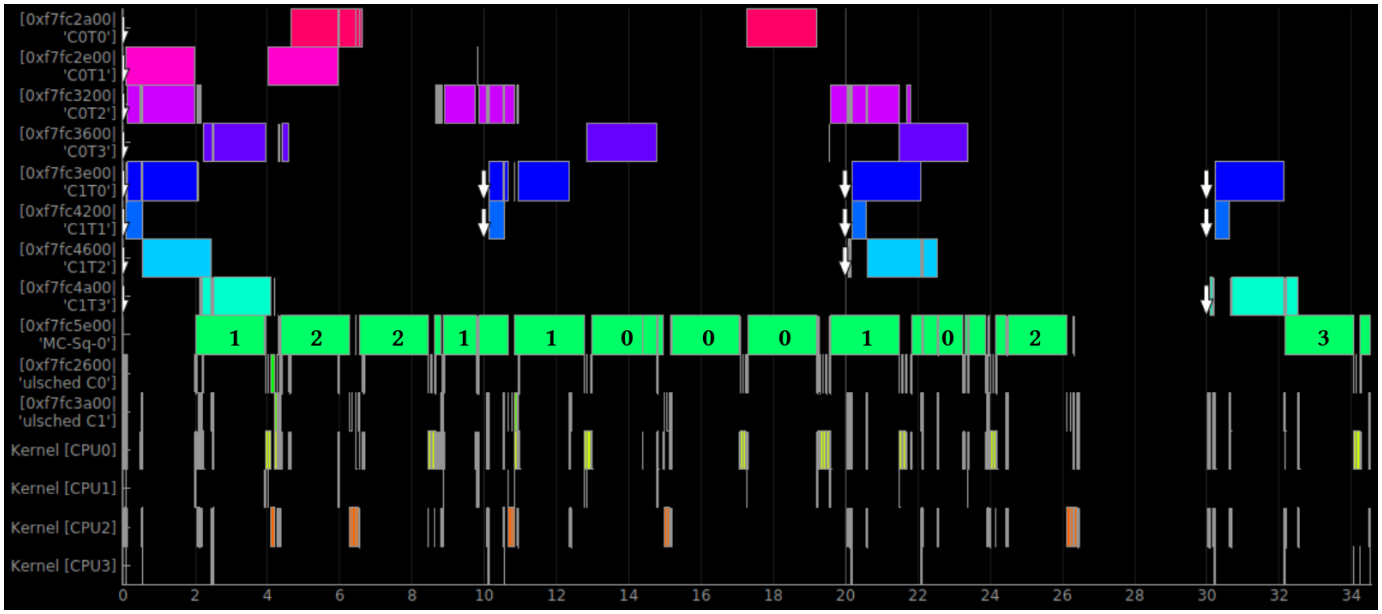

Figure 7.27: 2x2-CEDF MC-IPC trace

Whilst developing our prototype, we used our plotting tool to verify that our MC-IPC implementation operated as expected. An example of such a trace is shown in Figure 7.27. In this trace, the x-axis is in milliseconds, and we are running a system with two 2-CPU G-EDF clusters across 4 cores (i.e C-EDF with cluster size 2). We choose 2x2 C-EDF here as it is the most complex combination of MC-IPC and our scheduler which runs on 4 cores, exercising most of the components of our implementation. Cluster 0 (which contains all tasks C0TX),

has comparatively long EDF deadlines, whereas cluster 1 (which contains tasks C1TX) has comparatively short EDF deadlines. Each task contains a loop whereby it spins for a certain period and then makes an MC-IPC request. The MC-IPC server, on a request, performs 2ms of arithmetic operations before replying. We have overlayed the CPU that the MC-IPC resource server ('MC-Sq-0') is situated on at every point in time. In this example, the MC-IPC server migrates between cores & clusters as required. Because in this example we have purposefully created two clusters with vastly different average deadlines, this demonstrates that even though the first cluster has tasks of comparatively low EDF priority, that cluster is still able to have its requests served. Note that this trace was taken in debug mode (so we can see task names), meaning the execution time of the user-level schedulers and kernel invocations as shown are not representative of their true behaviour – they are barely visible in release mode.

7.4.2 Invocation Latency

To test the invocation latency experienced by a high-criticality task, we constructed another 2x2 C-EDF system which models EDF-VD criticality in LO mode. On the first cluster, we placed a single high criticality task (with artificially shortened deadline in accordance with EDF-VD) as well as 4 denial-of-service tasks. The denial-of-service tasks on the same cluster as the high-criticality task had much longer deadlines than the high-criticality task (thus representing lower criticality tasks under EDF-VD). On the second cluster, we placed 5 'malfunctioning' high-criticality tasks, which have the same or smaller artificially shortened deadline as the high-criticality task on the first cluster, but denial-of-service the resource server. All the denial-of-service tasks constantly make MC-IPC requests in an infinite loop. The server itself performs arithmetic operations for 1ms before returning.

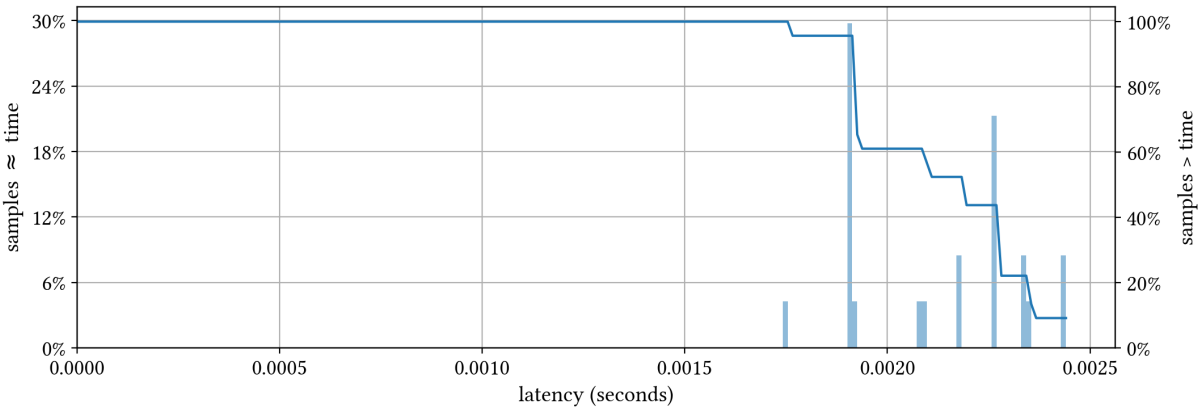

Figure 7.28: x86 MC-IPC invocation latency

A probability histogram of the invocation latency experienced by the high-criticality task on the first cluster is shown in Figure 7.28 – over a period of 10 seconds. Latency is measured using overhead-accounted scheduling dumps, between when the task makes a request, and when a response is received from the MC-IPC server. As is visible, all requests take less than 2.5 milliseconds to complete, which is less than the MC-IPC upper bound (of 9ms, given our server execution time and cluster arrangement). In future work, it would be interesting to implement priority & FIFO-ordered IPC to perform a comparison – unfortunately we were not able to do this due to time constraints.

7.4.3 Implementation Complexity

Operation	Lines	Comments/Blank	SLoC
Inserted	1421	263	1158
Deleted	380	25	355

Table 7.4: Code required to implement MC-IPC (excluding C-EDF scheduler & testbench)

A numerical summary of the 'Source Lines of Code' (SLoC) required to add MC-IPC support and idling reservations to our C-EDF system is provided in Table 7.4. Recall that one of the key reasons we wanted to investigate building resource protocols at user-level is because they have the potential of being easier to prototype. Our 1158-line implementation (which implements

MC-IPC and idling reservations), is much smaller than the 3700-lines Brandenburg [2014] required to add MC-IPC to LITMUS (who also augment their scheduler with MC-IPC and reservations). One could argue that this is an unfair comparison given the amount of boilerplate required to add any functionality to the linux scheduler, and that LITMUS' implementation has more features. However, an almost factor-of-3 difference demonstrates that it is possible to build protocols like MC-IPC with less engineering effort on top of a microkernel than is required in a monolithic kernel.

Still, we believe that modifications to the seL4 API are required to fairly demonstrate the performance of a microkernel in this scenario. Toward the end of our implementation, we discovered some small changes which could be made to the seL4 API which would simplify our implementation and improve MC-IPC performance. This, and more avenues for future work, will be covered in chapter 8.

7.5 Summary

In this section, we investigated the properties of our tracing infrastructure, evaluated the performance of our C-EDF scheduler, and performed functional tests on a preliminary MC-IPC implementation. Crucially, we demonstrated that it is possible to build a C-EDF scheduler, and mixed-criticality resource sharing protocol, as user-level policy on a microkernel. We found that our new system call for time management made a large performance difference to user-level scheduling performance, and investigated the scalability properties of our scheduler. We discovered that our C-EDF scheduler is performance-competitive with a linux-based monolithic approach, and that the implementation effort required to implement our mixed-criticality resource sharing protocol is low, when compared to an existing monolithic implementation. In the next chapter, we will examine some possibilities for future work – particularly with respect to our MC-IPC implementation.

8 | Future Work

8.1 Changes to seL4 API for MC-IPC

Toward the end of of implementing our MC-IPC prototype, we discovered that some minimal modifications to the model of the seL4 microkernel would have allowed for a cleaner MC-IPC implementation. These changes we propose are:

- **Idling Scheduling Contexts:** To prevent one having to implement idling reservations purely as user-level policy, we propose adding an option to scheduling contexts such that threads may consume budget whilst queued on an endpoint. This would involve an extra parameter to `seL4_SchedControl_Configure`, and changes to how the sporadic server implementation in the kernel interacts with endpoints.

- **Augmented Timeout Exceptions:** A user-level scheduler needs to know whether a timeout fault emerged from a thread blocked on an endpoint (or not), to decide whether to respond with a failure message to the task (or not). Additionally, if a timeout exception emerges from a task blocked on an endpoint, the request must be synchronously pruned under MC-IPC. We propose an additional message word be returned from timeout exceptions indicating a boolean as to whether the fault came from a task blocked on an endpoint.

- **Augmented `seL4_YieldTo()`:** A user-level scheduler needs to know if a thread is blocked on an endpoint which has a different thread (on another core) donating its scheduling context to a server. This is so that the scheduler can schedule another task in place of the blocked thread. For a high-priority user-level scheduler using `seL4_YieldTo()`, there is no feedback to the scheduler as to whether the thread associated with the scheduling context yielded to is actually runnable. We propose

`seL4_YieldTo()` return a boolean indicating whether the yield is to a blocked thread or not.

An implementation of MC-IPC on top of a kernel with these modifications might take the following form:

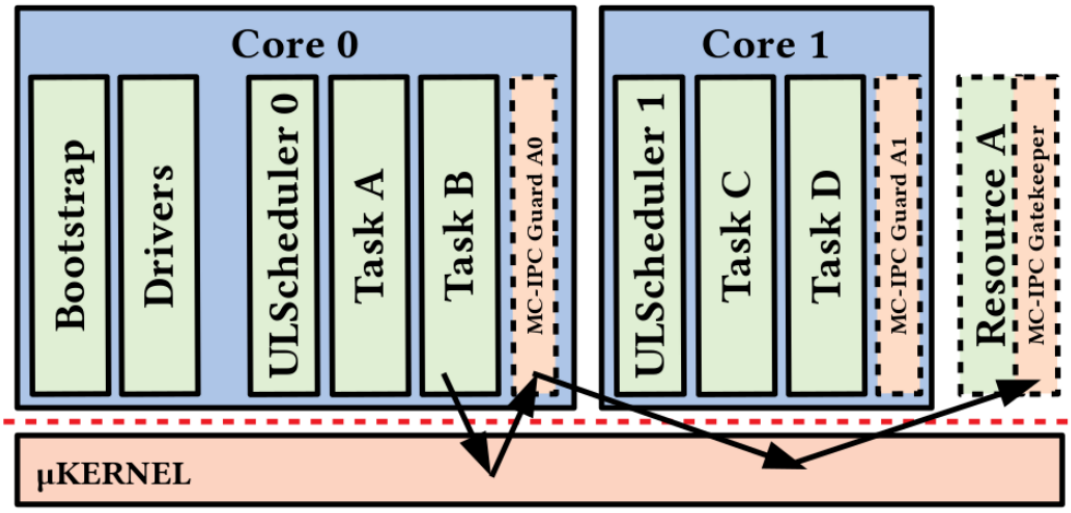

Figure 8.1: MC-IPC design

Assume (for simplicity) that we have only 1 resource server. On each cluster, we place a user-level scheduler, a set of tasks, and *passive* MC-IPC guard server (whose sole purpose is to proxy requests between tasks and the *actual* resource server). Additionally, we have the resource server itself, also a passive thread. When a task makes a request, it performs an `seL4_Call` on its badged endpoint capability to its cluster-local MC-IPC guard server, donating its scheduling context in the process. The guard server would have to cache & prioritize cluster-local requests according to the MC-IPC protocol - perhaps through a shared-memory datastructure with the user-level scheduler that indicates instantaneous task priorities, and shared datastructures with other MC-IPC guards to keep track of global queue length. The guard server would then set itself as the timeout fault handler for the resource server, and forward the correct request onto the resource server. Note that this is a similar arrangement to the bandwidth inheritance model proposed by Lyons et al. [2018] - the key difference is our API changes as described above.

One might think that it would make sense to construct the system such that all guard servers are the same priority and simply use synchronous IPC between the guard servers and the

resource server to directly model MC-IPC's global FIFO queue. Unfortunately, if a scheduling context is currently donating time to the resource server and expires, it is not possible to easily configure the resource server's timeout fault handler to be the correct (cluster-local) MC-IPC guard - we believe it would be simpler to keep this queueing structure as responsibility of the guard servers. A solution which would maintain the 'best of both worlds' in this specific scenario would be to add an option such that ignored timeout faults automatically return to their original task and permit donation from the next blocked thread on an endpoint. This, however, would add unnecessary policy to the microkernel - it essentially implements the well-studied 'helping' protocol, which is already possible to model in seL4.

Another interesting consideration is how user-level schedulers should handle `seL4_YieldTo()` calls to blocked tasks. Assume at first that a user-level scheduler yields to a lower-priority, but runnable task (since the highest priority task is blocked on a request, as determined by a yield call). When a high-priority task becomes unblocked (i.e able to donate time to a resource server), we want that highest priority task to be scheduled. A simple solution to this would be for the MC-IPC gate to check for pending requests on a reply or timeout fault for the current-served client, and send an IPC to the correct user-level scheduler (similar to the IDLE message we implement in our prototype).

8.2 Extended Evaluation

The main goal of this thesis was to investigate the properties of a user-level MCS scheduler. It would be interesting to extend our work beyond the platforms we evaluate here, i.e to different CPU architectures, or to investigate scalability to core counts above 4. Another obvious next step is to apply what we have learned here and construct an actual real-time system (e.g quadcopter or autonomous vehicle) which utilises our work - and evaluate its properties.

8.3 Scheduler WCET Accounting & Admission Control

Since seL4-MCS will eventually have a sound worst-case execution time analysis (WCET), it might be interesting to see how a similar analysis could be performed on the user-level scheduler itself. An analysis tool might be able to use the existing seL4 WCET properties in tandem with an analysis of the user-level scheduler implementation, to estimate the scheduler

WCET. This WCET could then be used to augment an admission control policy enforced by the user-level scheduler when admitting hard-real-time tasks.

8.4 Prototyping framework in a high-level language

```
rootThread :: String -> ThreadContext
rootThread id = do
    atomic $ puts ("Init thread: " ++ id)
    -- Spawn some threads (bypass TCB cap manipulation)
    syscall $ SysSpawnThread 255 (loopyThread "[child1]") Nothing
    syscall $ SysSpawnThread 255 (loopyThread "[child2]") Nothing
    whileM (return True) (do
        -- Spin, child threads should get CPU time
        atomic $ puts ("In thread: " ++ id)
        )
    atomic $ puts ("End of thread: " ++ id)
return ("Return: " ++ id)
```

Listing 8.1: This 'virtual thread' was run against the actual seL4 scheduler from the Haskell spec with some minor modifications (to remove dependencies to the rest of the kernel).

In an investigative effort, during the early stages of this thesis we attempted to prototype user-level schedulers against the in-progress real-time seL4 Haskell specification (essentially an implementation of seL4 but in a higher-level language); to test whether this might be a viable prototyping (and correctness verification) approach. Completing this work on verifying user-level scheduling behaviour in a high-level language was deemed too much work for this thesis, however it would make interesting future work. Before moving onto different approaches, we created a rudimentary functioning self-contained Haskell model of the (non-RT) seL4 kernel scheduler and ran it against virtual threads. An example of the implementation of such a thread can be seen in Listing 8.1. We believe that a user-level scheduling prototyping framework would be very useful in future work, allowing the implementor to create high-level 'litmus tests' of their design and test it against the true seL4-MCS semantics without having to create an entire functioning prototype. In our case, we would have discovered the previously described seL4 model changes for MC-IPC much sooner if such a framework existed.

9 | Conclusion

As described in earlier sections, a user-level scheduling architecture provides many potential advantages when compared to in-kernel, monolithic approaches. We began by exploring existing monolithic architectures with mixed-criticality support, existing user-level approaches with real-time support, a mixed-criticality resource sharing protocol, and other kernels of interest. Unfortunately, modern mixed-criticality schedulers (in particular multicore-applicable flexibly partitioned architectures) remain untested in a user-level context.

In this thesis, we sought to rectify this – by implementing a flexible multicore scheduler & resource management protocol as user-level components on the seL4 microkernel. To achieve this, we first constructed specialized tracing infrastructure and evaluated its effectiveness. We then demonstrated that not only is it possible to build a C-EDF scheduler as user-level policy on a microkernel, but that such a scheduler can be performance-competitive with existing monolithic approaches. We found that a small modification to the kernel API vastly improves user-level scheduling performance, and that the implementation effort required to implement our mixed-criticality resource sharing protocol is low, when compared to an existing monolithic implementation. Finally, we proposed a new model for implementing MC-IPC for future work, which would facilitate the construction of a more efficient implementation, and fair comparison with an optimized monolithic implementation in the future.

There is enormous scope for future work in this space – evaluation across more schedulers & architectures, constructing a high-level modelling framework, further experimentation with kernel model changes – each of these enough for another thesis taken alone. Our hope is that research continues in this space, and that we may one day live in a future of truly trustworthy systems.

Bibliography

Aravindh Anantaraman, A Mahmoud, Ravi Venkatesan, Yifan Zhu, and Frank Mueller. EDF-DVS scheduling on the IBM Embedded PowerPC 405LP. In *Proceedings of the IBM P= ac2 Conference*, 2004.

Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

Sanjoy K. Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne van der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *Algorithms – ESA 2011*, pages 555–566, 2011.

Aaron Block, Hennadiy Leontyev, Bjorn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 47–56. IEEE, 2007.

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

Björn B. Brandenburg. Binomial heaps. `https://github.com/brandenburg/binomial-heaps`, 2012.

Björn B. Brandenburg. A Synchronous IPC Protocol for Predictable Access to Shared Resources in Mixed-Criticality Systems. In *IEEE Real-Time Systems Symposium, 2004*, pages 196–206, Dec 2014. doi: 10.1109/RTSS.2014.37.

Alan Burns and Robert I. Davis. Mixed Criticality Systems - A Review. `https://www-users.cs.york.ac.uk/burns/review.pdf`, 2018.

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

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

Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. In *Journal of Parallel and Distributed Computing*, volume 16, pages 306–318. Academic Press, Inc, 12 1992.

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

Philippe Gerum. Xenomai-implementing a RTOS emulation framework on GNU. *Linux, White Paper, Xenomai*, 2004.

Hermann Hartig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*, pages 10–pp. IEEE.

Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems*, 34(1):1:1–1:29, April 2016.

H. M. Huang, C. Gill, and C. Lu. Implementation and evaluation of mixed-criticality scheduling approaches for periodic tasks. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 23–32, April 2012. doi: 10.1109/RTAS.2012.16.

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

Jochen Liedtke. On microkernel construction. In *ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

Anna Lyons, Kent Mcleod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM.

A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, 1983.

Malcolm S. Mollison and James H. Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 283–292. IEEE Computer Society, April 2013.

S. Oikawa and H. Tokuda. User-level real-time threads. In *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 7–11, May 1994. doi: 10.1109/RTOSS.1994.292570.

R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 741–746, March 2010.

Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 3. ACM, 2015.

S. Ruocco. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 246–256, Dec 2006. doi: 10.1109/RTSS.2006.50.

John Rushby. Design and verification of secure systems. In *ACM Symposium on Operating Systems Principles*, pages 12–21, Pacific Grove, CA, USA, December 1981.

Lui Sha. Solutions for some practical problems in prioritized preemptive scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium, 1986*. IEEE Computer Society Press, 1986.

Brinkley Sprunt, Lui Sha, and John K. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.

Mark J. Stanovic, Theodore P. Baker, An-I Wang, and Michael González Harbour. Defects of the POSIX sporadic server and how to correct them. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 35–45, Stockholm, 2010. IEEE Computer Society.

Jan Stoess. Towards effective user-controlled scheduling for microkernel-based systems. *ACM SIGOPS Operating Systems Review*, 41(4):59–68, 2007.

Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *IEEE Real-Time Systems Symposium*, pages 239–243, 2007.

Marian Vittek, Peter Borovansky, and Pierre-Etienne Moreau. A Simple Generic Library for C. In *Proceedings of 9th International Conference on Software Reuse, Tur in, Italy*, pages 423–426. Springer, 2006.

Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, pages 299–308, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4739-8.

Appendix A: Log-Buffer Mappings

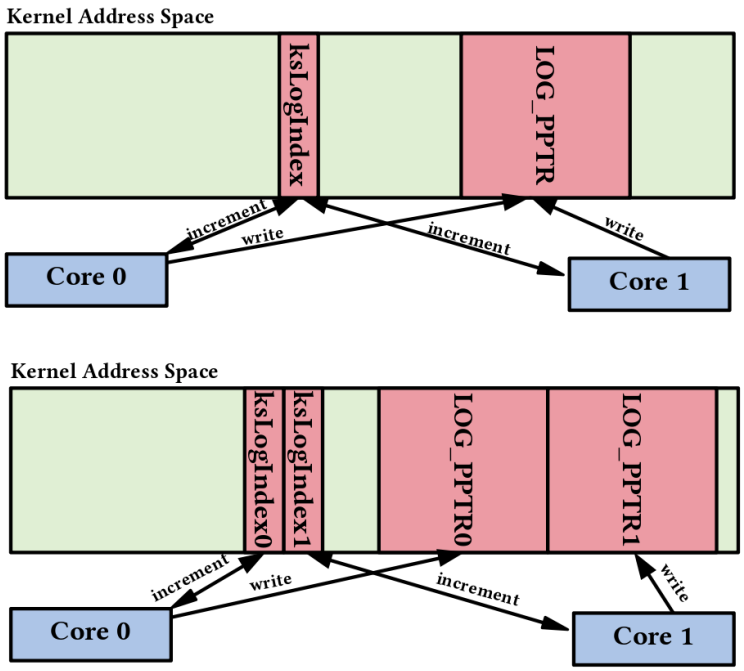

Figure 1: Types of log-buffer mappings tested (existing above, prototype below).

As part of this thesis, we investigated whether it would provide a performance improvement to modify the logging infrastructure to instead have a separate logging page mapped in for each CPU core, as well as move the shared position variable to be core-local (to reduce cache contention). An illustration of the difference between the existing & prototyped approaches is provided in Figure 1. Note that in these figures, the address spaces are not to scale, and both `ksLogIndexX` variables are assumed to sit in different cache lines. In early stages of development, it was discovered that this core-local approach was actually *slower* than the original for our purposes (which are not especially cache-heavy). To evaluate the approach, we constructed a benchmark on ARM whereby userspace would make system calls on all cores simultaneously at random intervals. In investigating the slowdown, we disassembled our log

writing function and found that its increase in effective length by 13 cycles was the same as the performance penalty. Hence, the slowdown under our prototype was found to be entirely accounted for by the increase in instruction count in the trace function. The majority of this overhead was simply the logic required to determine which core we are currently executing on (by interpreting the current stack pointer), such that we write to the correct log buffer. This dominated over cache conflict effects in our tests, which should be relatively small anyway as we use a write-through mapping for the buffer (but not the index variable). In future work it would be interesting to explore this further in tests under a cache-heavy userspace – but for our purposes, it was lower-overhead to stick with the current implementation.

Appendix B: C-EDF Measurement Details

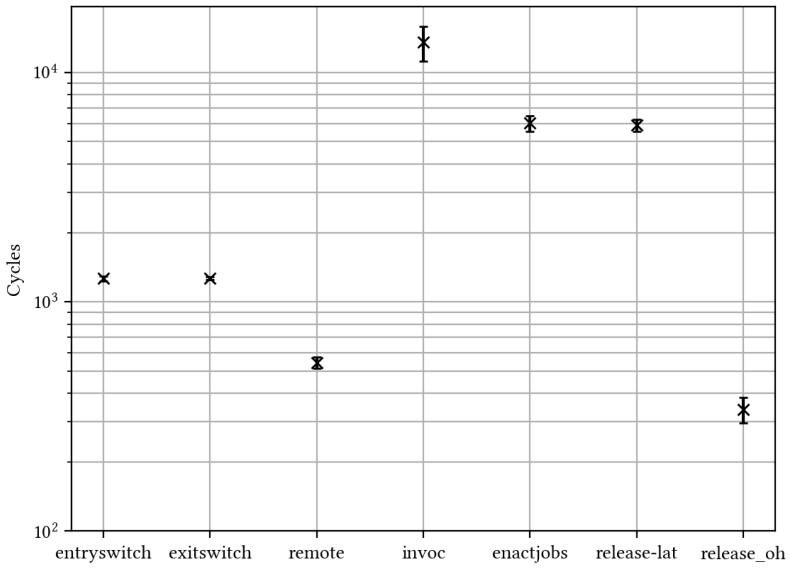

Figure 2: Example of 4-GEDF overhead measurement components (Sabre)

Although we have described what the overheads which we measured *are*, we did not describe in detail how they were actually measured. Using G-EDF as an example, we actually make 7 separate measurements, all using tracepoints placed carefully throughout the scheduler and userspace. These 7 measurements are then processed by a script to form our final 5 overhead measurements. To make each measurement, we re-compile the scheduler such that *only* the required measurement tracepoints are included, and then subtract the tracepoint overhead after an experiment – so that we do not include the cost of 'tracepoints-within-tracepoints'. These 7 measurements are shown in Figure 2, and are described below:

- `entryswitch`: the time between when a task is preempted, and just after the corresponding user-level scheduler `Recv`. We measure this by constantly measuring cycles

elapsed in an infinite loop in one of the userspace tasks. If the time elapsed is suddenly higher than usual, the task detects it was preempted and logs the time just before the preemption. The user-level scheduler has a corresponding tracepoint after the `Recv` – subtracting the two (via post-processing in our case), provides the entry-switch cost.

- `exitswitch:` the time between just before a user-level scheduler `Recv`, and a user-level task starts to execute. This is measured using the same method as `entryswitch`.
- `remote:` the time between when a migration invocation completes, and when that task is actually scheduled on a remote CPU. Again, we measure this in a similar way to `entryswitch`, but use a tracepoint in the migration code of the scheduler rather than in the main loop.
- `invoc:` the total scheduler invocation time, without the entry and exit switch. This is simply 2 ordinary tracepoints within the scheduler.
- `enactjobs:` the length of execution of our context-switch function.
- `release-lat:` time delta between when a release interrupt is set (before control returns to userspace), and the current time when control returns to the user-level scheduler..
- `release-oh:` overhead incurred when moving a thread from the release queues to the ready queues.

Once these measurements are made, our postprocessing scripts collate them into overall overhead statistics. For example, the 'SCHED' overhead is formed by adding `entryswitch`, `exitswitch` and `invoc`, and then subtracting `enactjobs`.

Appendix C: ARM C-EDF Measurements

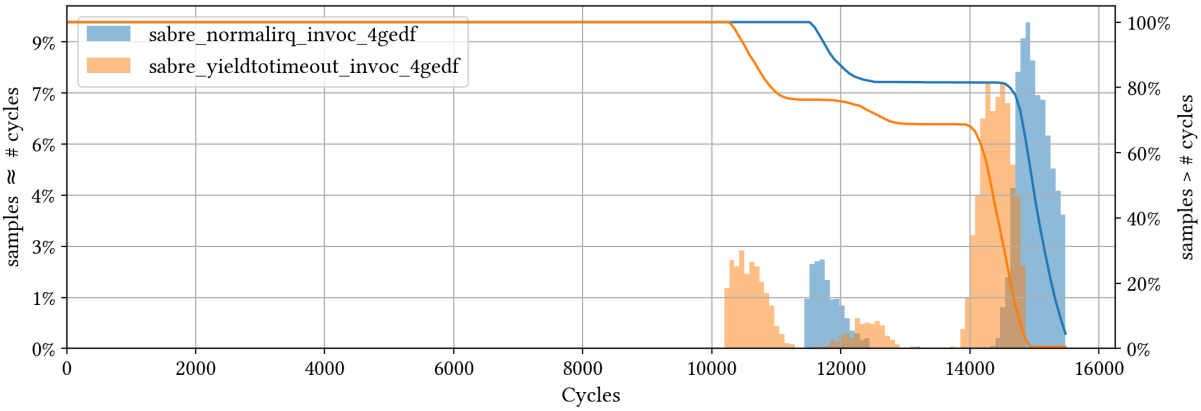

Figure 3: ARM 4-GEDF invocation times with different time sources

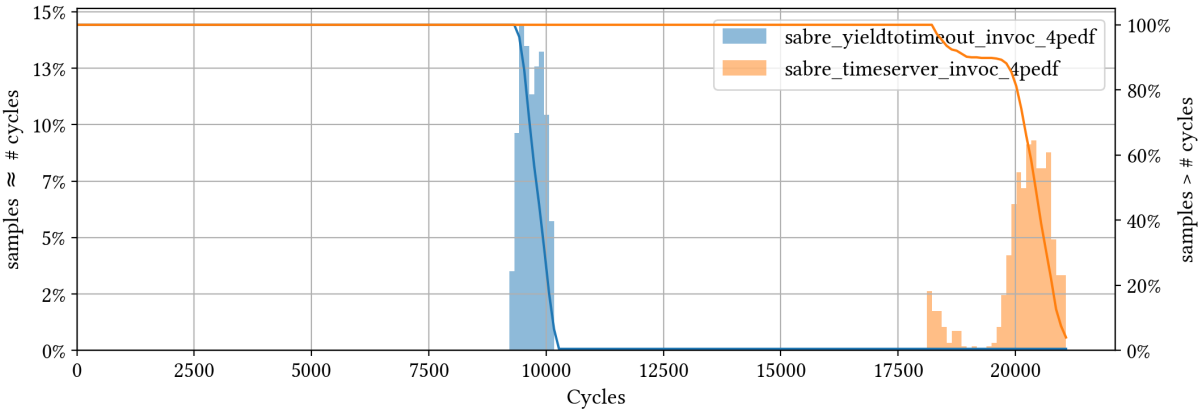

Figure 4: ARM 4-PEDF invocation times with different time sources

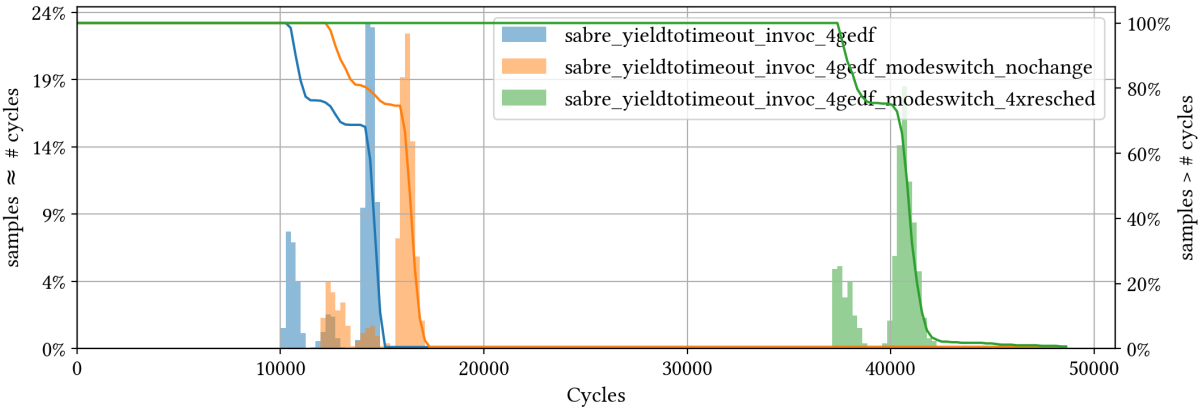

Figure 5: ARM 4-GEDF invocation times with criticality mode switch

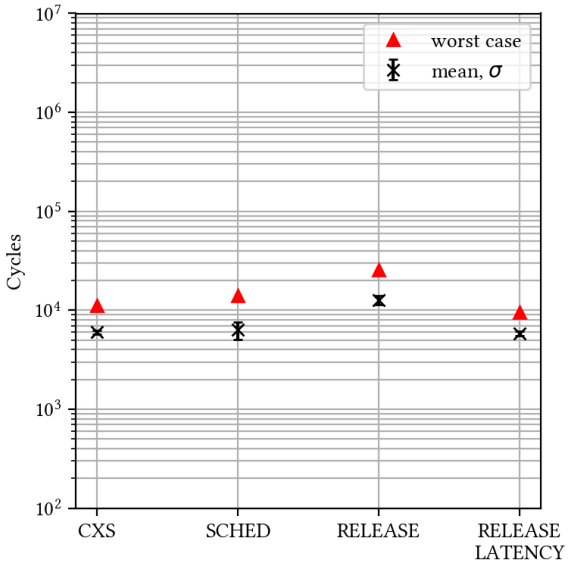

Figure 6: ARM 4-PEDF overheads

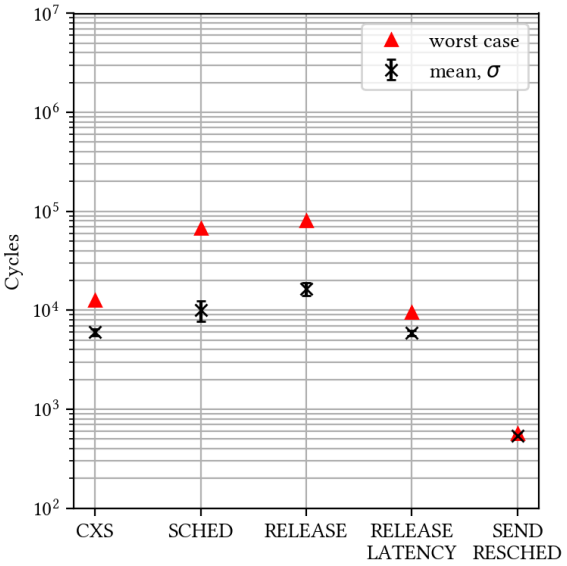

Figure 7: ARM 4-GEDF overheads

Appendix D: x86 4-PEDF null-result tests

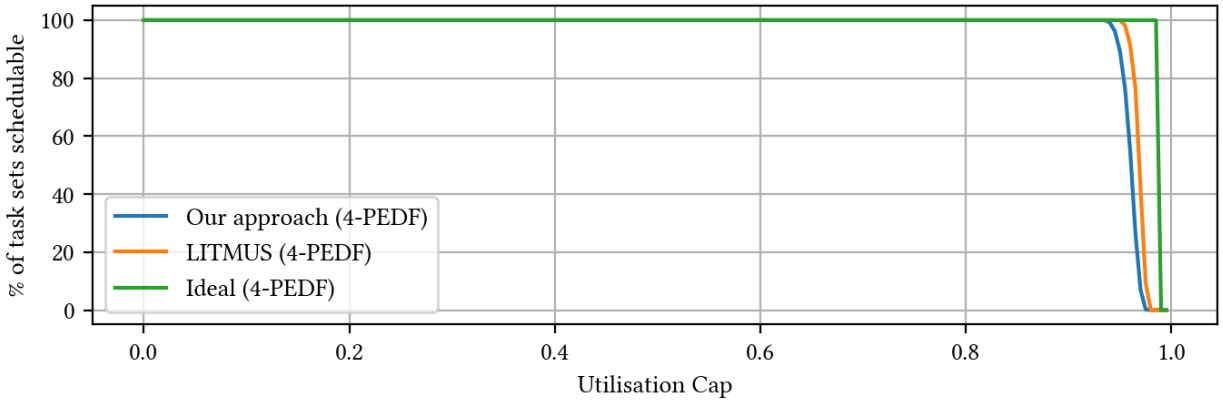

Figure 8: (mean) 4-PEDF schedulability, 1ms to 10mS task period distribution

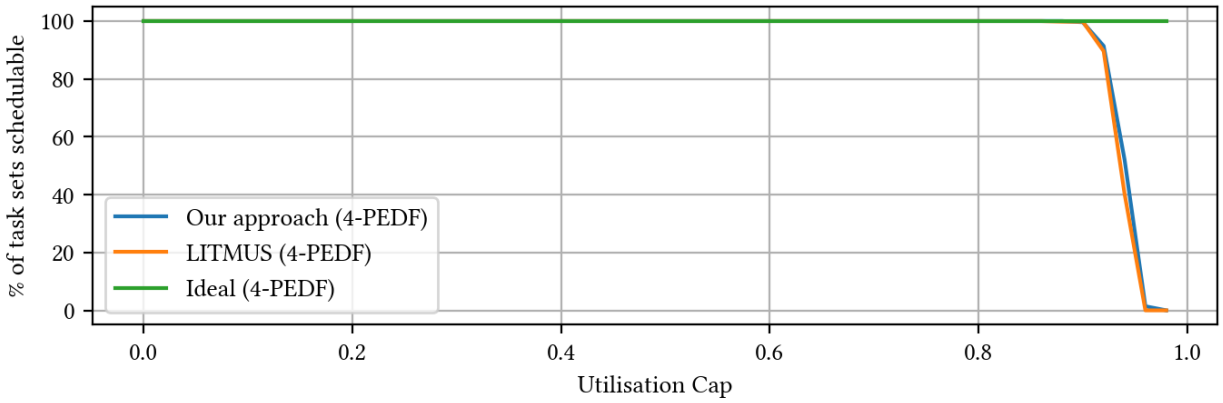

Figure 9: (worst) 4-PEDF schedulability, 1mS to 10mS task period distribution

Appendix E: Implementation Metrics

This appendix lists some metrics on our implementation changes, as reported by `git diff --stat`.

C-EDF Scheduler

```
 libsel4bench/include/sel4bench/kernel_logging.h  |   4
 libsel4utils/include/sel4utils/aligned_alloc.h   |  18
 libsel4utils/include/sel4utils/benchmark_track.h |  40
 libsel4utils/include/sel4utils/linksched.h       |  74
 libsel4utils/include/sel4utils/rt_job.h          |  44
 libsel4utils/include/sel4utils/trace_helpers.h   |  11
 libsel4utils/src/edf.c                           | 770
 libsel4utils/src/linksched.c                     | 312
 libsel4utils/src/rt_job.c                        |  45
 9 files changed, 1314 insertions(+), 4 deletions(-)
```

Listing 1: C-EDF Scheduler (core).

```
 CMakeLists.txt                           |      3
 apps/hardware/src/main.c                 |     84
 apps/sel4bench/src/benchmark.h           |      1
 apps/sel4bench/src/main.c                |     19
 apps/ulscheduler/CMakeLists.txt          |     70
 apps/ulscheduler/src/main.c              |    347
 libsel4benchsupport/include/hardware.h   |      6
 7 files changed, 523 insertions(+), 7 deletions(-)
```

Listing 2: C-EDF Scheduler (test bench).

MC-IPC

```
 libsel4utils/include/sel4utils/linksched.h     |    6
 libsel4utils/include/sel4utils/mc_ipc.h        |   61
 libsel4utils/include/sel4utils/mc_ipc_msg.h    |  125
 libsel4utils/include/sel4utils/rt_job.h        |   16
 libsel4utils/include/sel4utils/rt_request.h    |   62
 libsel4utils/include/sel4utils/sched.h         |    7
 libsel4utils/src/edf.c                         | 1057
 libsel4utils/src/linksched.c                   |   22
 libsel4utils/src/mc_ipc.c                      |  385
 libsel4utils/src/rt_request.c                  |   60
 10 files changed, 1421 insertions(+), 380 deletions(-)
```

Listing 3: MC-IPC (core, diff against C-EDF implementation).

```
 apps/ulscheduler/src/main.c | 272
 1 file changed, 257 insertions(+), 15 deletions(-)
```

Listing 4: MC-IPC (test bench, diff against C-EDF test bench).

`schedplot`

```
 rt_tasks.py     |  13
 schedplot.py    | 296
 sel4_types.py   | 154
 trace_events.py | 207
 4 files changed, 670 insertions(+)
```

Listing 5: `schedplot` implementation.

Kernel Changes

```
config.cmake                                          |  6
include/api/debug.h                                   |  6
include/arch/arm/arch/32/mode/types.h                 |  2
include/arch/arm/arch/benchmark.h                     | 13
include/arch/arm/arch/benchmark_overflowHandler.h     | 15
include/arch/arm/armv/armv7-a/armv/benchmark.h        |  4
include/benchmark/benchmark_track.h                   |  6
include/kernel/sporadic.h                             |  4
include/plat/imx6/plat/machine/timer.h                |  2
libsel4/include/sel4/benchmark_track_types.h          | 17
src/arch/arm/c_traps.c                                |  1
src/arch/arm/config.cmake                             |  2
src/benchmark/benchmark_track.c                       | 29
src/fastpath/fastpath.c                               |  7
src/model/statedata.c                                 |  3
src/object/schedcontrol.c                             |  2
16 files changed, 96 insertions(+), 23 deletions(-)
```

Listing 6: Log buffer implementation.

```
include/kernel/thread.h                                    | 25
include/object/structures.h                                |  2
libsel4/include/interfaces/sel4.xml                        | 13
libsel4/include/sel4/constants.h                           |  2
libsel4/sel4_arch_include/aarch32/sel4/sel4_arch/constants.h |  2
libsel4/sel4_arch_include/aarch32/sel4/sel4_arch/faults.h  |  4
libsel4/sel4_arch_include/aarch32/sel4/sel4_arch/types.bf  |  4
libsel4/sel4_arch_include/ia32/sel4/sel4_arch/constants.h  |  2
libsel4/sel4_arch_include/ia32/sel4/sel4_arch/faults.h     |  5
libsel4/sel4_arch_include/ia32/sel4/sel4_arch/types.bf     |  4
src/api/faults.c                                           |  5
src/kernel/thread.c                                        |  4
src/object/schedcontext.c                                  | 76
13 files changed, 142 insertions(+), 6 deletions(-)
```

Listing 7: `YieldToTimeout` implementation.

```
 CMakeLists.txt                                     |  2
 include/arch/arm/arch/32/mode/hardware.h           |  2
 include/arch/arm/arch/32/mode/model/statedata.h |  2
 include/benchmark/benchmark_track.h                |  9
 include/kernel/traps.h                             |  2
 include/kernel/vspace.h                            |  2
 include/model/statedata.h                          | 13
 libsel4/arch_include/arm/sel4/arch/syscalls.h   | 13
 libsel4/include/sel4/syscalls.h                    | 12
 src/api/syscall.c                                  | 33
 src/arch/arm/32/kernel/vspace.c                    | 38
 src/arch/arm/32/model/statedata.c                  |  2
 src/arch/arm/32/object/objecttype.c                |  3
 src/arch/arm/64/kernel/thread.c                    |  2
 src/benchmark/benchmark_track.c                    | 24
 src/model/statedata.c                              |  8
 16 files changed, 102 insertions(+), 65 deletions(-)
```

Listing 8: Multicore core-local log-buffer mappings (ARM).

Benchmarking/Processing scripts

```
  135 ./data/get.py
   60 ./cedf_good/context_outer/parse.py
   95 ./cedf_good/hist_mcipc.py
  104 ./cedf_good/components.py
  100 ./cedf_good/hist.py
  125 ./cedf_good/process_final.py
   77 ./cedf_good/compare.py
    8 ./cedf_good/make_trace_extrapolate.py
   42 ./data_sched/hist.py
   57 ./data_logbuf/get_nullsyscall_sabre.py
   57 ./data_logbuf/get_ia32_breakdown.py
   55 ./data_logbuf/get_nullsyscall_ia32.py
  915 total
```

Listing 9: Benchmarking/Processing scripts. (reported by `wc -l`)