Karlsruhe Institute of Technology

Improving Energy Efficiency with Hardware-assisted Scheduling on Heterogeneous CPUs

Bachelor's Thesis
submitted by

cand. inform. Nikita Thomas

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Dipl.-Inform. Thorsten Gröninger

27. November 2025 – 27. März 2026

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, March 27, 2026

Abstract

Energy efficiency has become a significant factor in the realm of computing, extending from battery life concerns in laptops and portable computers to data centers, where single watt differences can cost thousands of dollars in electricity.

A recent development has been the creation of heterogeneous processor architectures with high-performance and high energy-efficiency cores, e.g., Arm's big.LITTLE or Intel's P/E core design. To maximize the potential of these architectures, Intel has developed Thread Director, which assists operating system schedulers with hints on which core to place specific threads. In this work, we evaluate the hints given by Thread Director from an energy efficiency perspective by designing and implementing a userspace scheduler in the seL4 microkernel.

Microkernels, although often lacking in functionality compared to a monolithic kernel, make for a good candidate for measuring the impact of such new features, as these systems exhibit far less noise and interference during execution, allowing for more definitive conclusions to be drawn. Typically, they are easier for a developer to become familiar with and allow easy extension inside the kernel, allowing more rapid prototyping and testing of features once one is familiar with their features.

We evaluate our scheduler and Intel Thread Director on a desktop Arrow Lake CPU, finding that while we did not improve energy efficiency compared to a scheduler aware of heterogeneous processors, we found unusual behavior in the classification behavior of Thread Director, showing several orders of magnitude difference between class zero and other class types, which should be more thoroughly examined by future work. Furthermore, we evaluated different methods for our userspace scheduler to sleep without kernel involvement, achieving up to a 50% uplift in energy efficiency compared to busy-waiting.

Contents

Chapter 1

Introduction

Energy efficiency has become a significant factor in the realm of computing [1, 2], extending from battery life concerns in laptops and portable computers to data centers, where single watt differences can cost thousands of dollars in electricity.

Ever since the breakdown of Dennard Scaling around the mid-2000s [3], processor manufacturers have shifted focus from higher frequencies to core counts to increase performance. However, this trend can only continue so far until energy efficiency becomes an issue once again.

One attempt in this regard has been the development of heterogeneous processors, where processor cores are divided into a few high-performance (P) and many high energy-efficiency (E) cores, with some manufacturers like Intel and ARM further dividing some cores into a third category, low-power energy-efficiency cores (LPE). This has already been successfully done by ARM with their big.LITTLE architecture [4], Apple with their M series of processors [5], Intel since Alder Lake [6], and AMD with Zen 4 and Zen 4c, found in some of their 8000G series of processors [7]. Although AMD's approach differs in that their efficiency and performance cores show significantly fewer differences, compared to Intel, where two different core architectures are combined into a single chip [6]. To properly take advantage of the different core types, modern operating systems have to adapt their schedulers to differentiate between them and schedule according to the respective core's strengths. As this task is not trivial and often requires complex knowledge about a task, it is commonly not fully utilized.

In order to assist developers of operating systems, Intel also introduced Thread Director, an extension to the existing Hardware Feedback Interface (HFI), which provides the OS scheduler with hints from the hardware layer to help with scheduling decisions [8]. Thread Director is exclusive to x86_64 and Intel, with AMD implementing a similar version for its processors [9]. Additional support is, at the time of writing, only fully implemented in Windows 11 [10], with support for Linux still pending [11].

In this work, we investigate whether the hints Thread Director provides deliver a meaningful uplift in energy efficiency and/or performance. In order to do that, we designed and implemented a userspace scheduler taking advantage of Thread Director and some of the *User wait instructions*, namely `UMONITOR` and `UMWAIT` to reduce the overhead of kernel entries and exits [1].

Userspace scheduling allows the user or developer a high degree of flexibility, which can be used to allow for higher energy efficiency, performance, or lower latency of applications. Although this has traditionally been associated with an unacceptable loss in performance [13], we show that the gains achievable by bypassing the kernel as much as possible. With the currently available userspace instructions such as `UMONITOR/UMWAIT`, it is already possible to reduce the power consumption that comes from waiting in userspace by up to 50% compared to busy-waiting, see section 6.4.5, while avoiding the overhead of relying on the kernel for putting our thread to sleep. It could further benefit from userspace interrupts. Coupling these extensions with the hints provided by Thread Director, we show that losses can be lowered compared to a scheduler without Thread Director. We expect even more savings with upcoming CPU extensions, see section 7.2.3.

The kernel for which we develop this scheduler is the seL4 microkernel, as microkernels allow easy implementation of features normally found in the kernel in userspace. We chose seL4 as the development platform, as it is a mature and, in theory, formally verified member of the L4 Microkernel family that still sees active development and use in the industry.

Furthermore, research in the area of microkernels has not been concerned with scheduling or traditional performance, the focus historically being on IPC performance [14–17] and more recently on formal verification [18–20]. Aside from introducing mixed-criticality into the seL4 scheduler [21], the scheduler in L4-based microkernels has historically been a simple fixed-priority Round Robin scheduler [22], where taking advantage of multiple cores has to be done manually by the running application [23].

We see an opportunity for significant gains in this area, even if that would lead to the loss of fixed priorities, core assignments, or the formal verification[2] of the kernel. However, we try to minimize the changes done to the kernel.

In the following chapters, we introduce concepts essential to our thesis in chapter 2, like Intel Thread Director, microkernels, and capabilities. Then continue with a comparison to previous works in the areas of Thread Director, userspace scheduling, and scheduling in L4 microkernels.

[1] Do note that on some processors, `UMONITOR` and `UMWAIT` are faulty and may cause performance issues, see [12]

[2] The x86 configuration has not yet been fully verified [24]

After this, we introduce our design, explain what needs to be provided from both the kernel and userspace, rounding off with the compromises we had to make.

With our design set, we detail the implementation process, first the prerequisites, then work done on the kernel, following up with the concrete scheduler we implemented in userspace and detailing what we failed to do, e.g., due to time constrains or scope of our thesis, and what could be done better in a future implementation, explaining the issues we encountered along the way.

Following this, we evaluate our prototype for energy efficiency, by constructing various scenarios we let our scheduler run through, rationalizing how we measured our results and what we measured, with a discussion of our findings at the end. In addition we also evaluated the overhead incurred by userspace scheduling, taking note of and discussing possible slowdowns caused by running in a virtualized environment in section 6.4.3, and the duration Thread Director takes to classify threads, finding large disparities between class zero reporting compared to the other classes, alongside differences in reporting between P and E-cores, visible in section 6.4.4.

Finally, we conclude by discussing our findings, drawing connections to related work, and finishing with possible starting points for future work.

Chapter 2

Background

This chapter provides the necessary background knowledge for this bachelor's thesis. We start with an explanation of heterogeneous processors, leading into an in depth explanation of the Hardware Feedback Interface (HFI)—Intel Thread Director, what it extends on, and how it works. Next, we explain related mechanisms that can be used to save power, notably the wait extensions found in newer Intel processors [8].Following this is a brief explanation of Performance Monitoring Counters (PMC) and the Running Average Power Limit (RAPL). To conclude the chapter, a brief explanation of microkernels, capabilities, and the currently implemented scheduling in seL4 is provided, ending on a short section on virtual machine passthrough.

2.1 Heterogeneous CPUs

A heterogeneous CPU is a processor in which not every core in the same package is built with the same microarchitecture. Various types of heterogeneous processors exist that integrate coprocessors, accelerators, or FPGAs to achieve higher efficiency or performance for specific tasks [25]. This is not a new concept in computing, dating back to the late 1980s with the Cydra-5 [26]. In this context, we restrict ourselves to heterogeneous processors with one socket and one Instruction Set Architecture (ISA), but different types of microarchitectures, where the focus is either on high power and performance or low power and higher energy efficiency.

The goal of such processor designs is to increase power efficiency by allowing lower priority or I/O tasks to be scheduled on less energy intensive cores and vice versa. To this end, the operating system scheduler or the developer must be aware of the different core types to actually achieve higher efficiency in heterogeneous processor systems.

Designing a microarchitecture that increases energy efficiency can take the form of removing power intensive features such as instruction-level parallelism or parts of the ISA such as AVX-512 support, like in the case of Alder Lake [27]. Alternatively, one can simplify the design of the core compared to the high performance version by reducing the size of the cache or shortening the instruction pipeline, lowering clock speeds. An example of reducing cache size can be seen in the E-core clusters of an Alder Lake processor, where E-cores are clustered in groups of four around a shared L2 cache [28]. Additionally, there exists a high degree of variability for processor manufacturers when designing heterogeneous processors, e.g., the ratio between performance and energy efficiency cores, the number of core types, or the degree of similarity between the performance and energy efficiency microarchitectures. Pioneered by ARM in 2011 with their *big.LITTLE* [4] and later *dynamIQ* [29] series of System on a Chip (SoC) microprocessors, these types of heterogeneous processors are commonplace in the embedded and mobile market, being present in Samsung's *Exynos* [30] and Qualcomm's *Snapdragon* line of microprocessors [31]. Recently, these architectures started appearing in the laptop and desktop market, starting with the M series of processors from Apple [5] and the Alder Lake series [6] of processors from Intel. AMD also produces a series of heterogeneous processors in the form of their 8000G lineup [7]. However, these differ from the previous examples, as they only appear in AMD's low power and mobile oriented line of processors and only being a heterogeneous processor by name, with Zen 4c being a more compact version of Zen 4 [7].

Core ID	Perf	EE
0	33	67
1	0	0
2	255	192
...	...	...
n	22	155

Figure 2.1: Hardware Feedback Table with example values

2.2 Hardware Feedback Interface

To allow for more flexible power management, the Hardware Feedback Interface (HFI) is a feature on modern x86 processors that monitors each logical processor and provides the operating system with scheduling hints via a memory resident table [8].

This table consists of a global header followed by a field for every logical processor that contains two numbers in the range [0-255], referred to as capabilities, one being for performance (Perf) and the other for energy efficiency (EE) [8]. fig. 2.1 serves as an example for how this table is laid out in memory.

The edges of these capabilities are assigned special meaning, with 255 for a core indicating that threads should primarily be scheduled on that core and 0, also referred to as "idling hints", indicating that the core should not be scheduled on [8].

When the scheduler is invoked, it is expected that a thread is scheduled on either the processor with the highest performance or energy efficiency capability, depending on the policy of the scheduler [8]. Taking the table from fig. 2.1 as an example, when the scheduler prioritizes for performance, it would preferentially schedule on core two and avoid scheduling on core one.

2.3 Thread Director

Intel Thread Director, sometimes described as the Enhanced Hardware Feedback Interface (EHFI) [1], is an extension of HFI found on Intel x86 processors. This extension takes the form of assigning every thread a class ID and expanding the memory resident table with capabilities for every class [10].

[1]This used to be the term found in the reference for future instruction set extensions [32], but has since been fully replaced by Thread Director [33]

Core ID	Class 3		Class 2		Class 1		Class 0	
	Perf	EE	Perf	EE	Perf	EE	Perf	EE
0	43	21	168	51	105	44	67	33
1	40	22	157	54	98	46	63	35
2	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...
n	41	105	41	71	41	94	38	115

Figure 2.2: Thread Director Table with example values, assuming four classes, modeled after [10]. Example values taken from [34].

The exact number of classes is intentionally left open, but both Alder Lake [10] and Meteor Lake [34] assign class IDs ranging from zero to three. An example of how the Thread Director table looks like is provided in fig. 2.2.

The class ID of a thread is stored in the `IA32_THREAD_FEEDBACK_CHAR` Model-specific Register (MSR)[2] on the core the thread is running on [8]. If the content is marked as invalid, either by manually disabling Thread Director, resetting Thread Director History via the new instruction `HRESET`, if enabled via the `IA32_HRESET_ENABLE` MSR, or the hardware not having collected enough information to assign an ID, the scheduler is expected to use the last known class ID for scheduling [8]. Collection of information only starts once Thread Director has been successfully configured by the scheduler, such as setting the memory address of the Thread Director table in the `IA32_HW_HEEDBACK_PTR` MSR, followed by setting the first two bits in the `IA32_HW_FEEDBACK_CONFIG` MSR [8].

Updates to the Thread Director table are signalized by the hardware either setting a sticky bit in the `IA32_PACKAGE_THERM_STATUS` MSR, which the OS has to regularly poll and clear to check for updates and acknowledge them, or enabling Thread Director interrupts during configuration by writing to the `IA32_PACKAGE_THERM_INTERRUPT` MSR [8]. Once the table has been updated for the first time, Thread Director can be used by the scheduler.

The assignment of classes to threads depends on the workload it is executing. In the case of a Meteor Lake processor, class zero is for scalar applications, one and two for vector applications, and three for applications with spin loops [34]. The same holds true for Arrow Lake, as we were able to replicate the same class assignments, see section 6.4.4.

When the scheduler is invoked, it should obtain the class ID of the thread and choose the highest capability of that class, similar to how it was previously

[2]MSRs are a set of control registers that contain features not found on all processors, some are architectural, others are only available on some revision, and are often used by Intel to experiment with new features [8]

done with HFI. To aid with understanding, we replay the example scheduling scenario from before, now using the Thread Director table from fig. 2.2. Assuming the scheduler prioritizes performance and the thread has a class ID of two, the scheduler looks into the Thread Director table, reads out the values of the class two column and choose to schedule the thread on core zero.

Backward compatibility with HFI is provided by treating a processor with only HFI as having just one class [8], which should be obvious when comparing the layout of the two tables. This allows a scheduler designed for Thread Director to also accommodate processors that do not have this feature, for instance, processors from AMD.

Obviously, a scheduler does not need to fully rely on Thread Director to make scheduling decisions, as both the processor itself and software developers can also contribute hints to the scheduler. An example of the former would be monitoring the processor's temperature, current power consumption via RAPL, see section 2.6, or utilization by keeping track of how long the idle thread was scheduled on that core, then based on these hints adjust scheduling decisions. Another, albeit less reliable, way is for a program to provide hints to the scheduler. Assuming one knows that a program is running as a background process or requires high amounts of computation then a programmer could either annotate their code or use system functions such that this information can be forwarded to the scheduler, which then changes behavior based on the type of program specified, for example changing the affinity of the program, setting a higher or lower priority or assistance in following a specific power plan.

To summarize, Thread Director is expected to provide an uplift in performance and energy efficiency on Intel's line of heterogeneous processors by assisting operating system schedulers with hints for optimal placement of threads.

2.4 Wait Extensions

As part of version three of the Streaming SIMD Extensions (SSE3), Intel introduced the `MWAIT` and `MONITOR` instructions, allowing the processor to arm a cache line to monitor for writes, putting the processor in a low-power state until a write to that region occurs [8]. The major issue with these instructions is that they can only be executed in ring 0 (supervisor mode), restricting them to the kernel. With the Tremont microarchitecture [8], Intel also introduced the *User wait instructions* `UMWAIT`, `UMONITOR`, unprivileged versions of `MWAIT` and `MONITOR`, and `TPAUSE` to pause for a specified length of time. `UMONITOR` and `UMWAIT` are utilized the same way as their privileged counterparts, with the operating system being able to define a maximum number of cycles that a processor may sleep using these instructions.

AMD offers a comparable pair of instructions, referred to as `MONITORX` and `MWAITX` [35]. Additionally, one can specify one of two idle states, allowing one to decide between higher energy efficiency and lower latency, and a duration of time. Intel's implementation expects a Time Stamp Counter (TSC) target [8], while AMD's implementation expects a duration specified in TSC cycles [35].

The intended use case for these instructions is to provide greater energy efficiency compared to the `PAUSE` instruction in synchronization primitives, where spin loops are commonly used [36], as they allow the CPU to enter a lighter or deeper sleep state, referred to as C0.1 and C0.2, depending on which one was requested. When executing either `TPAUSE` or `UMWAIT`, one can specify which of these sleep states to enter by setting the lowest bit of the `ECX` register prior to execution, with a one corresponding to C0.1 and a zero to C0.2. Do note, however, that this is merely a suggestion as to which state userspace desires the processor to enter, as both the hardware and the operating system may decide to demote the deeper C0.2 sleep state to the lighter C0.1 sleep state. From the operating system side this demotion can be configured in the `IA32_UMWAIT_CONTROL` register by setting the lowest bit [8], while the hardware performs this demotion if the requested duration is perceived to be too short to justify entering the deeper sleep state. In the case of Alder Lake, this seems to occur below durations of $\sim$22,800 cycles, as reported in [36].

An, as of writing unexplored, alternative use case is using these instructions for inter-thread synchronization, as mentioned in [36]. Additionally, these instructions could find practical application in development of more energy efficient userspace schedulers as a communication mechanism between scheduler and threads.

2.5 Performance-Monitoring Counters

To obtain low-level information about the current processor state, instruction execution, or cache misses, to name a few, Performance-Monitoring Counters (PMCs) are counters specific to a logical processor or across all cores[3], that either count the duration or number of occurrences of events [8]. They are implemented as a variable amount of MSRs `IA32_PMCx` and fixed counters (`IA32_FIXED_CTRx`), with the number being specified by `CPUID` [8]. The events able to be monitored are mostly model specific, excluding architectural performance events [8], including the number of decoded instructions, received interrupts, L1/L2/L3 cache misses, etc.

[3]Support was added in Architectural Performance Monitoring Version 3 [8]

Setting up a PMC involves a write to the specific performance event select MSR `IA32_PERFEVTSELx` or `IA32_FIXED_CTR_CTRL` that specifies what event to monitor and the condition that the logic unit detects, along with flags specifying whether counting should occur when operating at a privileged or unprivileged level [8]. Additionally, an interrupt can be sent in case of an overflow of the counter [8]. Then, reading the corresponding `IA32_PMCx` MSR provides the specified information [8].

PMCs are a helpful tool to collect various low-level processor statistics, allowing for lightweight performance measuring for various tasks, such as influencing scheduling decisions or evaluating the power draw of a system. These counters can be read either by `RDMSR` or also from userspace using `RDPMC` [8].

2.6 Running Average Power Limit

Running Average Power Limit (RAPL) interfaces consist of several MSRs that allow monitoring and controlling power consumption for x86 processors [8], introduced by Intel with Sandy Bridge in 2011, followed by AMD with a similar interface in Bulldozer [37]. Most modern processors include the `PP0`, `PP1` and `PKG` RAPL interface, with the interfaces providing energy consumption information about the cores, integrated graphics adapter, and socket, respectively [8]. What interfaces are present depends on the CPU platform, with server CPUs generally lacking a `PP1` domain but providing a `DRAM` instead. Each interface reports the current energy status and power limit of the component they are monitoring [8]. Some processors provide additional information, such as performance status, power information, which specifies ranges for a domain, e.g., minimum and maximum power, or policy [8].

At the hardware layer, measurement is done via voltage regulator current monitoring (VRIMON), where voltage regulators supply current to the CPU, estimating power consumption, which is regularly sampled by the processor. These samples are then used to calculate the reported energy draw, then written into MSRs to be read out by software [38], currently limited to ring 0 due to the need to read it with `RDMSR` [8]. This allows them to be used for high accuracy and low overhead [39] power measurement.

Therefore, RAPL finds its use in system evaluation purposes, such as measuring the energy efficiency of a program or operating system component, or identifying high energy consumption tasks in a system. With a resolution of up to 1 ms, it is also suitable to characterize fast changing distinct phases in a program during execution, which would otherwise be imperceptible by external power measurement devices, like a wall plug power meter, as their resolution is in the order of 100 ms [39].

2.6.1 Limiting Power Draw

Another feature that RAPL provides is the ability to limit the maximum power draw of individual RAPL interfaces via the `MSR_(PKG/PP0/PP1/DRAM)_POWER_LIMIT` MSRs [8]. What MSRs are available once again depends on the CPU model. The granularity of the interface can be inferred from the *Power Unit* field of the `MSR_RAPL_POWER_UNIT` MSR, the default being increments of $\frac{1}{8}$ Watts.

When setting a power limit, there are two values to be aware of: Power Level 1 and 2 (PL1 and PL2), where PL2 represents the maximum power that can be drawn for a short duration, which is intended for boosting performance for short-term loads, while PL1 is the average power draw, and the value typically set to the Thermal Design Wattage (TDP) of the processor, that PL2 decays to for sustained loads [8]. PL2 only exists for the `PKG` domain, with the other domains only providing a field for PL1. Additionally, PL2 may be hardcoded by hardware, e.g., set by the firmware, ignoring writes from software [8]. The range of wattage and time frame that the processor stays in a given mode can be found in the `MSR_PKG_POWER_INFO` MSR [8], specifying the maximum time frame alongside the maximum, minimum and TDP of the processor.

Alongside PL1 and PL2 exists PL4, which governs the absolute maximum power draw of the package, meaning a hard limit is set by the hardware that cannot exceed the limit specified in PL4 [8]. This is normally intended as a safe-guard, preventing physical damage to the processor due to drawing too much current, but can also be used to enforce a power limit lower than what PL2 and PL1 allow. Documentation for PL4 is poor, only a brief appearance in the listing of all MSR. PL4 is configured in increments of $\frac{1}{8}$A, which has to be set in the `MSR_VR_CURRENT_CONFIG` MSR. We assume this MSR is locked on some platforms, e.g., laptops, and complete documentation thus only available to firmware manufactures.

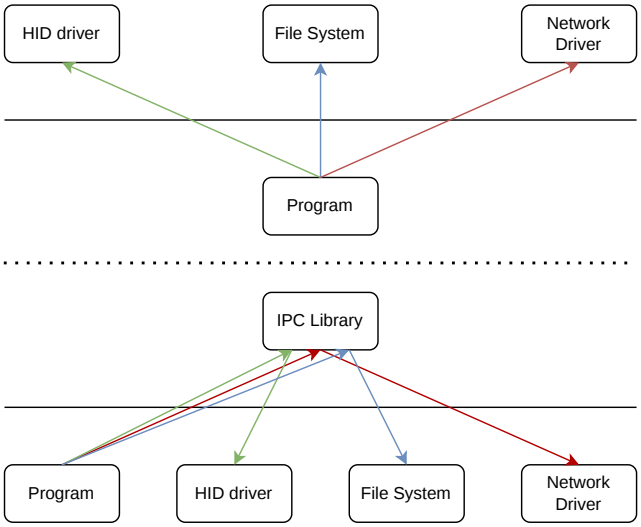

Figure 2.3: Example program running on a monolithic kernel and a microkernel, which are the top and bottom half, respectively

2.7 Microkernels

In general, when designing an operating system, there are two approaches one can take. Either one bundles everything needed into one large component with all privileges to maximize performance. This approach is better known as a *monolithic kernel* and is, as of writing, the predominant design, found in the Linux kernel, some BSD kernels like NetBSD or FreeBSD, and to a lesser extent as a hybrid design, in the Windows NT Kernel [40].

Alternatively, one can only bundle what is absolutely necessary into a smaller privileged component and run everything else as regular processes. This is known as a *microkernel*, the design of which is best summarized by Liedke's minimality principle, stating that concepts are only permitted in the kernel if moving it out of the kernel would compromise the functionality of the system [15]. Therefore, a microkernel only provides abstractions for processes, memory, and Inter-Process Communication (IPC), the rest being handled by userspace processes known as servers.

What this means in practice can be seen in fig. 2.3, where we show an example program running on a monolithic and a microkernel. In the monolithic kernel, all system components reside in the kernel, meaning every interaction requires a system call, which can be expensive on a monolithic system, and all components run in kernel-mode, meaning a malfunctioning service may take down the entire

system. In a microkernel we still need to enter the kernel, perhaps even more frequently than in a monolithic kernel, but mostly with the sole purpose of transmitting data between programs via IPC [4], which can (and has been) optimized in a microkernel. Components are split off into their own programs, isolating them in the case of failure, which also allows the user to customize them to their needs.

2.7.1 Historic and Current Microkernels

The first generation of microkernels, like the Mach or L3 microkernel, promised better fault isolation compared to their monolithic counterparts, successfully moving parts that were not essential from kernel to userspace, an example being Mach's external pager [16]. However, these microkernels suffered greatly in performance compared to their monolithic counterparts because of slow communication between parts of the operating system. Within a monolithic kernel, communication is simple, as everything is inside one program, one address space, but in a microkernel, different parts of the OS are encapsulated into their own programs, and communication needs to be handled via dedicated endpoints, IPC. If the IPC mechanism in a microkernel is slow or inefficient, it slows down the entire operating system, as programs spend a majority of their time waiting for responses from other programs, which led to the development of the second generation of microkernels, the L4 family. Through careful design, IPC speeds were improved by a factor of 20 compared to Mach [14], disproving the then-common myth that a microkernel is inherently slower than its monolithic counterpart. Further developments in L4 revolved mostly around the port from Assembly to higher-level languages like C++ with the release of L4Ka::Hazelnut and L4Ka::Pistachio [22], and further improvements in IPC performance [15, 41], as this is directly linked to general system performance on microkernels.

The third and current generation of microkernels concerns itself with the formal verification. Microkernels, being comparatively small programs, proved to be a viable candidate for formal verification, where a formal proof exists verifying that a system is behaving according to its specification. This is especially important given that any code that is inside a kernel has to be trusted to function exactly as specified to guarantee system security, building a so-called Trusted Computing Base.

[4] Other purposes exist as well, such as performing tasks that can only be handled in kernel-mode

Combining these two concepts provides one with a kernel that is proven to behave exactly as specified, allowing for use in high-trust environments. One of these third generation microkernels is seL4, a reimplementation of L4 in C avoiding the issues found with earlier revisions of L4 [22], and the first general purpose operating system to be formally verified [19].

2.7.2 Capabilities

When managing system resources, one has to design a system such that these resources are protected from malicious actors, such as a malicious program interfering with other programs. This takes the form of access control, where access to resources is restricted to only authorized actors.

An example of such a system are Access Control Lists (ACLs), used in most operating systems like Microsoft Windows [42] and Linux [43], where permissions to objects are handled by a list containing every user or group of users of a system and their rights to specific resources of the system.

Although such a system is simple to implement and effective, it suffers from an unsolvable security flaw, termed the confused deputy problem [44]. This is a scenario where a program with lower privileges invokes another with higher privileges to perform malicious actions, by taking advantage of the permissions granted to the program with higher system privileges [44, 45].

A common example is that of a compiler, where the compiler has access to a file the user does not. The user can invoke the compiler and specify an output file, to which the compiler writes to. If the user acts maliciously and specifies the inaccessible file as the output, the compiler has been tricked into writing to a protected file [44, 45].

This is solved by a capability-based access control system, where access to resources is controlled by capabilities [44, 45]. A *capability* is made up of an unforgeable reference to an object and the permissions given to the actor for said object [23].

In the above example, the user would use their capability to invoke the compiler and provide the compiler with capabilities to the input and output file. Since the user lacks the permission to write to that file, the compiler fails to overwrite the protected file, preventing the confused deputy problem from occurring [44, 45].

This is the predominant access control model in the L4 family of microkernels [22], including seL4 [23], where all interaction with the kernel is handled via capabilities. Creation of capabilities is done by retyping untyped memory capabilities into the desired capabilities, including the creation of smaller untyped memory capabilities, which are then stored in the kernel. The kernel maintains

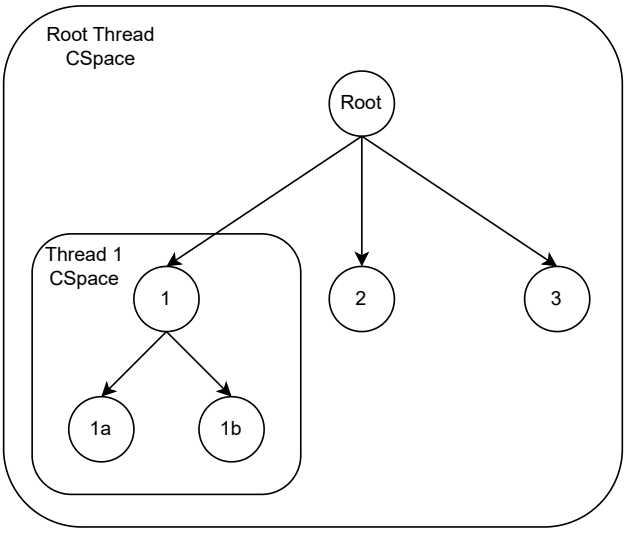

Figure 2.4: Visualization of CNodes of threads and their corresponding CSpaces

Capability Nodes, or *CNodes*, which are a table of slots which may contain a capability, which could be another CNode, forming a directed graph [23].

Each thread has a capability space, or *CSpace*, which are all capabilities that a thread owns. This CSpace encompasses all CNodes reachable from a thread's CNode root, which defines their CSpace [23].

Capabilities can be moved, copied, sent in messages (which is called a capability transfer) or revoked, the latter causing a cascading revocation of all capabilities derived from the revoked capability, as the kernel keeps track of derivations to a capability with a Capability Derivation Tree. Additionally, new capabilities can be created from old capabilities with a subset of their permissions [23]. Management of all in-kernel data structures is done in userspace, including the management of CSpaces.

2.7.3 Scheduling in L4

Scheduling in L4 has largely remained untouched by researchers or developers, being implemented as a simple hard-priority round-robin scheduler [22]. In the case of seL4 [23], the scheduler does not schedule threads to different cores on its own, degrading performance. While this leads to very predictable behavior, it presents the issue of violating the minimality principle by hard-coding policy into the kernel.

Generally, this also leads to lower overhead when compared to other schedulers, like the ones found in Linux or Windows, as simply fewer calculations have to be done inside the kernel.

In terms of overhead, we can isolate two areas: scheduling and dispatching, where scheduling overhead refers to the time spent deciding which thread to run, including possible accounting for other factors in the system like current load or temperature. Dispatching overhead refers to the overhead incurred from a context switch from one thread to another on one core, which is negligible on a sensible system, where timeslices are sufficiently long, as the cost of scheduling in this case far exceeds the overhead of dispatching a new thread. Should a switch happen across cores, we then refer to this as thread migration, as state data now needs to be migrated from one core to another. This carries with it additional overhead, as now the execution of another core needs to be interrupted for scheduling, and performance degradation, as local caches need to be invalidated, leading to loss of data locality and higher latency, with these issues being worsened the more frequent thread migration occurs.

When we compare to an approach where scheduling occurs in userspace, the cost of scheduling increases depending on the amount of data required that is now privileged, while dispatching now requires an entry point into the kernel, where it was previously included with scheduling.

Typically, the longer an entry into the kernel takes, the worse the performance degradation will be. Conversely, if the duration it takes to enter the kernel is very short, like it usually is in microkernels to facilitate fast IPC, the difference compared to normal interruptions caused by timer interrupts should be negligible.

2.8 Virtual Machine Passthrough

Under normal circumstances, the state of a virtual machine during execution is fully isolated from the state of the host machine as a security measure. For Virtual Machine Monitors (VMMs) like QEMU+KVM, where the virtual machine runs as close to bare metal as possible, accesses to hardware information, like for example the `CPUID` instruction, need to be trapped and emulated by KVM, which carries with it a performance penalty, as this carries with it an exit and reentry into the virtual machine, which are expensive operations.

Sometimes, it is acceptable to pass some host information to be accessed by the guest machine, which is where KVM allows a VMM to disable interception of specific operations or access to a specific MSR, allowing a guest to run or access them directly [46]. This in turn allows us to effectively work inside a guest as if we were running on a bare-metal system, which we took advantage of in our implementation for some MSRs like the class ID, `APERF/MPERF` and `RAPL` data, see section 5.1.2.

Chapter 3

Related Work

This chapter presents related work in the areas of Thread Director and Userspace Scheduling, concluding with a brief mention of userspace scheduling in L4.

3.1 Thread Director

Research into Intel Thread Director is limited, but this can mostly be attributed to the recency of Thread Director and the lack of documentation on its behavior until recently [10].

The work of Chun et al. [34] provides valuable insights into the behavior of Thread Director on a Meteor Lake processor. The demonstration of how capabilities are set and changed by Thread Director in various states of the processor, such as different loads or behavior in a throttling scenario, proved to be of significant value in our implementation, see chapter 5.

Despite being examined on a Meteor Lake processor, we found that most insights into Thread Director presented here stay valid for our Arrow Lake system. One caveat we found is that we cannot replicate their findings regarding the assigned class IDs assigned to threads. While they were able to observe full classification on both P- and E-Cores, we only obtained all four classes on a P-Core, while E-Cores failed to report class two. We assume this to either be intentional behavior, as class two is intended to represent high power-consumption vector instructions, which do not necessarily scale with power efficiency on E-cores as the classification would imply, or a bug in the hardware, which as of writing is still unknown to Intel [47].

Additionally, we did not test scenarios where other components of the processor, such as integrated GPU or NPU, are utilized, as drivers for these components do not exist for seL4 and would fall outside the scope of this work to

implement, despite being relevant for Thread Director when updating capabilities in the Thread Director table [34].

Earlier work investigating the hints provided by Thread Director on an Alder Lake processor by Saez and Prieto-Matias [48] showed more mixed results, concluding that a Machine Learning model based on PMCs is more accurate. We see several issues with this work, notably the exclusion of E-cores, due to being unable to obtain a valid class ID, possibly tainting the results. Additionally, this work is only concerned with the performance capability reported by Thread Director, while energy efficiency was not taken into account. Whether idling or consolidation hints were also taken into account is uncertain, as they are not mentioned in this paper.

In their later work for PMCSched, performed by Bilbao et al. [49], they showed that the inability to obtain a valid class ID was also present in the patchset submitted by Intel, but still does not take the energy efficiency values provided by Thread Director into account.

We can assume that this will not be an issue on our Arrow Lake processor, but should this be fixed in Alder Lake processors in the future, a reevaluation of Thread Director on Alder Lake that also takes the provided energy efficiency values into account would provide valuable insights whether Thread Director predictions have improved over time with newer processors.

Another paper, released after we started working on this thesis, presents HARP, a resource management framework for heterogeneous processors, which serves as an extension to existing scheduling, which shows high promise for these systems, reducing both execution time and energy consumption significantly [50]. As part of their evaluation, Smejkal et al. implemented a version of HARP using the classification of Thread Director for thread-to-core assignment, showing that for both energy efficiency and performance, improvements from Thread Director are within margin of error in single-application workloads and worse in multi-application workloads, lining up with previous observations from Saez and Prieto-Matias, Bilbao et al.

3.2 Userspace Scheduling

Userspace scheduling is a very active area of research [51–60], extending back into the early 1990s with scheduler activations [61]. Most of these projects are centered towards datacenter usage [51–55], where flexibility in scheduling is key in achieving lower tail latencies. This is done primarily to keep up with increasing network speeds, where the limiting factor becomes the speed at which requests can be processed and sent off again [53]. To achieve these increasing networking speeds, networking cards are installed in data centers that support

faster and faster interfaces, like kernel-bypass and RDMA. One advantage of these networking cards is that they are not concerned where the data ends up in the system, meaning processing of networking requests can just as well be done in userspace instead of the kernel, and with the tools provided to userspace in a microkernel, seL4 may be a good candidate to develop a networking driver for, which then could be coupled with an userspace scheduler to further increase processing speed.

More recent approaches are starting to generalize their interfaces for broader use cases [55, 56], take advantage of kernel bypass, such as userspace interrupts [56, 57] to achieve even more impressive results, or shorten development time of new schedulers while retaining support for existing approaches and their performance benefits [58].

All of the listed approaches implement some sort of scheduler in userspace that contains the desired scheduling policy, which then communicates with a kernel module that performs the actual scheduling.

If one can work within the limitations imposed, a userspace scheduler can already be created in Linux without resorting to a kernel module by making use of `sched_ext`, or the extensible scheduler. This is a feature found in the Linux kernel since version 6.12 that allows users to write custom schedulers in eBPF and dynamically load them into the kernel [62]. Compared to writing a kernel module, using `sched_ext` is both far simpler for a developer, as there are plenty of resources to work with and no need to figure out kernel development, and safer, as a malfunctioning scheduler at worst stalls execution of threads before being unloaded and replaced with the default kernel scheduler. The drawback here is that should one want to perform additional functions that `sched_ext` does not provide, they would either need to implement them in a compatible way or resort back to a kernel module.

Additionally, we see that other approaches in the realm of storage, such as *SPDK+* [63], *Sandman* [64] or *Aeolia* [65], benefit massively from kernel bypass when looking at performance or energy efficiency.

We conclude that porting these approaches to a microkernel, such as seL4, where the kernel can almost fully be bypassed, could show further potential for these projects and that our approach would inspire more research into microkernels, broader adaption of L4, or future improvements of hardware interfaces.

We are aware of at least two implementations of userspace scheduling that have been implemented in an L4 kernel. The first is the work of Stoess [13], with the other being a partitioned scheduler in seL4, which is the work of Åsberg and Nolte [66]. However, both implementations were tested on very old processors, a Pentium D830 and a Pentium 3 respectively, which are architecturally too different and archaic to be comparable to the performance of modern processors. Additionally, as both have been tested on such old processors, techniques to improve userspace performance, such as HFI or unprivileged version of instructions necessary for scheduling, have not been implemented, leaving room for possible reductions in measured performance overhead.

3.2.1 Partitioned Multikernel

An interesting approach to SMP proposed for seL4 is the concept of a partitioned multikernel, where instead of making the kernel SMP-aware, which is hard to formally verify, a single-core version of the kernel runs on each core and communication is handled over IPIs. This has the benefit of supporting multiple cores, while retaining the formal verification of the single-core variant [67].

We position our proposed design as a variant of this system, where the kernel only runs on one core, the one responsible for scheduling, and the other cores are fully managed by our userspace scheduler. We still could retain the single-core verification for our scheduling core, while the other cores, being managed fully in userspace, need not be verified. Furthermore, we also support not only "true" SMP support, but also support for heterogeneous processors. Some details remain unclear for this to be a viable alternative, which we discuss in section 7.2.1. We also see value in the other approach and do not claim that our idea is superior in any way, as both approaches may be applicable depending on what is built on top of the seL4 kernel.

Chapter 4

Design

This chapter presents our design ideas for a userspace scheduler on a microkernel system taking full advantage of the features provided by modern multicore hardware. First, we define what makes up a scheduler, how it can be divided, and what can reasonably be shifted into userspace. Next, we introduce the *ITD* capability and detail the invocations needed to facilitate scheduling from userspace. In line with the original design goal of microkernels, we explain why we added another data structure for managing threads to the kernel and how we use it to avoid as much policy in the kernel as possible.

As we wish to examine possible energy efficiency improvements that can be gained with Thread Director, we next elaborate what needs to be added to the kernel to allow userspace access to Thread Director or, in general, to similar information provided by hardware. While not strictly necessary for userspace scheduling, it does highlight that userspace schedulers can benefit from hardware feedback during policy decisions and the kernel should allow for userspace to be able to access it via capabilities.

With the necessary considerations made for kernel space, we follow up with the userspace side of scheduling, detailing the functions required to stay the same across schedulers and what userspace can freely design. For this we discuss what userspace now has to manage and possible starting points for scheduler design. Finally, we summarize what responsibilities can currently be shifted into userspace and what still needs to remain in the kernel, mentioning possible future extensions that can help further decrease kernel involvement in userspace scheduling.

4.1 Responsibilities Of A Scheduler

Before continuing, we feel the need to mention that the scheduling we are concerned about is the scheduling of executable threads on processor cores on a heterogeneous processor. We assume scheduling for other computing resources can already be accomplished by the capabilities and invocations granted to userspace, but we will not further investigate these claims, as this falls well outside of the scope of this thesis.

Within the scope of our thesis, we observe four areas that scheduling needs to be concerned with:

1. Abstraction and management of executable threads
2. Selecting threads to run on a specific core
3. Balance threads across all available cores
4. Account for other resources in a system, such as current power draw, load, other hardware metrics, and integrate them into policy decisions

This allows for a rough split of management and policy between the first and the latter three points. We are under the assumption that management should stay within the responsibility of the kernel [1], while policy can and should be migrated to userspace. Though, given current developments for future features [33] allowing for more operations to take place in userspace, more features could be moved to userspace in the future, leaving only the minimum amount required to the kernel. And given similar findings in prior work [56, 57, 63, 64] shifting operations to userspace, we conclude that this would probably yield further improvements in speed and efficiency.

We term this concept, for which a direct analog does not really exist [2], but is closest to *scheduler activations* [61], *user-managed kernel-level threads*. The aim with this approach is to still work with kernel-level threads, so the kernel is still aware of them, e.g., for IPC between threads or fault handling, but avoid setting a scheduling policy in the kernel. Programs written to take advantage of our design can probably gain performance and energy-efficiency benefits, which we put to the test in section 6.4. Even if the results do not show improvements, there may be a specific scheduling policy we did not look into that may perform better for specific tasks, in performance or energy efficiency.

Historically, this flexibility has come at the cost of reduced performance [13, 66], to the degree that, despite violating the minimality principle, every L4-based

[1] Note, it is still possible to use pure user-level scheduling on top of these abstractions.

[2] Likely for good reason, given the relative scarcity of OSes with capability-based access control

microkernel retains scheduling in-kernel to avoid this performance hit [22]. However, given the recent developments of CPU hardware and the addition of more and more formerly-kernel-exclusive instructions, driven by the increased interest in userspace scheduling (see section 3.2). We propose that designing such a system on x86_64 is now not only feasible but no longer carries the burden of performance degradation, nearly to the degree it has historically.

4.2 Scheduling With Capabilities

Our designed system, which allows this degree of flexibility, needs to first extend the current kernel. As we are working with seL4 for this thesis, the natural starting point for extending functionality is the addition of a new capability, *ITD*, which controls access to the Thread Director Table and userspace scheduling [3]. This has several benefits, including cleaner integration into the existing infrastructure of seL4 and security by means of restricting access to scheduling to only those who have access to the necessary capability, see section 2.7.2 as a refresher.

4.2.1 Capabilities And Invocations

seL4, being designed around capability-based access control, does not have system calls in the traditional sense, with only a few exceptions:

- `Yield`, which is intended for yielding the current time slice to the kernel scheduler and is thus of no interest to us. Should a thread controlled by a userspace scheduler invoke this system call regardless, then this needs to be caught and either ignored or responded to with an error.
- `Send`, which sends data via a capability.
- `Receive`, which receives data via a capability.

For reasons of optimization, there also are non-blocking variants of `Send` and `Receive`, which are prefixed with `NB`, and a combined send and receive system call, called `Call` [23]. Every other interaction with the kernel requires the use of `Call` with a capability. To make this process easier, seL4 includes a generator script that works in conjunction with a documentation generator script. To add new invocations, one documents their function in XML, specifying the name, capability, and in- and outputs. The documented function in XML is converted to a stub in `libsel4` [68], the API of the seL4 kernel, which correctly handles input and output of variables.

[3] Later we realized this to be flawed, see section 7.2.3

For scheduling, we decided to add the following invocations:

- `X86ITDUSchedAttach`, which adds a thread to a core.
- `X86ITDUSchedDetach`, the complementary function to attach.
- `X86ITDUSchedRelocate`, a combination of a detach from one core to an attach to another
- `X86ITDUSchedAdvance`, which selects a new thread to run. For further details, see section 4.3.

4.2.2 Maintaining Compatibility

An important topic for even relativity new operating systems is compatibility for legacy applications. One concern we had when designing userspace scheduling was that of scope, specifically whether userspace scheduling should extend to all cores or only a predefined subset. This led to the addition of a reservation mask into the kernel, which is a bitmask of all cores detected at boot [4]. To keep things simple, we explicitly do not support hotplugging of CPUs in our design, which is not a concern for the vast majority of platforms, but does present some potential edge cases for platforms that do support it. Should one deploy our design in such a setting, then support should be possible to add on in a similar matter to how it is implemented in the Linux kernel [69].

During runtime, via the `X86ITDSetCoreReservationMask` invocation, userspace can "reserve" cores for userspace scheduling, which are then marked as unavailable for the kernel scheduler. We designed this to be dynamic, meaning later during runtime the reserved cores can be "reclaimed" to be used by the kernel and its scheduler again. Additionally, threads not assigned to userspace cannot be moved into reserved cores and vice versa. To be able to tell whether threads are assigned to userspace or not, we extended the recently introduced `TCBFlags` [70, 71] mask with a bit designating a thread as user controlled. With this, both schedulers can coexist at runtime, with newer software being able to utilize userspace scheduling, while older software can continue without needing to be adapted.

4.2.3 Extending Support

Many operations in seL4 follow the same trend and structure. Therefore, it is very common for systems built on top of seL4 to abstract these into a library, with thread creation being no exception. Therefore, it may be possible that support

[4] This means that, for example, on a 20 core machine, the first 20 bits are set.

for userspace scheduling to be handled by such a library. Either checking for an existing, or starting its own userspace scheduler and adding the thread to it during thread creation. As the process of thread creation is mostly the same between our approach and the unmodified kernel, adding support transparently may be possible with specific changes in either the `musl` or `libsel4` library. However, this should be viewed as theoretical, as we did not pursue this in our implementation and cannot assess the difficulty of this approach. If this is pursued, special care should also be taken to transparently handle a process trying to yield, which would need to be intercepted by the corresponding library and redirected to the right scheduler, kernel or userspace. There should not be a reason that this is not possible, considering similar approaches have been implemented in the past.

4.3 Avoiding Policy In The Kernel

With the interfaces set up, we now need a data structure to keep track of threads on a specific core. For this we cannot simply reuse the existing ready queues found in the kernel, as we aim to sidestep the existing seL4 scheduler in our design and want to avoid any sort of policy in the kernel, which the ready queues with their priority-based sorting impose. In Linux, it would likely be possible to share the queue between the two schedulers, but here in seL4, to keep changes in the kernel as minimal as possible, we did not pursue this. Additionally, we have two other design considerations to make:

- We want to spend as little amount of time in the kernel as possible.
- We cannot dynamically allocate memory in the kernel [22], as all memory in seL4 has to be explicitly given to the kernel via capabilities.

We are also unaware of an object in seL4 that allows sharing an object between kernel and userspace, likely for security concerns and the number of required checks to implement for this mechanism, which is why we keep this potential structure fully in the kernel.

Ultimately, we settled on a per-core ring buffer with two indices, one storing the capacity, the other pointing to the current thread. This design fulfills the previously laid out requirements and also allows for addition, removal, and iteration [5] in O(1) time. *Addition* appends to the end of the ring buffer, if there still is capacity, advancing increments the pointer index and sets the current thread. If a core is flooded with threads, it will only attach as many threads as it has capacity, while rejecting further additions with an error message. *Relocating* from another core to the flooded core is also considered the same as addition, meaning

[5] Recall the attach, detach, and advance invocations from section 4.2.1

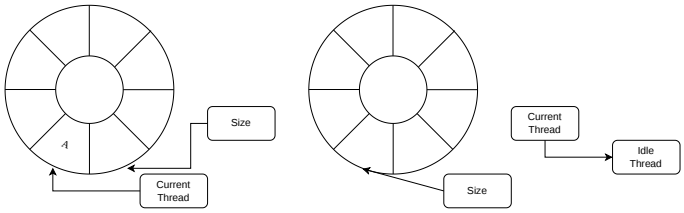

Figure 4.1: Removing the last element, ring buffer is now empty

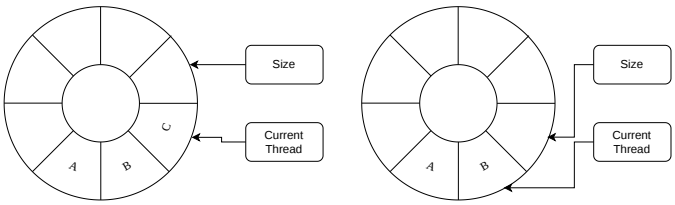

Figure 4.2: Removing the last element, buffer not empty.

the thread stays attached to the original core. What capacity to choose depends on the maximum amount of threads one expects to be located on one core at a time, and needs fine tuning based on what the system is designed for. While high values only waste memory, low values could hinder system performance. For this, we added a build-time define, which specifies the capacity, to the build system, allowing deployments of seL4 to adjust based on their needs. The exact value we settled on for the capacity is discussed in section 5.3, but is of rather low priority as a whole.

Removal either removes the last element, if we request to remove the last element, or replaces the current element with the last. With removal we also need to take care when removing the current thread, by setting the current thread to the thread where the pointer index now points to or the idle thread, if no more threads are in the buffer. To clarify what we mean, we illustrate these different scenarios in figs. 4.1 to 4.3.

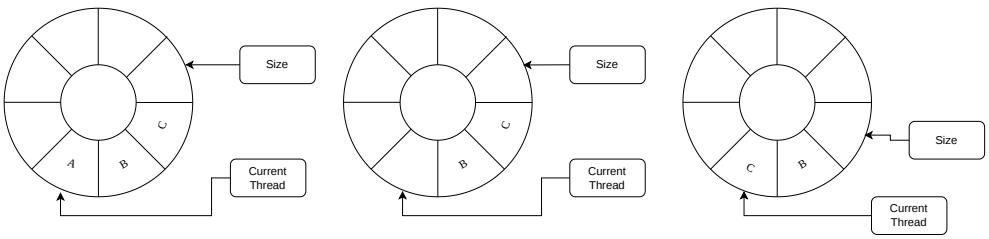

Figure 4.3: Removing an element which is not the last element

To ease removal of arbitrary threads in a ring buffer, we appended the `arch_tcb` structure, which is part of the Thread Control Block (TCB) and stores architecture-specific information, with the index of where the thread is in the ring buffer.

For kernel and userspace to cooperate, the current state of the ring buffer needs to be synchronized between the two. Therefore, the bookkeeping that the kernel performs needs also be done by the userspace scheduler [6]. Should user and kernel become desynchronized, the kernel will discard nonsensical requests instead of raising an exception or faulting. This could come from a faulty implementation of an userspace scheduler, which does not correctly manage thread metadata, as was the case during development of our prototype, or interference from another userspace scheduler, where threads that one expected to be there are now found in another scheduler. For one, as the kernel, we are only concerned with not exhausting our internal buffer, as this would endanger the security and safety of the kernel, as this buffer overflow might lead to data corruption and possible code execution. Additionally, if we would instead fault, this would open up an attack vector where an application floods a core with more threads than it can handle, causing the userspace scheduler to fault, or outright crash the system with an exception. Therefore, discarding nonsensical requests is both the safest and easiest approach for both the kernel and userspace.

4.4 Hardware Assistance

With all changes and extensions in the kernel, which are strictly required for userspace scheduling, being present, we can further take advantage of hardware-assisted scheduling to, e.g., optimize for energy efficiency. Therefore, we also need to add a way for the userspace to be able to obtain relevant information useful for userspace scheduling.

To support Thread Director, we first need to check that the hardware has support for this processor feature via `CPUID`, then configure the necessary MSRs, and finally provide the userspace component with the Thread Director table and feedback classes.

For providing feedback, we added two more invocations: `TCBGetThreadClass`, returning the thread class of a thread, and `X86ITDGetITDTable`, which writes a copy of the Thread Director table to a provided address. Note, this could easily be extended to support more information like current frequency, temperature, and other core related information.

[6] We also realized that we did not consider a way for userspace to "recover" from a bookkeeping error, alongside other fault conditions, which we mention in section 4.7

Obtaining the class ID of a thread is relatively straightforward: after configuration and some time in userspace [7], the OS can read out the class ID via the `IA32_THREAD_FEEDBACK_CHAR` MSR, which contains a value from zero to currently three alongside a valid bit. The class ID of a thread is stored the same way as its position in the ring buffer, as a field in the `arch_tcb` structure. In our design, we read out this MSR when invoked, updating the stored value in the TCB, if valid, and returning the stored value to userspace. This follows the recommendation of the manual that should there be no valid class available on the current core, then the last known class ID should be used [8].

In an ideal setting, we would allow userspace direct access to the Thread Director table, however, the `IA32_HW_FEEDBACK_PTR` MSR, which stores the address where the hardware will write the table into, requires a physical memory address of a page frame [8], while userspace can only provide us with a virtual address. Additionally, it is uncertain whether the hardware requires the page to be a kernel page or not [8], which the manual does not clarify [8]. Instead, we reserve a static page in the kernel for the hardware to write the table into, and copy the values on invocation to a user-provided address [9]. On a system like Linux, where the kernel is far more involved, this could be handled via a read-only mapping for the userspace process. But as seL4 does not provide virtual memory management [23], instead handing this task off to userspace, this is not an option. The only page table that is mapped by the kernel is the highest-level page table for the specific architecture, for the root task, with threads needing to map the architecture-specific intermediary paging structures themselves for their *VSpace* [10]. Once these are mapped, userspace can define their own address space as desired [23].

For a cleaner design, we do not enable Thread Director by default, if the hardware has support for it. Instead, we encapsulate it into two more invocations: `X86ITDInitialize` and `X86ITDTeardown`, which enable and disable Thread Director and its associated MSRs accordingly. When run for the first time, `X86ITDInitialize` checks for hardware support, then enables Thread Director. This saves us from rechecking support when disabling and re-enabling Thread Director at a later point.

[7] 6 to 21 microseconds according to [34], we have made other observations 6.4.4

[8] This means whether or not the user bit is set or cleared on that page table entry, marking it as accessible for userspace or not

[9] We are aware of the security implications, see section 4.4.1

[10] The seL4 term for the virtual memory space of a thread [23]

4.4.1 Invocation Return Types

In seL4, invocations can return values, which can be specified in XML. During generation of the stubs, this in turn generates a C structure with the return values and an additional `error` value for returning error codes to userspace. For more complex return types, one can define a `struct` to be returned. However, as of writing, this structure must be statically defined, which is not possible for the Thread Director table, as its size is dependent on the number of cores of the physical hardware. For this reason, we circumvent this restriction when returning the Thread Director Table by using `memcpy` on a userspace-provided memory address, which could be abused by malicious actors to overwrite arbitrary regions of memory, along any other exploit that can come with an easy-to-manipulate `memcpy`. The correct way to do this would be implementing the seL4 equivalent of the Linux `copy_to_user` function, which performs additional checks like checking if the provided address does not point into kernel memory, graceful handling of faults or other architecture-specific considerations [72].

As a side note, according to prior work [34], the Thread Director table is updated rather infrequently, in the order of hundreds of milliseconds. Therefore, we expect this approach of periodically polling the kernel once every half second for updates does not need much resources or reduce efficiency.

4.5 Userspace Scheduling

Now that we have the required foundation, we can now detail our ideas for a userspace scheduler built with the kernel tooling we designed. First, to stay synchronized with the kernel, we need to keep track of each thread, we currently manage and account for how the kernel handles additions and removals. For this, we designed the `track_addition`, `track_deletion` and, for convenience, `track_relocation` functions, which should always be invoked after their corresponding invocation in the kernel. For safety, a library interface would likely bundle the two together in a wrapper function, with their current implementations being hidden inside the library. For invocations, we solely rely on the kernel state, such that userspace mismanaging their state does not cause security concerns for the system. We see it as unlikely that more changes would be needed, but a function that allows userspace to synchronize state, while expensive, could be useful as a last resort before termination, should userspace desynchronize. Some additional functions that check state between the two, which can be run at will by the scheduler, could also be useful, but we leave a concrete design for such a checking function to future work.

We expect these bookkeeping functions to be present unchanged in any implementation based on our design, but further design is intentionally left open to the user. Should one design a scheduler where threads carry more metadata in userspace, then these functions also allow taking in a function pointer, alongside a pointer and a size, with the function being implemented by the user according to their needs. Then this function, if provided, is executed after the mandatory part that is specific to our kernel implementation, to handle the additional data.

Next, we need to consider the type of scheduler we design for. For this, there are two main possible designs: a centralized scheduler, where the scheduler resides on a single core, with the reserved cores being fully subordinate to a userspace scheduler, or a distributed scheduler, where every reserved core has a scheduler running on it.

For this thesis, we focus on a centralized scheduling model, but a distributed scheduler built on top of our changes would be possible but requires changes needed to facilitate communication and data transfer between threads. The main reason why we decided against a distributed scheduler was the issue of preemption: When running a scheduler in userspace alongside threads, we need a way to be periodically preempted such that the scheduler can run.

For this we would need to either make use of an application threading library that maps many user level threads to one kernel thread, with a scheduling thread that is periodically chosen, or extend the kernel to handle this case for us, either one coming with its own downsides. Or we can avoid both of these and instead employ *userspace interrupts*, e.g., as currently available on newer Intel CPUs [8], which solves this problem in userspace for machines with fewer than 64 cores, by interrupting the running thread and redirecting execution flow to our scheduler, both without entering the kernel. For machines with more than 64 cores, this somewhat falls apart, as an affinity mask (based on machine word size) can only accommodate at most 64 cores. Here, a solution would be something similar to the processor groups found in Microsoft Windows [73], where cores are encapsulated into groups of at most 64 cores and a scheduler can only be assigned to at most one cluster at a time.

In a future design, accounting for new instructions added since [33], one could likely redirect a timer interrupt into userspace with Userspace Interrupts, or make use of user timers, and design a distributed scheduling model with this, see section 7.2.

Additionally, during the implementation of our prototype, we discovered that while a remote call (IPI) can be broadcast to any number of cores, only one IPI at a time is supported, with subsequent IPIs being stalled until the first is delivered [74].

In a distributed system, where frequent interrupts from one core to another is key, this presents a non-trivial problem for a potential design, which would

involve a rework of how IPIs are handled internally, a solution which we present in section 7.2.1.

With a centralized scheduler the main scheduling loop should at least perform the following tasks:

- Deciding what threads to run on a specific core.
- Deciding what threads to move between cores.
- Setting up asynchronous notifications from threads to the scheduler, for instances like a thread finishing execution or signaling a blocking operation.
- Handling faults/exceptions. This is usually handled by the kernel in other operating systems, in seL4 we have to do it ourselves.
- Updating hardware feedback information.
- Entering a low power state.

We also went with a preemptive scheduling model, which means we require a reliable way to be interrupted regularly without kernel involvement.

The first two can be encapsulated into `next_thread` and `load_balance` functions, and be left for the user to be implemented using the invocations provided by the kernel, an example of which we present in section 5.3.

For communication between threads and the scheduler several approaches exist, with equal merit:

- IPC via the seL4 kernel
- Shared Memory with polling
- Shared Memory with `UMONITOR/UMWAIT`, combining communication with low-power state.
- Userspace Interrupts

Communication in our case would be notifying the scheduler that they wish to yield or have completed execution, but could be expanded to more complex cases, like signaling a blocking operation or dependence on other threads.

To allow threads to communicate their completion to the scheduler, we designed a `thread_add` function that creates a new thread and wraps it to send a notification to the scheduler once the function provided by the user returns.

On older hardware, one had to enter the kernel to enter a low-power state, as the instructions to do, such as `MONITOR/MWAIT` or `HLT`, are privileged. Thus, older

userspace scheduler had to rely on the kernel to enter low power states or resort to busy-waiting, which has an impact on the overall power and energy efficiency, which we measured in section 6.4.5.

To enter a low-power state on modern hardware, we take advantage of the `WAITPKG` extension, notably the `UMONITOR/UMWAIT` pair and `TPAUSE` instructions. `TPAUSE` allows us to enter a low-power state for a set number of TSC cycles, waking up early if an interrupt fired, while `UMWAIT` allows us to do the same, but also wake when the address specified in `UMONITOR` is written to. This makes `UMONITOR/UMWAIT` a good fit for combining with shared memory, as we can wake up when a thread completes and handle cleanup of the thread quickly.

4.6 Example Scheduler

To better elaborate our design, we present a few illustrations of common operations of a possible userspace scheduler with our design, seen in figs. 4.4 to 4.6. The illustrations depict interactions between the scheduler, the kernel and individual cores, which are under the control of userspace, depicted by an arrow from a ring buffer to a core. The dotted line represents the reservation mask, which in this case is every core except our scheduling core. To keep the illustration simple we reserved all cores, but a practical application may reserve a non-continuous number of cores. Arrows with numbers represent the order of operations described in the individual sections for each illustration.

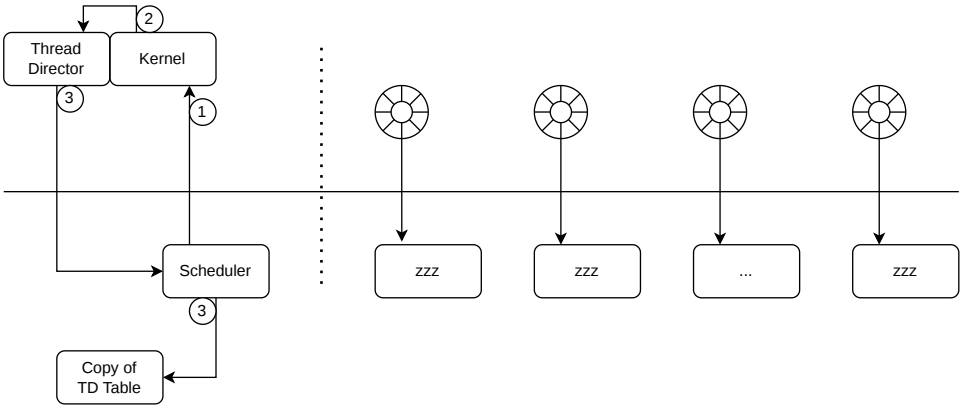

Figure 4.4: Example scheduler showing the initialization including the *ITD* capability

The first illustration, fig. 4.4, shows how initialization with our system works.

1. First, the kernel is invoked with the *ITD* capability to initialize Thread Director.
2. The kernel then checks if support for Thread Director is present, then enables it by writing the necessary information into the required MSRs.
3. Finally, a copy of the initial Thread Director table is copied to the userspace scheduler for load balancing, such that we need not enter the kernel for load balancing.

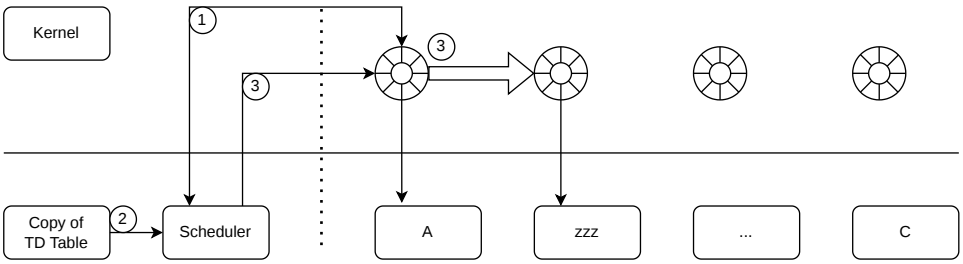

Figure 4.5: Example scheduler showing different possible scenarios

Next, we show how load balancing works, shown in fig. 4.5. In this case, we assume initialization has occurred, a few threads were assigned to some cores, and the scheduler has decided to perform load balancing after some unspecified time. In order to perform load balancing with Thread Director, the following steps need to be performed:

1. First, the class ID of the running thread is read out, which is then used to index into the Thread Director table.

2. A suitable core is selected, in this case core two, and the thread is migrated there.

3. Then, we first detach the thread from its previous core, causing the core to go idle, and attach it to core two, which was previously idle and now runs the thread from before.

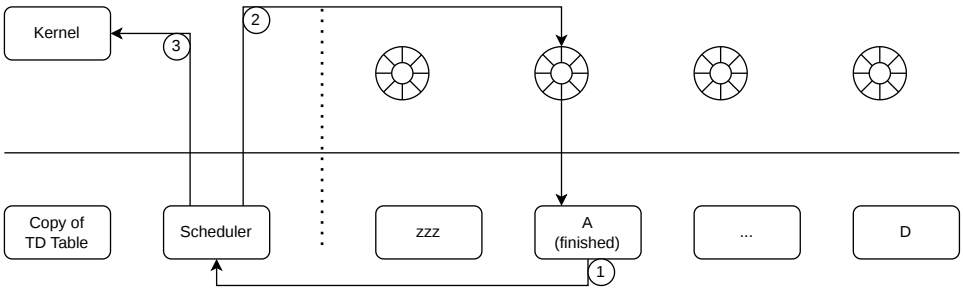

Figure 4.6: Example scheduler showing thread completion

As a final example, we show how threads can signal the scheduler in userspace, in this case for signaling completion, seen in fig. 4.6. Here we assume our scheduler has completed initialization, started a few threads and has put the core to sleep, waiting for the next time slice to end.

1. We begin by receiving a notification from a thread in userspace that it has completed execution and is waiting to be terminated.

2. This causes our scheduler to wake up early and detach the thread from its ring buffer, causing the core to switch to the next thread or go idle.

3. To clean up resources associated with the corresponding thread, the capability to the associated TCB is deleted.

4.7 Compromises

To wrap up our proposed design, we give a brief insight into the limitations and drawbacks of our design.

While, we were able to design a system that allows management of kernel threads in userspace, we still rely on the kernel for crucial features of our design, notably the reading and writing of MSRs, which as of writing, still have to be done in the kernel [8]. This would be a good starting point for future work, see 7.2.1 for what features a future implementation may integrate to further reduce kernel involvement.

Furthermore, to communicate with other cores, e.g., for scheduling and data collection, we rely heavily on IPIs, which also necessarily need to stay in the kernel for security reasons. This could be a reason for the lower than expected savings in power consumption we see in section 6.4.1, but considering that not many IPIs are otherwise sent, may make up a less significant part than we assume. Also, the creation and deletion of threads should definitely remain in the kernel, as we do want the kernel to be able to safeguard against scenarios threatening the entire system, such as arbitrary code execution or resource exhaustion. Additionally, the kernel should intervene when the userspace scheduler crashes or exits, by restoring control over cores reserved by the scheduler. Theoretically, this could be done by a watchdog thread that corrects anomalous behavior should an userspace scheduler become unresponsive, but as we are not experienced with an actual microkernel-based system and no complete system is publicly available, this may or may not be practically viable. In our current design, should another thread own an *ITD* capability, they would be able to restore these threads back to the kernel, after either migrating to another userspace scheduler or killing off the threads that used to be controlled by the userspace scheduler. A design in which the kernel itself resolves this condition is more difficult to accomplish in a microkernel-based system. Therefore, we delegate this task to future work.

In our design, we merely expanded upon the existing TCB structure, as it fit our needs, but another design making use of a reduced subset of the original TCB could possibly be feasible, reducing the need to interface with the kernel for this reason to a certain degree.

It should also go without saying that our design is Intel x86_64-centric, as we originally designed this scheduler with the intent of evaluating Thread Director and the `WAITPKG` extension, meaning we do not expect many of our presented concepts to carry over to AMD's x86_64 implementation or have a direct analog on other platforms supported by seL4, such as ARM or RISC-V. In the case of AMD we would need to fall back to the classless Hardware Feedback Interface for hardware feedback, as we are unaware of a Thread Director-like system for their platform, but would be able to retain efficient waiting in userspace via

the MONITORX/MWAITX instructions. As mentioned in section 2.4, the interfaces between the two pairs of instructions are not identical, so a fully x86_64 compatible implementation would need to check if it is running on an Intel or AMD processor and present the correct interfaces [11].

Additionally, we rely on very modern processor extensions and features for our design, meaning older hardware would benefit significantly less from such a design, as we laid out in section 4.5, and the amount of devices that do support these features, while growing with every year, is still very small. Furthermore, WAITPKG and Thread Director may not yet be fully ready for high-reliability platforms, either due to implementation defects or being a vector for a side-channel attack.

In the case of UMONITOR/UMWAIT, a defect on some Xeon processors was found where these instructions may affect sleep states or performance loss [12].

If not taken care of by a scheduler, Thread Director hints may be exploited by software running specific workloads. These workloads are then given known class IDs, leading to preferential treatment by the scheduler,which can be used as a side-channel attack, through which information may be gathered. The exact process is detailed by Chun et al. in [34].

[11] This would best be done by a library that wraps the raw assembly instructions.

Chapter 5

Implementation

This chapter details how we implemented the support for Thread Director and userspace scheduling in seL4. Development took place inside a virtual machine using KVM and QEMU, so we first needed to virtualize Thread Director and other processor features to make it accessible to seL4.

After this, we implement the necessary mechanisms to use Thread Director in seL4, first in the kernel, then adding interfaces to also utilize Thread Director in userspace.

With all pieces in place, we wrap up by adding support for and implementing a userspace scheduler in seL4. To make sure we can compare with an unmodified version, support for Thread Director and userspace scheduling can be disabled at runtime via an invocation or entirely via a build option.

Implementation was fully done in C, as this is the language in which all projects we need to modify are written, although one could likely implement the userspace scheduler in Rust, since the seL4 team has mature support for Rust in userspace via crates for rust support [75] and the Microkit [76] framework.

However, this was not pursued, but could be used in future work to ease implementation of more complex userspace schedulers, potentially mitigating memory bugs often associated with C.

We also chose to work with the non-Mixed Criticality System (MCS) version of the seL4 kernel, as we were uncertain of the stability of the MCS version in conjunction with simultaneous multiprocessing (SMP) [77–79].

5.1 Virtualizing Thread Director

This section details the initial work of adding support for Thread Director to KVM and QEMU, such that a virtual machine implementing support for Thread Director would be able to use it. In order to allow Thread Director to work, the following changes have to be made:

1. Setting the `CPUID` bits indicating support, alongside other bits for metadata related to Thread Director
2. Pass through or emulate the necessary MSRs:
    - Mandatory:
        - `IA32_HW_FEEDBACK_PTR`, memory address for hardware to write the table to.
        - `IA32_HW_FEEDBACK_CONFIG`, enables Thread Director at the hardware level
        - `IA32_HW_FEEDBACK_THREAD_CONFIG`, enable classification of threads, per core.
        - `IA32_HW_THREAD_FEEDBACK_CHAR`, reports the current class of the thread.
        - `IA32_PACKAGE_THERM_STATUS` (Bit 26), reports that the Thread Director table has been updated.
    - Optional:
        - IA32_HRESET_ENABLE, for resetting Thread Director Feedback.
        - IA32_PACKAGE_THERM_INTERRUPT (Bit 25), for sending an interrupt when the Thread Director table is updated.
3. Keep the Thread Director Table of the Guest and Host in sync (or pass a synthetic version of this table).

Thankfully, a patchset from Intel already existed for both KVM [80], which we assume to have been made for version 6.9, but is not explicitly stated in the patchset, and QEMU [81], which we ported to version 6.18 of the Linux kernel and installed on a machine running Ubuntu. The process was relatively straightforward, only being slowed down by functions that had since been renamed, changed behavior, or deleted [1]. After applying both patchsets, we were able to enable and configure the necessary MSRs in seL4, but notably did not receive

[1] For example `guest_cpuid_has` becoming `guest_cpu_cap_has` or `kvm_arch_sched_in` being deleted

the Thread Director Table from KVM. As our initial implementation used busy-waiting until the table was present, this caused the kernel to hang. This was partially caused by a misunderstanding of what the value of the address field of `IA_32_HW_FEEDBACK_PTR` should be. This MSR tells the hardware which page frame to write the *Hardware Feedback Table* into, and as it works with page frames, it expects the address to be shifted by the page size. While this now caused KVM to write to the correct address, we still only received a zero-page. This caused us to notice that, despite being set as a build option, the Thread Director table was not activated and being maintained in the host, which we confirmed by reading out `IA_32_HW_FEEDBACK_PTR` in the host, which is not clearly stated in the patch-set. The thread classification feature of Thread Director worked without issue, which made us assume that all of Thread Director was functional correctly, when it actually was not enabled.

5.1.1 Conflicting Patchsets

Around the same time, another patchset had been submitted to the kernel mailing list [82], which caused the Hardware Feedback Interface, and by extension Thread Director, to only be activated when userspace requests it, which the patchset we were trying to port conflicts with. This patchset had since been merged into the kernel, while the patchset we were trying to port is not. While infuriating to discover, it was simple enough to revert the offending commit and observe that Thread Director was now fully passed through to the guest and to our seL4 kernel.

5.1.2 Miscellaneous Patches

While the aforementioned patches added what we are trying to examine, we had to apply additional patches for our purposes. The first is a simple patch that adds the ability to QEMU to pin a virtual CPU (vCPU) to a logical core. Without this addition our vCPUs can land on arbitrary cores, which no longer correspond to the cores of the Thread Director Table, leading to bogus results. Complex remapping is not a viable option.

For benchmarking we applied another patchset to QEMU and KVM. As we try to evaluate the possible gains in energy efficiency from Thread Director, we need the ability to obtain energy related data from hardware. To this end, we disabled interception of `IA32_MPERF/APERF,PPERF` and RAPL MSRs `MSR_RAPL_POWER_UNIT`, `MSR_(PKG/PP0/PP1)_ENERGY_STATUS` [2]. For `APERF/MPERF`, we followed the "intended"

[2]Note, we do not require a full virtualization of these MSRs, instead we want to measure the values including the time and energy spent in the host environment.

method of disabling interception via `KVM_CAP_X86_DISABLE_EXITS` and setting the related CPUID bit for the virtual machine. Doing this requires that vCPUs are pinned to their corresponding physical CPUs, which we guarantee with the previous patch to QEMU.

For the remaining MSRs we unconditionally disable interception in KVM via `vmx_disable_intercept_for_msr`, accepting that this is not the correct approach, but the one that is most convenient.

We are aware that, in theory, there already exists a way for RAPL MSRs to be passed to QEMU [83]. However, we chose not to pursue this method because of the limited RAPL domains exposed and questionable quality of the implementation [84] affecting our measurement.

5.2 Implementation in Kernel

With all necessary features for Thread Director accessible in seL4, we continued our implementation in seL4. We initially augmented the kernel scheduler to support migration to other threads to give us a first impression for what is needed to support Thread Director. We also were not completely familiar with extending kernel functionality via capabilities at this point. Therefore, we saw this as an easier alternative at the time. Additionally, this would ease development of the userspace scheduler prototype as we could carry over most functionality into userspace.

5.2.1 Scheduling In seL4

The seL4 scheduler is a rather simple Round Robin (RR) scheduler, which additionally supports defining time slices in the MCS configuration in the kernel with scheduling context and domain scheduling, but we restricted our view of the scheduler to exclude these two components. Relevant for us here is the `schedule` function, which is the main entry point from a timer interrupt into the kernel scheduler. After handling various checks and early exits, the scheduler chooses a new thread to run via `chooseThread`, which chooses the highest priority thread in a ready queue on each core. After choosing a thread, the thread is switched to via `switchToThread`, which handles architecture-specific methods regarding thread switches, then sets the thread as the current thread. Notably, at no point does the kernel relocate threads between cores, but allows us direct access to the ready queues of every core. After examining how the affinity of a thread is changed via the corresponding invocation, we found it rather trivial to add this support to the scheduler. For our kernel implementation then, we added a function, `chooseCore`, which is called by `chooseThread` after all checks have passed

and a thread has been chosen. There, we select a core to move our thread to check if migration needs to happen, then migrate if this is the case, otherwise we exit and call `switchToThread` as before. We split off actual core selection into an architecture-specific function `Arch_chooseThread`, under the assumption that should this feature actually be desirable in the kernel, and another platform in the future like RISC-V or ARM adds a similar hardware feedback interface, then integration into our additions would be made much simpler and cleaner. In our case, as there were no platforms other than x86_64, all other functions return an invalid core ID to default to the normal kernel behavior. For x86_64, we implemented the core selection like it is recommended in the Intel manual by first obtaining the class ID of the running thread, then selecting the core with the highest value [3] that is available [8]. As we later found out in our userspace implementation, these values cannot be blindly trusted to sort out core assignment.

It was here that we also encountered the issue of $\frac{3}{4}$ of the Thread Director table being empty. Under guidance from the Intel manual [8], we constructed the table as advised, including adding a padding of 6 bytes for each block of reported values, which are reserved for possible future values that can be reported by Thread Director. We do not know if this is the fault of the Linux patchset or false information in the Intel manual [8], but the added padding caused most values to be hidden inside the padding. Removing the padding correctly gave us the entire Thread Director table in the guest as reported in the host.

While the manual specifies behavior for when cores are free [8], there exists no further guidance for when all cores are occupied. Therefore, the provided values need to be weighed and adjusted based on the load of the core they are currently assigned to, ensuring load is distributed according to highest value and avoiding oscillations between groups of equal value. We are by no means experts in scheduler design. Therefore, a production version would need the most adjustment in this area to be worthy of consideration.

When we migrate a thread, we follow the implementation as it exists already in the `TCBSetAffinity` invocation, which sets the affinity of a thread. Copying the behavior of the function successfully moves our thread from one core to the other without issues, right up until vector instructions were attempted to be used, the thread would fault. This was partially caused by the complexity of the seL4 build system, as properly configuring seL4 at build time is not trivial. SSE and AVX instructions require the kernel to setup `XSAVE` and `XRESTOR` before being usable by userspace. A simple configuration change allowed us to use these instructions. However, they still faulted at inconsistent intervals.

[3] The Intel Manual refers to this value as “capability” [8], but to avoid confusion with seL4’s capabilities, we avoid using this term

5.2.2 Management Of The FPU

The Floating Point Unit (FPU) handles, unsurprisingly, floating point operations. Originally a separate chip on the motherboard of an x86 processor, it is now an integral component integrated into every core for not just floating point operations, but also vector instructions like SSE and AVX. seL4, being a system that can be deployed on many devices, has configurable support for the FPU at build time and does not make use of vector instructions in the kernel to keep the design and implementation simple. At any point in time, a thread "owns" the FPU on a core, and the kernel has to manage and reassign the owner regularly to ensure it functions correctly, including when threads switch between cores. This also involves saving and restoring the state of the FPU once it switches over, which can be costly. seL4 currently implements a "semi-lazy" approach, where a thread can configure whether or not it wants to use the FPU or not. The kernel then always saves the FPU state when switching away from and restores the FPU state when switching to an FPU-enabled thread [85]. If all threads make use of the FPU, then the FPU state is always saved and restored. We are unsure how this impacts our results given the unreliable nature of the provided benchmarking results [85], but the difference should be minor at best.

For unknown reasons, to fix our faulting thread problem from before, we have to call the `lazyFPURestore` function after moving our chosen thread to properly handle the transfer of ownership of the FPU, despite all our threads being configured to always use the FPU. With this, we had a functional prototype that could relocate threads based on hardware feedback.

5.2.3 Other Kernel Additions

This section details smaller necessary changes we made to enable userspace scheduling on seL4. The first is the addition of an affinity mask, `tcbAffinityMask`, for threads and an invocation to set it. We implemented this mostly for the sake of completeness, as we were convinced that, now that the scheduler can move threads arbitrarily across cores, some threads may wish to restrict the cores that they can be placed on. Furthermore, with this being a common feature found in every kernel that handles multiple cores, it felt like a natural addition to the system. With two methods to restrict where threads can be placed, we now also had to ensure that the reservation mask [4] and the affinity mask could work in conjunction with each other. When moving threads with a set affinity and reservation mask, the scheduler can only place threads where both allow a thread to be. Should this lead to no core being allowed to be used, the scheduler falls back

[4]This is also the reason we did not refer to the reservation mask as an affinity mask, despite serving a very similar purpose

to the first available core, which by our design, is guaranteed to exist. If a reservation mask is already set and a thread tries to set an affinity mask overlapping with the reservation, this affinity mask is rejected by the kernel, but the other way around is harder to check, as we would need to check the affinity mask of every thread for possible overlaps with the new reservation mask.

This is not the only way to handle this situation and other solutions are viable, but may be more invasive, like blocking execution until the core reservation is cleared or terminating the thread outright.

Another small change we made is to allow the use of the `IA32_TSC_AUX` MSR. This MSR is unused by seL4 on x86 and can contain arbitrary data, which can be retrieved by userspace via the `RDTSCP` instruction. In other kernels, like Linux, this is used to store the ID of the core a thread is running on [86], which we also proceeded to do by writing the current core's ID into the MSR on boot. This gives userspace a low-overhead method of knowing which core they are running on, and was used extensively during development.

The last addition was simple support for Arrow Lake in seL4. Some code in the kernel checks for the type of processor seL4 is running on, e.g., a runtime check if seL4 has been compiled for the correct microarchitecture, disabling of prefetchers, and initialization of the TSC. This is done via a long list of defines for the model ID and arrays containing the list of valid models, ranging from the Nehalem to the Skylake-X microarchitecture [87]. While AMD is also listed as a known x86 vendor, we did not find any model IDs for AMD microarchitectures. We assume, as development in seL4 continues, for this to be handled via a better system, but we did not concern ourselves further with this matter.

5.2.4 Missing Kernel Features

One of the features lacking from our implementation, which a production version would require, is the ability to evict cores to other cores for reservation. When a core is reserved to be managed by userspace, any threads currently scheduled to run on that core need to be moved away, in order to not interfere with userspace. The same would need to happen when clearing the reservation, but for userspace threads. For our prototype, as we do not run other programs beforehand or change the reservation mask, we implemented a function, `relocate_threads`, but did not test it thoroughly, as we do not need it for our evaluation. To avoid unexplainable results stemming from a function we currently do not need, we temporarily removed it, where the function would normally be used.

Another feature would be to disable the timer interrupts. Currently, if we receive a timer interrupt, it is still processed by the kernel but is simply acknowledged instead of ticking down the kernel scheduler. Ideally, when reserving a core for userspace, we would disable the timer interrupt completely, by writing

to the appropriate APIC [5] MSRs on every core. Then, when clearing a reservation and returning a core to the kernel, we then enable the periodic timer interrupt again by writing to the necessary MSRs [6]

5.3 Userspace Scheduling With Thread Director

With all pieces in place, we have finished our implementation by making our own userspace scheduler. For reasons listed in section 4.5 we went with a centralized scheduler started on the root task. We also went with a ticked scheduler, meaning the scheduler wakes up at a fixed, regular interval to perform scheduling duty, before going back to sleeping. For our implementation, we went with a 20 ms tick length, as this seems to be a reasonable length that is neither too long or short. This is also the time slice of threads attached to our prototype scheduler. A time slice that is too short risks stalling a core with too many context switches, while a time slice that is too long causes programs to perform sluggishly. Theoretically, one could also implement a variable-length time slice similar to Linux' Completely Fair Scheduler (CFS) [89], as sleeping is done fully in userspace. Therefore, how long one sleeps is up to the scheduler. The default setting for seL4 is 2 ms tick length, with a thread being preempted after 5 ticks, leading to a 10 ms time slice, which gave us the idea of how long a timeslice should be.

Furthermore, we decided on performing load balancing every ten scheduler ticks and updating the Thread Director table every 25 ticks, which correspond to every 200 ms and 500 ms, respectively. The value for load balancing was chosen based on how frequently Linux performed Load Balancing [90] and the value for Thread Director updated is a rough guess based on observations from Chun et al. [34] on how often Thread Director updates the table.

While on the topic of numbers, we chose a capacity of 64 threads per core as this should still allow many threads to be scheduled simultaneously while still leaving our scheduling state at a reasonable size. The limitation is rather arbitrary and depends on the size of the expected workload to be run on a machine. This value has to be set when building the kernel, as the underlying data structure is statically allocated, because the kernel has no access to memory after booting that is not specifically given to it via capabilities. The code is written to take arbitrary capacity values, but very high values may cause the kernel to waste significant amounts of space on unused capacity. seL4 technically has no

[5] Advanced Programmable Interrupt Controller, a programmable circuit on every core for managing interrupts

[6] The MCS version of the kernel would make this process easier, as the kernel is tickless, and take advantage of optimizations like TSC Deadline mode [8, 88]

limit on how many threads can be on one core, as they construct a linked list with the TCB objects. We leave open the discussion on the practical implications of many threads on one core, but admit to this being one of the downsides of our design.

For entering a low-power state during downtime, we use the instruction pair `UMONITOR/UMWAIT` monitoring an address visible for all threads, so that we can both make use of sleeping in userspace and wake up early if necessary. There also exist other ways of reducing the power consumption when idle, which we later evaluate, see section 6.4.5, and found that `TPAUSE` would work equally well here.

Our scheduler expects threads to be created via a special `thread_add` function that handles creation of the thread alongside its stack and IPC buffer. Both stack and IPC buffer are a single page, created via the `seL4_X86_4K` capability, with no support for resizing. Additionally, both C- and V-Space of threads are shared with the main scheduler. These two issues are not a concern for our prototype, but separate spaces and larger stacks would be better suited in a production environment. After mapping stack and IPC buffer, the registers of the thread are configured with the necessary values, then started. Instead of starting the thread directly, we wrap the function around our standard thread block, such that we can notify the scheduler once we finish execution without the function itself needing to do that.

For the `load_balance` function we copied over the function we used to determine the best core to place our thread on and adjusted it in two major ways:

- Add a penalty to threads with higher load, to avoid all threads aggregating on one core. For this we deducted a flat value scaled by the number of threads on a core from the reported values of a core during the selection of a best core.

- Add a threshold for thread migration, to avoid oscillating between equal cores. In order to allow a thread to migrate, the proposed best core would need to be higher than the current core plus the threshold.

Both the penalty and the threshold were chosen arbitrarily, *40* for the penalty and *10* for the threshold, as both parameters would need extensive trial and error to set right. Why this was necessary we discuss later in section 6.4.2. During evaluation, this caused suspiciously high power consumption for some schedulers as they decided to schedule on more cores than others, which would have been different given a higher threshold, visible in the graphs discussed in section 6.4.1.

For our actual scheduling policy, which in our design can be found in the `next_thread` function, we implemented a simple RR scheduler to have a similarity to the original scheduler found in the kernel. As we chose to implement

an RR scheduler, we expect more complex schedulers to outperform our naive implementation.

For the sake of completeness, we also made an initialization function that sets up the state of the scheduler and maps intermediate mapping structures, as seL4 does not do that for us [23], and we depend on them to create new threads. This is likely necessary for any scheduler to have and aside from differences in scheduler state are identical across schedulers.

5.4 Problems Of The Modern World

We mostly avoided talking about problems encountered, as the majority of them were self-inflicted, but some came as a result of the tooling we employed not providing basic features which we sorely needed.

We had to patch our software several times to pass through support for what we needed, which, while expected, has taken up a good chunk of development time that could have been deployed elsewhere. Adding pinning support to QEMU was relatively trivial, passing through RAPL data took several days, tracking down why Thread Director is only semi-active took a week. We expected some features that have been around in some form for over a decade to be supported, like in the case of RAPL, only to find bare-bones support that did not work on our machine. Ultimately, we are trying to observe features that are bleeding-edge with software that does not keep up.

5.5 Summary

We do not claim this to be a definitive way of how a scheduler should be built on top of our seL4 kernel changes, but wish to show the possibilities that this approach provides, should one wish to take advantage of it. Other scheduler designs should be able to be built on top of our changes and work with little interference from the kernel. For our evaluation, see section 6.3.1, we built two other schedulers, which are derivatives of our original scheduler, where one is unaware of heterogeneous multiprocessing (HMP) and assigns cores from first to last, and the other that is HMP-aware and prioritizes scheduling on E-Cores for efficiency. We welcome future work 7.2 to attempt to port more complex schedulers like CFS to seL4 using our work as a basis and evaluate how the two versions hold up in comparable workloads.

In summary, we were able to implement the majority of our design in seL4 with little issue once we understood how the system functions, which took up the majority of our time and effort. Differences between our prototype and a

production-ready, final version should mostly concentrate around the features we failed to implement. This will likely lead to more invocations than are currently present, to handle cases like thread eviction, and likely more overhead, as simply more things have to be done. However, we expect no significant changes to be needed between what we already implemented and what winds up in a final version.

Now that we have worked with the seL4 system, we see the value in its approach, but have to acknowledge that a monolithic kernel provides many conveniences normally taken for granted that seL4 does not provide. A simpler system does not translate into a simpler environment to work in, as is apparent in our rather bare-bones scheduler, but despite this we succeeded in creating a userspace scheduler in seL4 that also makes use of hardware assistance.

Chapter 6

Evaluation

In this chapter, we evaluate our implemented userspace scheduler and determine whether Thread Director provides a meaningful uplift in energy efficiency, and speculate, if a similar mechanism could also work with our scheduler abstractions. First, we describe our evaluation platform and what data we wish to collect, followed by a small chapter on evaluation on microkernels. Following this, we explain how we implemented our benchmarks, wrapping up with a discussion on the results of these benchmarks.

6.1 Methodology

The table in fig. 6.1 details the specifications of our evaluation platform. Note that we left both *SpeedStep* and *Turboboost* enabled for our evaluation runs. This is because we plan on limiting the power budget during some of the benchmark runs to evaluate Thread Director in a power-constrained system. Furthermore, this is the typical configuration on modern, energy efficient systems, e.g., on laptops, and more data centers also tend to leave both enabled.

Hyperthreading is not listed in this table, as our Arrow Lake system is the first series of newer Intel processors that ship with Hyperthreading removed from both P- and E-Cores [91], meaning logical cores always equal physical cores.

CPU	Intel Core Ultra 7 265K
Cores	20 (8 P-Core + 12 E-Core)
Base Frequency Boost Frequency	3.9/3.3 GHz (P-Core/E-Core) 5.4/4.6 GHz (P-Core/E-Core)
L0 (P-Core)	48KB
L1	64KB Instruction + 192/32 Data (P-Core/E-Core)
L2	3/4MB (per core P-Core, per cluster E-Core)
Total L2	36MB
L3	3MB (per core P-Core, per cluster E-Core)
TDP	125/250W
DRAM	2 x 16GB DDR5-4800
Motherboard	ASRock Z890 Pro-A
CPU Features	
Speedstep Turboboost	enabled enabled
seL4 Configuration Changes	
Meltdown Mitigations	disabled
Linux	Ubuntu 24.04
Kernel	Version 6.18 with patches
Hypervisor	KVM with ITD+RAPL patches QEMU with ITD+Pinning+RAPL patches
Guest OS	seL4 v14.0.0 with patches
#VCPUs	20, pinned to physical cores
Assigned Memory	2GB

Figure 6.1: Description of evaluation system

For our benchmarks, we measured the following values:

- IA32_APERF, IA32_MPERF and MSR_PPERF for every active core
- RAPL data for the PP0 and PKG domain. We could also record the PP1 domain. However, we do not and cannot utilize the integrated graphics chip in our processor in seL4, meaning we would not get any meaningful results out of it.
- Current package temperature, obtained by reading out the maximum temperature TJMax in the MSR_TEMPERATURE_TARGET MSR and subtracting the distance from TJMax, found in bits 22:16 of the IA32_PACKAGE_THERM_STATUS MSR. [1]

As reading out most MSRs we need for benchmarking require privileged instructions [2], we added another invocation to the *ITD* capability, X86ITDGetPerfData, which collects the necessary data and returns it to userspace. This is currently a workaround and should, in this version, obviously not stay in a final implementation, but for benchmarking our prototype we considered this sufficient. However, for schedulers that use a user-defined model, this functionality might be helpful.

With IA32_APERF, IA32_MPERF and MSR_PPERF only their deltas have significance, meaning the difference between their current and present value, therefore we compute these in the kernel and return only the deltas to userspace. As a reminder, the PERF MSRs are counters that increment at different frequencies: APERF, at the current clock speed, MPERF at the reference frequency, and PPERF when the current cycle "contributed to instruction execution" [8].

From the PERF MSRs we derive the effective frequency e_{eff} and productivity p, which is a measure of the scalability of a workload, as follows:

$$e_{eff} = \frac{\Delta IA32\_APERF}{\Delta IA32\_MPERF} * TSC\_FREQ$$

$$p = \frac{\Delta IA32\_PPERF}{\Delta IA32\_APERF}$$

[1] We had to change this during evaluation, as it interfered with the kernel patchset. Instead of reading the value directly, we emulated it by reading out the individual IA32_THERM_STATUS MSRs and selecting the maximum value

[2] PMC MSRs can be read out from userspace via RDPMC, no need for a system call and thus kernel entry and exit.

We considered collecting data from the three fixed performance monitoring counters (PMCs) present on our machine, however, prior work [92] did not give us the confidence that these would carry much significance in our work, opting not to collect them.

What we would have considered, if we had more time, was to construct a Machine Learning (ML) prediction model trained on PMC data, which we could then compare against the hints provided by Thread Director. Related work by Saez and Prieto-Matias have shown that such a model is feasible, but would outperform Thread Director on Linux [48], which we would be interesting to compare to a microkernel-based system. For this case a collection with `X86ITDGetPerfData` might still be valuable in a production system.

Reading out the desired values happened on every scheduling tick, meaning every 20 milliseconds, see section 5.3, which strikes a compromise between the amount of data collected and interruptions caused by data collection. This also aligns with the time slice we assign to every thread, meaning, we measure the data at the end of every time slice. In effect, we run our usual scheduling tasks, then at the end before going to sleep again, we measure the data, allowing our interruptions to be as close to other interrupts as possible and letting the program run uninterrupted for the assigned duration.

Several steps were taken to reduce noise in our readings as much as possible. For one, our test bench is not running a graphical session, minimizing noise from the integrated graphics card. Furthermore, no other programs are running on our test bench other than part of our integrated development environment (IDE) and background services, and no two benchmarks run concurrently, meaning each benchmark is run on a system that can be considered “effectively idle”.

6.2 Benchmarking on Microkernels

Originally, we intended to evaluate larger, more “real world” workloads, as those would be more interesting and offer more “hard data”, allowing for more definitive conclusions to be drawn. However, we very quickly realized that this is not possible, as real programs require functionality that the seL4 microkernel environment does currently not provide. A non-exhaustive list of features found in real-world software includes:

- File systems
- Drivers
- User Input
- …

Any single point would consume more time than the project we were originally going to implement. While operating systems built on seL4 exist, for example the in-house LionsOS [93], they would still prove to be more of a hindrance than benefit, since now one has to learn both how seL4 and the OS operate to add features to seL4, which would fall well outside the scope of a bachelor's thesis.

For this evaluation, we ported the `stress-cpu` series of benchmarks from the stress-ng [94] benchmark suite to seL4. The process of porting a single file from an already very portable codebase ended up taking an entire day's worth of effort, which could only be completed by stripping down the file to only the individual benchmark functions. A pseudo-random number generator used by some programs, as they still relied on many commonly found utilities that seL4 does not provide, like user input, reliable wall-clock time, various system statistics for entropy, and a more feature-rich C library [3].

Additionally, we initially considered running seL4 on bare metal to obtain less noisy results, by virtue of the seL4 kernel doing less but quickly dropped the idea, as we once again ran into the issue of a lack of driver support. While less noisy results are nice to have, it would be better if we were able to obtain our results in the first place. However, we pin each vCPU to a dedicated core to minimize migration overhead in the host. Take for example the work of Gurres, who performed his evaluation on real hardware, but had to resort to excruciatingly slow serial-over-LAN, because no driver existed for their storage or network devices for seL4, drastically reducing the amount of data collected [92].

For a more extreme example, recently Furgala et al. published a paper detailing the port of NASA's core Flight System (cFS) to seL4, which took a team of three developers a year and a half to complete [95]. Ultimately, they had to develop an entire operating system on top of seL4 for running a single, admittedly complex, program, explaining the long development time. With this, we want to illustrate that, while adding new features to seL4 can be done in a relatively easy manner, building on top of it can quickly balloon in complexity and that one may take for granted how many conveniences a monolithic kernel like Linux provides. Being small does not mean being simple, is what we wish to convey and why the selection of applications we measured is on the simpler side.

[3] Instead of the more commonly found `libc`, seL4 relies on a modified version of the `musl` `https://musl.libc.org/` library, where most function that require system calls need to be implemented in userspace

6.3 Benchmarking

As mentioned in section 6.2, we are evaluating on a microkernel system. Therefore, we had to write our own benchmarking suite and find a way to extract our data.

For our benchmarking suite, we run all initialization and data collection on the root task [4], which is performed on core zero. This involves initializing the necessary capabilities, initializing the userspace scheduler, spawning threads, and entering the main scheduling loop.

When starting the QEMU child process, which runs our modified version of seL4, we monitor its standard output, waiting for a magic string, used to know when it can be terminated in order to start the next benchmark run. This string is printed by the root task after leaving the main scheduling loop, signaling that the current iteration has finished execution.

For our case, we do not concern ourselves with cleanup after execution as the virtual machine we are testing in will be restarted regardless. This can be done without issue, as the state of the operating system does not persist between reboots, meaning every reboot returns us back to a clean state.

Threads are initially placed on core one, begin execution immediately and are moved by the scheduler in the next tick. The first measurements begin after the first tick of the scheduler, which happens every 20 milliseconds.

We went with a virtual machine for several reasons: it avoids some of the problems mentioned in section 6.2 and allows for easier data extraction by writing console output directly to a file.

Furthermore, we think our implementation could be most useful in a data center environment, where virtualized operating systems and hypervisors are common and concerns with energy efficiency and high reliability are more pronounced than in a consumer environment. We examine Thread Director as an example of hardware assistance in operating system tasks, in this case scheduling, to see the possible benefits this approach brings, e.g., for a hypothetical future server CPU that also supports these features.

Data extraction was possible to do cleanly in a virtualized environment with no data loss or slowdowns by utilizing the debug console, or `debugcon`, found in QEMU alongside the `IOPortControl` and `IOPort` capabilities in seL4.

[4] The initial thread on seL4

For this, we first create the `IOPortControl` capability, through which we can invoke the creation of the `IOPort` capability, which we can then invoke to read from and write to a range of I/O ports we specified when creating the capability. For this to be actually useful, we need to know the port number of the device, we want to interface with, which is `0xE9` for the QEMU debug console. Writing to a file is as simple as instructing QEMU to redirect output on the debug console to a file.

6.3.1 Benchmark Design

For our workloads, we constructed two main types of programs, with one type having two variants.

- A simulated compute-bound application continuously running the `stress-cpu` suite of functions [5] from stress-ng [94]
- A simulated hybrid application using the Likwid [6] suite of microbenchmarks, coupled with periodic sleeping via `TPAUSE`.

For the hybrid benchmark, we define a static array, 16MB in size [7], for these threads to utilize and fill it with pseudo-random data, generated via the `rand` function provided by `musl`. The actual workload is a combination of the `load`, `peakflops` and `update` benchmarks to simulate receiving, processing, and storing of large files. After the three operations complete, we sleep for 50 milliseconds, leading to a semi-dynamic program that should alternate between class *three* and *one/two*. For these, we also want to compare two versions: one utilizing SSE instructions and the other AVX instructions, to test workloads corresponding to class *one* and *two* of Thread Director.

As our chosen programs for benchmarking do not have a "fixed" time when they terminate, we settled on an upper limit of five minutes per iteration, opting to perform six iterations per benchmark, with the first one being discarded.

We also want to evaluate how Thread Director performs in a power constricted environment, as the Intel manual suggests dynamic changes in Thread Director to occur in such a case [8].

Therefore, we also tested our workloads in two power-limited scenarios:

- Unrestricted, where we did not change the power limit, which on our machine is 200W for PL1 and 250W for PL2, see section 2.6.1 for a reminder on PL1 and PL2.

[5] With the exception of `stress_cpu_ackermann`, as this workload requires a larger stack size than we provide.

[6] `https://github.com/RRZE-HPC/likwid`

[7] Trying to define larger arrays led to boot failures.

- 30W, which represents an average laptop and should induce Thread Director to exhibit dynamic behavior. Originally, we wanted to evaluate even lower power limits. However, we were unable to go below 24 Watt, as this is the lower power limit defined in the `MSR_PKG_POWER_INFO` MSR on our machine. It is theoretically possible to restrict power draw further by setting the `MSR_VR_CURRENT_CONFIG` MSR, which controls the so-called PL4 power limit that controls the absolute maximum power the processor can draw.

For this reason, we measure the current frequency of the processor: if Thread Director does successfully hint where to place threads to maximize performance, we would be able to observe the average frequency to be higher than on a system without hints that has to throttle down further to stay inside the assigned power budget. Originally, we also planned to test an "in-between", but due to time constraints we were unable to do so.

On Linux, one can change the power limit by writing the maximum allowed wattage to the `sysfs` [8] entry:

`/sys/class/powercap/intel-rapl:0/constraint_(0/1)_power_limit_uw`,

where 0 and 1 correspond to PL1 and PL2, and wattage is measured in uW [96]. The entered wattage is converted to the appropriate unit for the hardware, and then written to the `MSR_PKG_POWER_LIMIT` MSR.

For this comparison to work, we also need another scheduler to compare against, for this we came up with three scenarios:

- Our current scheduler, heterogeneous-aware and dynamic, scheduling with Thread Director hints.
- A heterogeneous-processor aware scheduler that statically assigns threads first to E-Cores then to P-Cores, but does not know which cores are ideal.
- A heterogeneous-processor unaware scheduler that statically assigns threads from first to last with no regard to capability of each core. This also represents our stand-in for the initial kernel scheduler, as the kernel scheduler does not move threads between cores at all.

We do not compare with the in-kernel scheduler, instead opting for a stand-in, as the in-kernel scheduler is not aware of SMP and especially heterogeneous multiprocessing (HMP), relying on userspace to move threads between cores.

We implemented the other two schedulers by providing the current userspace scheduler with a synthetic Thread Director table that forces the specified behavior to occur, depicted in fig. 6.2.

[8] An interface on Linux for exposing kernel objects to userspace `https://docs.kernel.org/filesystems/sysfs.html`

Core ID	Class 3		Class 2		Class 1		Class 0	
	Perf	EE	Perf	EE	Perf	EE	Perf	EE
0	200	200	200	200	200	200	200	200
1	190	190	190	190	190	46	190	190
2	180	180	180	180	180	180	180	180
...	...	...	...	...	...	...	...	...
19	10	10	10	10	10	10	10	10

Figure 6.2: Synthetic Thread Director table with fixed increments, depicting the non-HMP aware case, meaning we choose cores sequentially from 0 to 19.

As a final variation, we also observed each workload under four conditions:

- No running threads. This is to establish a baseline and determine the overhead of our userspace scheduler. For this we did not bother to run them with power restrictions, as power draw for a single sleeping processor does not exceed the power limits we set.
- A single-threaded application, mostly to see if Thread Director actually chooses the most efficient core.
- Four threads running simultaneously, representing a light load on the system.
- Twelve threads, representing a heavier load. Again, we test Thread Director, choosing the most efficient cores in descending order. Increasing the number further decreases the significance of Thread Director's hints, as less cores become available to place threads on.

To make it easier to know which exact permutation we are talking about when discussing the results, we came up with abbreviations to identify each workload. A permutation follows the pattern `XW_XT_X_S`, where `XW` is the power limit set for the benchmark, `XT` is the number of threads launched for a given workload, the `X` being the type of workload, with **C** being for the stress-ng workload, and **HS** and **HA** being the hybrid workload with SSE and AVX instructions, respectively. The `S` is for the scheduler used, where **TD** is the scheduler using Thread Director, **SH** for the *static heterogeneous* scheduler, and **SN** for the *static non-heterogeneous* scheduler.

We also provide a table in fig. 6.3 as a more concise summary of what we described previously, as we noticed that it may be hard to to keep track of all possible combinations.

Variable	Abbreviation	Meaning
Wattage	200W	Unrestricted Power
	30W	PL1&PL2 set to 30W
Load	1T	One Thread
	4T	Four Threads
	12T	Twelve Threads
Workload Type	C	Compute with stress-ng
	HA	Hybrid with Likwid, AVX instructions
	HS	Hybrid with Likwid, SSE instructions
Scheduler Type	TD	Scheduler with Thread Director hints
	SA	Scheduler with static assignment, heterogeneous
	SN	Scheduler with static assignment, non-heterogeneous

Figure 6.3: Summary of variables examined, their abbreviations and what they refer to.

6.3.2 Miscellaneous Benchmarks

Beyond energy efficiency, we also want to benchmark other aspects adjacent our userspace scheduler. For this, we also measured how long Thread Director takes to change the class ID of a workload on our Arrow Lake system, by repeatedly running a synthetic workload with a known class ID and checking against what is reported in the `IA32_THREAD_FEEDBACK_MSR`. For accurate results, we ran 262,144 iterations per class, resetting the reported class every time with the `HRESET` instruction.

Furthermore, we wanted to analyze the overhead of our implementation by measuring the amount of cycles spent on operations that need to be performed in the kernel. To be more specific, the overhead of operations which we expect to be possible in the future in userspace, such as reading from and writing to MSRs and the IPIs needed for cross-core interrupts. For these, we ran 1,048,576 iterations, measuring the total time spent and the time spent on the actual operation. For the `RDTSC` test we read the class ID from the `IA32_THREAD_FEEDBACK_CHAR` MSR, and for the `WRMSR` test we wrote the ID of the current core to the `MSR_TSC_AUX` MSR, as there are not many other MSRs that can be arbitrarily written to and seL4 does not use this MSR for anything. The code for these test can be seen in figs. 1 to 5 for the classification and figs. 6 and 7 for the MSR latency.

As a final test, we also measured the power draw for different wait primitives in our scheduler, that being `UMONITOR/UMWAIT`, `TPAUSE` and a busy-wait loop with and without `PAUSE`. We did not have an AMD system ready to also test `MONITORX/MWAITX`, because adapting our code to also accommodate AMD processors would fall outside our scope, but we assume the results to be similar to `UMONITOR/UMWAIT`. For this, we ran the scheduler as usual, but without threads to just measure the power draw of the scheduling core.

6.4 Results

6.4.1 Energy Efficiency

The energy efficiency results unfortunately only offer few insights, as can be seen in figs. 6.4 and 6.5, showing that while Thread Director does reduce power consumption in most cases, this reduction overwhelmingly comes from the fact that Thread Director assigns our threads to E-cores for energy efficiency and leaves them there for the remainder of the benchmark. This leads to very static scheduling, similar to our other schedulers where we enforce such behavior, while we expected Thread Director to display more dynamic behavior. Comparing to our static heterogeneous scheduler, which simulates a hypothetical scheduler aware of hybrid architecture, but not ideal core placement, show differences within the margin of error when compared to Thread Director, suggesting that the ideal core placement of Thread Director may not be as relevant as it is commonly assumed. This can best be seen in the `1T` benchmarks, referred to in the graphs as “single”, see figs. 6.4 and 6.5. There is only one instance where we see Thread Director outperform expectations, that being the `200W_12T_HS_X` benchmark, showing that for some specific programs Thread Director may provide some slight benefits. These mixed results are not isolated to our benchmarking, as similar mixed-to-negative results are also reported in related work [48, 50], showing that the behavior observed is independent of both the complexity of the implemented scheduler or the specific implementation of Thread Director in any operating system, and can be reduced down to the hints provided by the hardware.

The prioritization of E-Cores for efficiency may sometimes lead Thread Director to perform worse than the non-heterogeneous scheduler, as can be seen in the `200W_4T_HS_X` and the `30W_XT_HA_X` workloads depicted in fig. 6.5. In this workload specifically, we saw Thread Director set values such that in some cases scheduling on P-Cores was preferred, see figs. 29 and 30, but only for Class two.

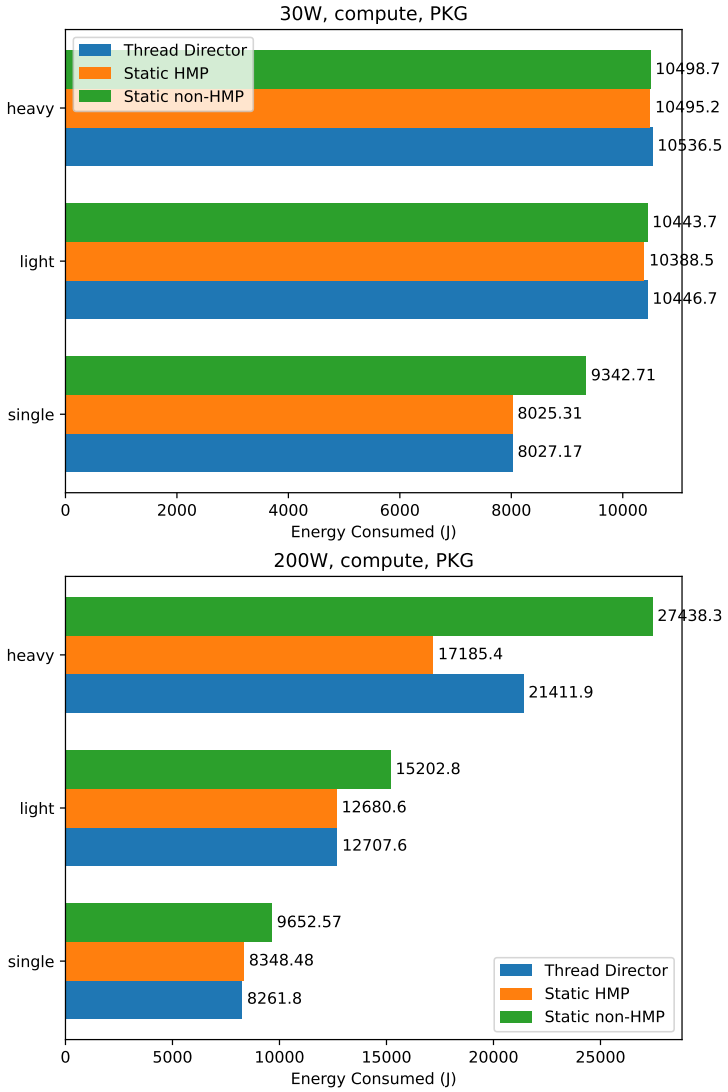

Figure 6.4: Total average Power consumption for the XW_XT_C_X workloads. In the 200 W workload, Thread Director scheduled on more cores, thus increasing power consumption, which explains the resulting anomaly

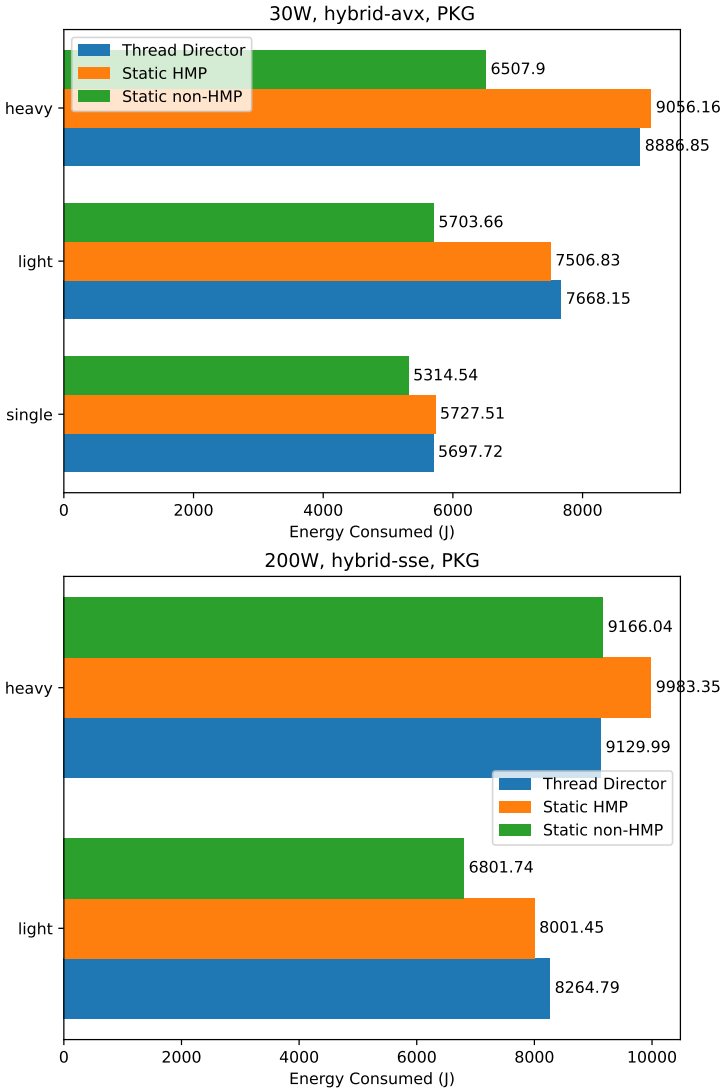

Figure 6.5: Total average power consumption for the XW_XT_HA/HS_X workloads. The threads running on E-cores spend less time sleeping, causing in higher overall power consumption.

But, as class two is never reported on E-cores, which we further elaborate on in section 6.4.4, this is never realized, perhaps showing part of the reason we do not see Thread Director behave as dynamically as expected.

This becomes particularly apparent when observing the frequency graphs, see figs. 6.6 and 6.7, where we see that Thread Director in our benchmarks never moves threads between cores after an initial wake-up period. While this is expected of the other schedulers, given they were designed that way, we expected to see some cores being put to sleep and others woken up during execution of the Thread Director scheduler, indicating a move from one core to another.

We see the compute workloads forcing cores to run at their maximum turbo frequency throughout the benchmark run, visible in fig. 6.6, which is expected behavior. With the hybrid workloads, we see expected behavior in the *static non-heterogeneous* scheduler, high variance in frequency with a low average frequency, seen in fig. 6.7, mirroring the "burst-type" workload we constructed. With the other schedulers, we instead see the cores mostly run at their maximum frequency, with some spikes to lower frequencies. We did not expect both core types to perform identically, but show similar behavior, below maximum turbo frequency with high variance. This is likely the result of poorly-chosen sleep duration, as the P-Cores complete their workload more quickly than the E-Cores, allowing them to sleep more on average compared to E-Cores, which hardly sleep at all.

We also found it notable that Thread Director never set the values of a core to zero in any of our test runs, despite severe load and external restrictions. We assume that these zero values are specifically reserved as a preservation mechanism in a thermal throttling scenario, which was not the case in our power constrained environment [9].

As we measured in a virtualized environment, we share execution time with other processes running on our host, which adds additional noise to our measurements and potentially increases power consumption. On Linux, we can prevent the scheduler from moving threads to specific cores with the `isolcpus` kernel parameter, which takes in a list of CPU cores to be isolated from scheduling [97]. Here, the only way to move threads to an isolated core is explicitly setting the affinity to that core. To see if our assumption holds up, we isolated all cores except the first one from scheduling and ran the `200W_XT_C_X` benchmarks.

[9] We were not able to run tests on a thermal restricted system, also partly due to time constraints.

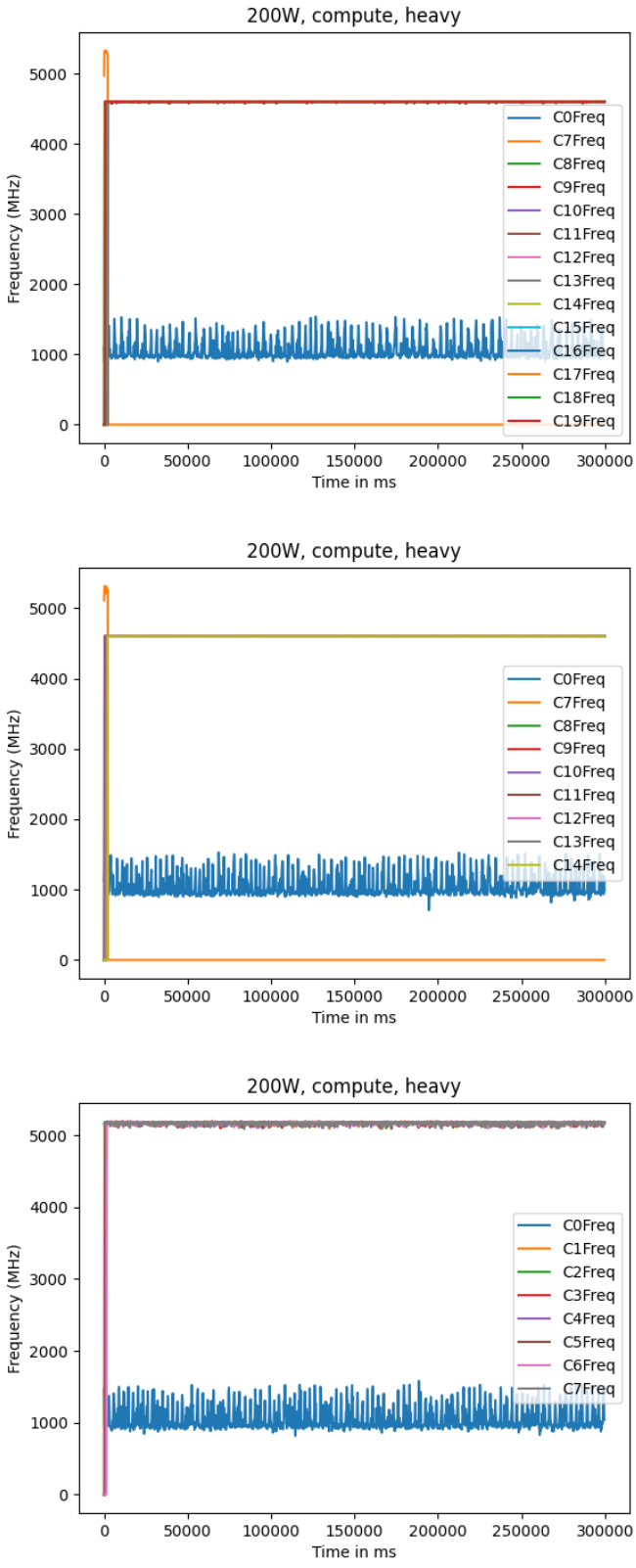

Figure 6.6: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

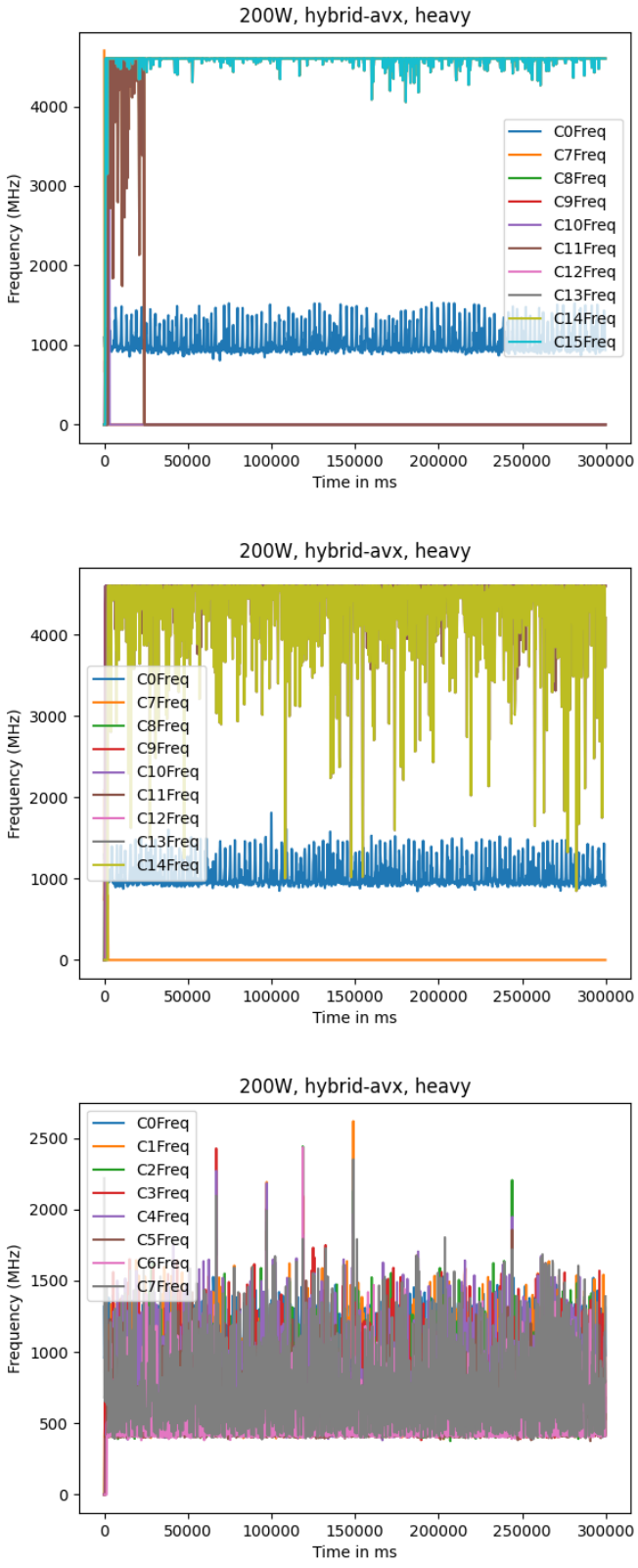

Figure 6.7: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

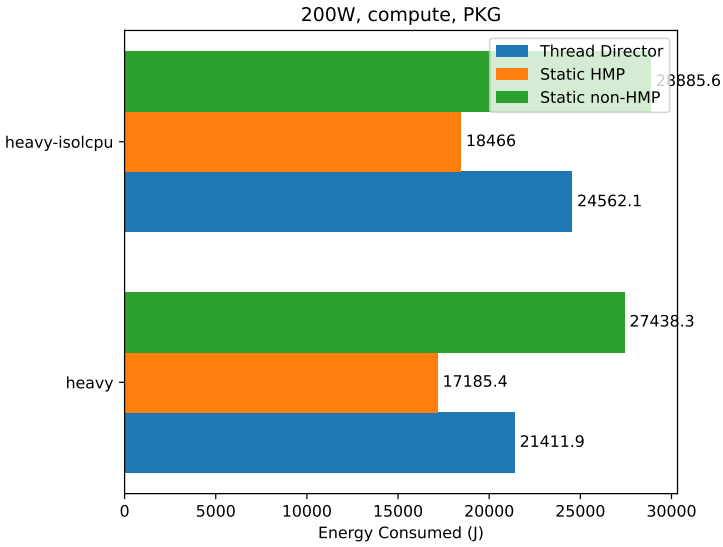

Figure 6.8: Total average Power consumption and frequency readings for the 200W_12T_C_X workloads, with and without isolation via `isolcpus`

Surprisingly, we actually see a slight increase in power consumption compared to when we did not isolate the CPU cores, see fig. 6.8. One explanation comes from the frequency readings, depicted in fig. 6.9, where we see our scheduling core `C0` both run at a higher average frequency and vary significantly more in frequency during execution. Another may come from the fact that the Linux scheduler takes advantage of our scheduling benchmarks not utilizing all cores, moving host processes to cores that our seL4 system leaves untouched, which averages out to a lower overall power consumption.

For a few select benchmarks, we also measured the temperature of our processor to see if Thread Director causes temperatures to be lower, which is also a form of energy efficiency. Unfortunately, here we also do not see notable results, see fig. 6.10. Our test bench is located in a server rack with adequate cooling and a temperature floor of $30C \pm 1C$ when the system is idle. Temperature readings were performed from software, with the accuracy of the measurements being $1C$. In-between values are the result of averaging between five test runs and do not represent a higher accuracy in measurement.

While we did not set a constant fan speed for our machine to standardize cooling, we doubt that the difference would have been notable, considering that we see only a slight increase in temperature, with our tests being at most $6 - 8C$ above the temperature floor, with differences between the schedulers being at worst $1 - 2C$. As temperature readings are also less accurate the lower the temperature is, this further adds a degree of uncertainty to our results.

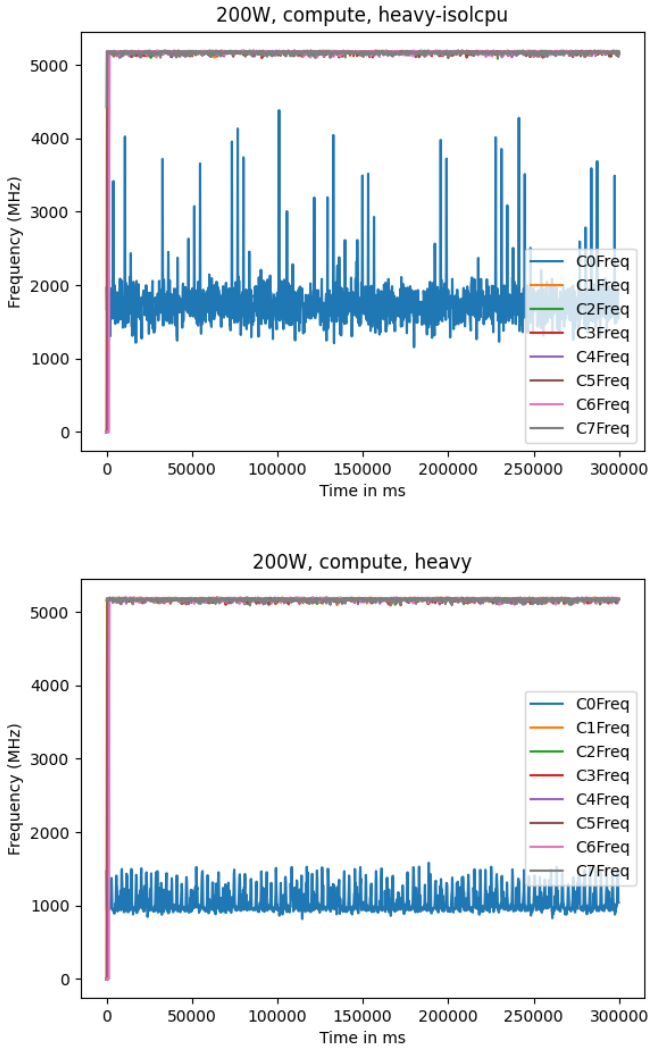

Figure 6.9: Frequency readings for the 200W_12T_C_X workloads, with and without isolation via `isolcpus`

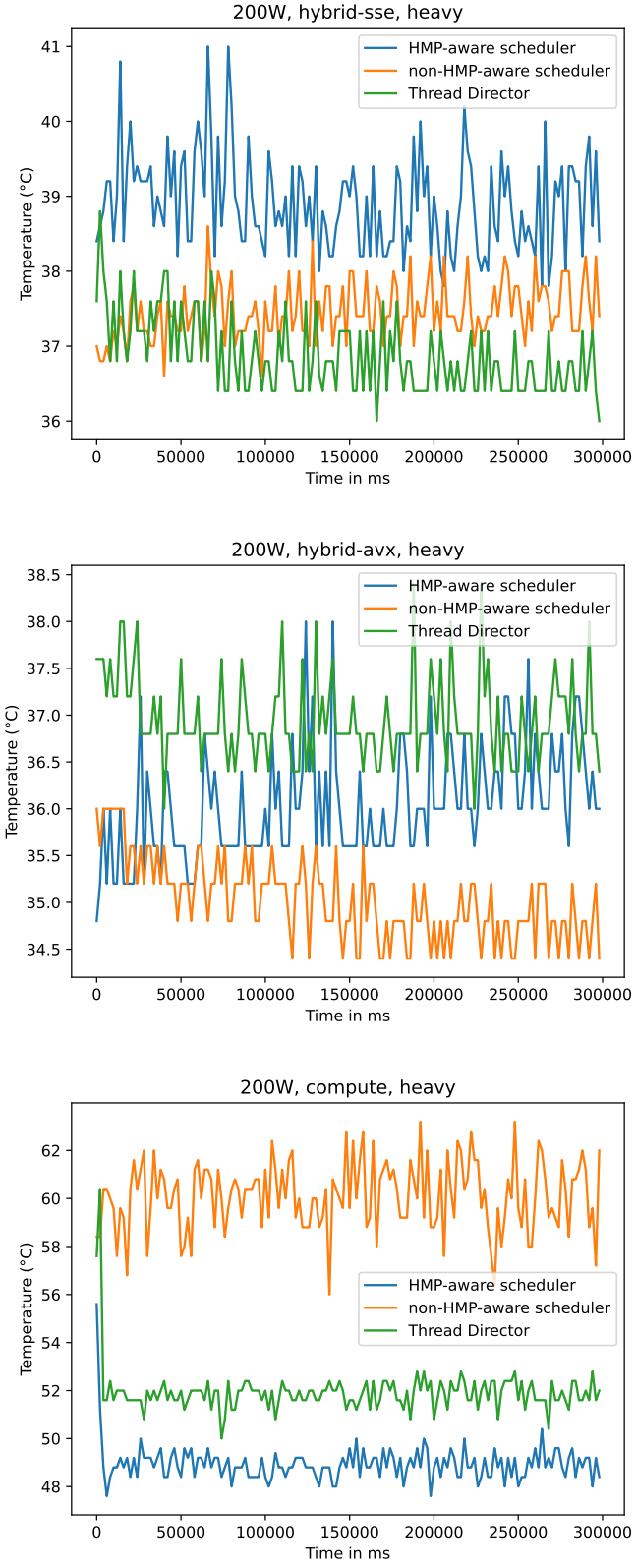

Figure 6.10: Temperature reading from select benchmark runs. Only every 100th sample shown. Baseline temperature at rest: 29-31°C. Accuracy of taken samples: 1°C

The compute workload shows no more information than what could be extracted from the total power consumption, showing a slight increase in temperature coinciding with an increased number of running cores.

6.4.2 On Oscillations And Load Balancing

Initially, we expected Thread Director to also assist scheduling in load balancing and oscillations, i.e., preventing threads from piling up on one core and preventing threads from "ping-pong"-behavior between two or more cores, but this proved to be not the case during our evaluation.

The assumption originally was that, based on increased load, Thread Director would deprioritize a core with many threads on it by lowering the associated value in the Thread Director table, possibly inferring this from a large amount of context switches on one core.

From observing some sample Thread Director tables figs. 28 and 30, this does not occur. Furthermore, from observing the values in the given tables, we see that many cores are given identical values even if they are heavily loaded with threads, leading to oscillations between cores if not explicitly taken care of and further solidifying that load on a core seemingly does not play into hints provided by Thread Director.

From this we conclude that Thread Director simply cannot be used as the only source for load balancing in a system, an idea we initially had and also suggested by the documentation when working on this thesis and the idea represented in our prototype implementation. While, this may seem foolish for others, given that what Thread Director provides to the OS is universally referred to as "hints", not directives the operating system has to follow, for us this dilutes the value provided by Thread Director if an operating system already has to do most of the work that it supposedly moves into hardware.

6.4.3 Overhead Analysis

The results we recorded are visible in fig. 6.11. A `RDMSR` operation on our system takes on average 148 cycles, with the system call adding on average 1,214 cycles. The `WRMSR` results are more surprising, showing that a `WRMSR` requires 3,715 cycles, with the system call overhead increasing slightly to 1,357 cycles.

We believe this discrepancy between `RDMSR` and `WRMSR` to come as a result of our virtualized environment, where a `WRMSR` may cause a `VMEXIT` to occur depending on hypervisor configuration. The slight increase in system call overhead may also be the result of a `VMEXIT` introducing a significant degree of noise, leading to overall higher values.

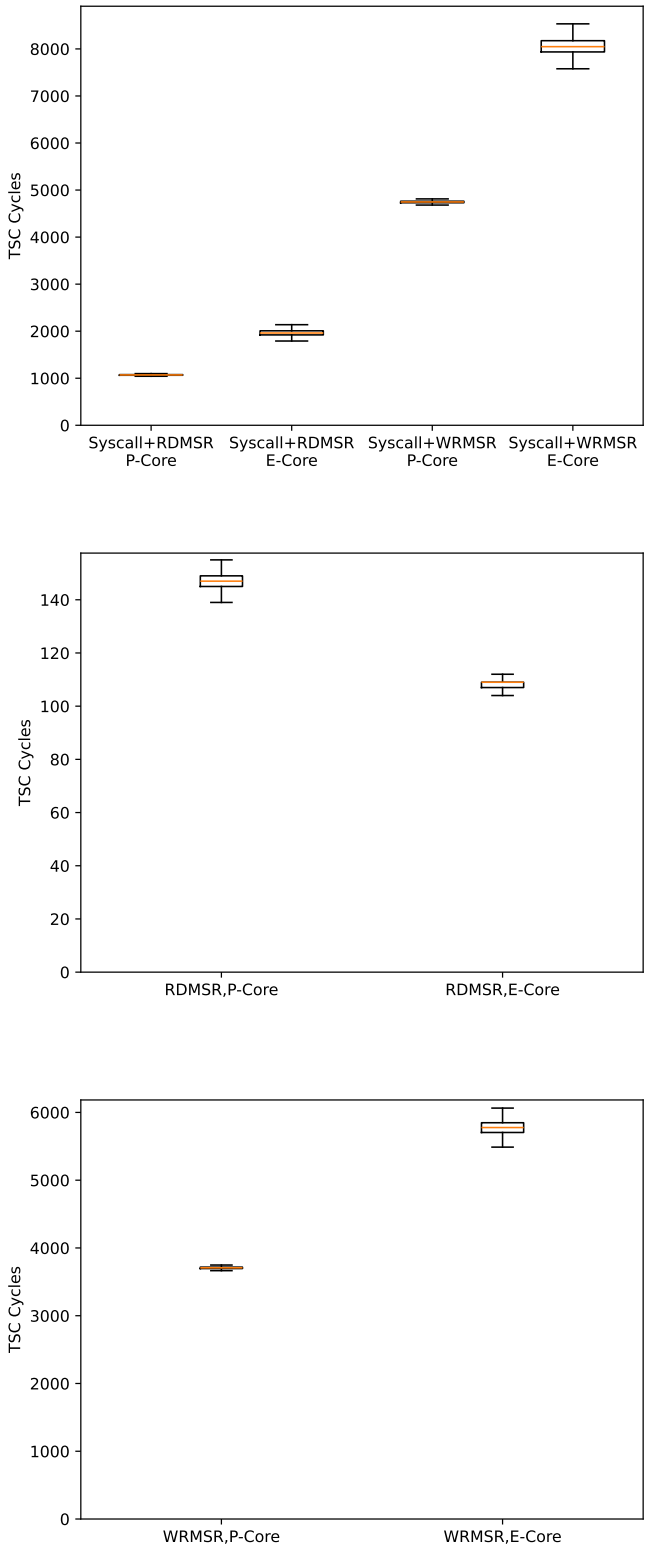

Figure 6.11: Time taken in TSC cycles. Results taken on a P-Core and an E-Core.

On an E-core the system call overhead increases alongside the latency for `WRMSR`. Surprisingly, `RDMSR` is slightly faster on an E-Core than on a P-Core, around 20 cycles, which is masked by the nearly doubled overhead of the system call.

6.4.4 Classification

The classification benchmarks, which can be seen in figs. 6.12 and 6.13, also show interesting behavior, starting with measurements taken on a P-Core. Here, we see that class one, two and three are classified very quickly, with class one on average being classified within 6,243 cycles, with class two and three taking slightly longer at 7,023 and 10,962 cycles, which, when converted to microseconds, correspond to Intel's claims of classification being done at a microsecond level [10]. However, class zero shows vastly different behavior, taking on average 1,248,458 cycles, or roughly 320 microseconds, to appear, with high variance across iterations. While it is still in the microsecond region, it is orders of magnitude slower than expected, and unlike the 6 to 21 microseconds reported by Chun et al. [34].

Running the same classification tests on an E-core show similar results for class one and three on average, where class three was reported slightly faster. Class two was never assigned to our thread on an E-core when using the same class two workload as the P-core, therefore we were unable to collect data for it. As of writing, this seems to be unknown to Intel, as we did not find a mention in the list of known errata for the Core Ultra 2 series [47]. For the sake of keeping the graphs legible we plotted the outliers separately in fig. 6.13. Compared to the same results on a P-Core, they are significantly worse, leading to an average time of 78,753 cycles. This result comes from a significant amount of outliers appearing on an irregular basis that took an order of magnitude longer than usual. For class zero, the outliers appear frequently enough to be visible in the plot, showing that class zero classification on an E-core may frequently take on the order of milliseconds to appear. We are uncertain whether this is a result of running in a virtualized environment or expected behavior on E-cores, and suggest further research into this phenomenon.

Furthermore, we do not know the exact workloads that Thread Director has been trained on, only the rough categories that trigger its behavior. It may be that a complex, real-world application running on a system with more noise like Linux or Microsoft Windows maps very well to this behavior, which does not hold for our microbenchmarks on seL4.

Besides this, we come to the conclusion that Thread Director is tuned to be very sensitive to changes in workloads that differ from normal execution, but sacrificing the ability to recognize normal operations in the process. As these changes in workload can generally be assumed to carry increases or decreases in power consumption with them, it is logical of Thread Director to be sensitive to those changes specifically when trying to maximize energy efficiency. Additionally, we note that this behavior is present on both types of cores, with the behavior being significantly more erratic on E-cores compared to P-Cores, which can be seen in figs. 6.12 and 6.13.

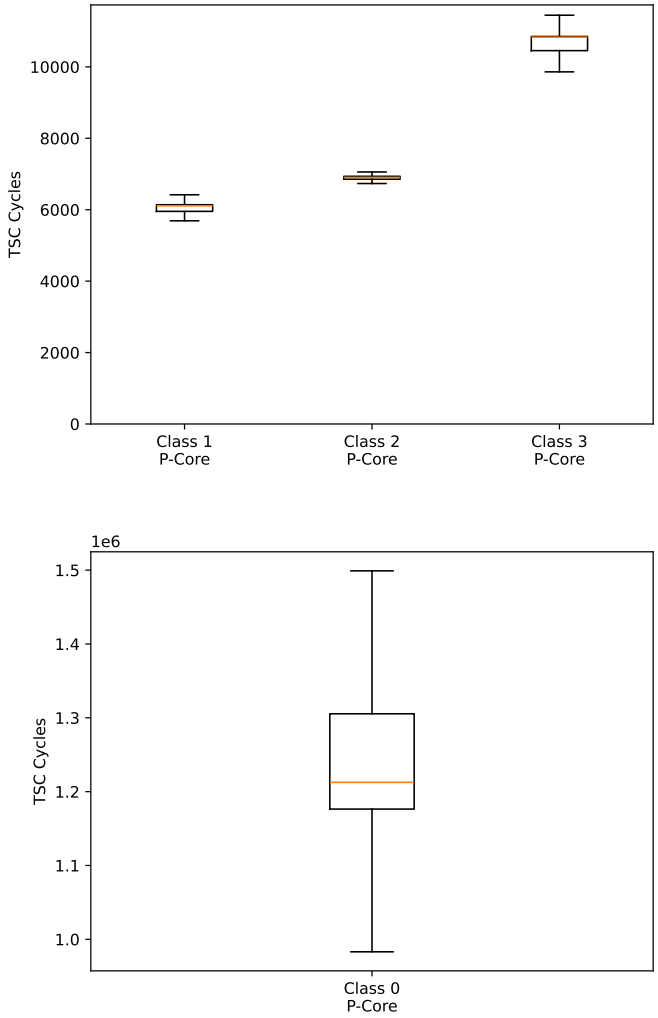

Figure 6.12: Time taken in TSC cycles for a class to appear. Results taken on a P-Core

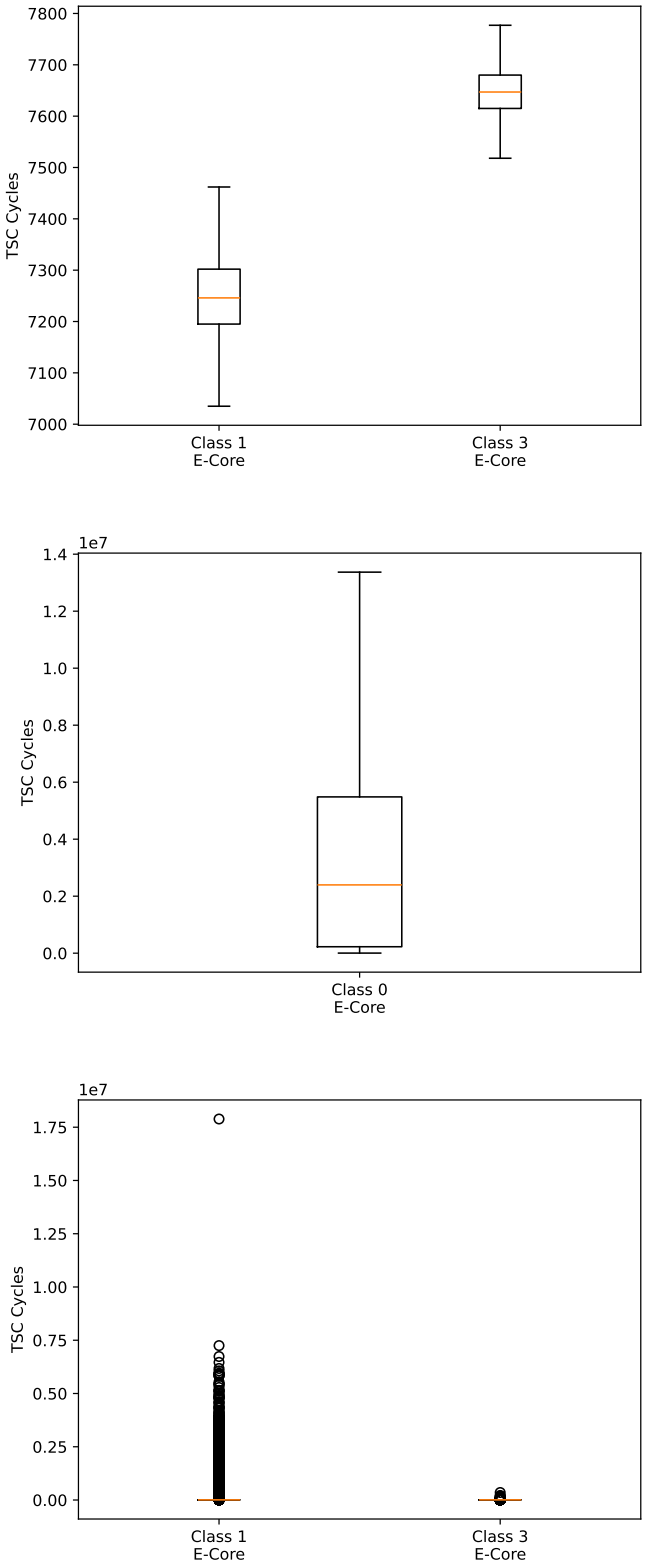

Figure 6.13: Time taken in TSC cycles for a class to appear. Results taken on a E-Core. Last graphic shows the first with outliers enabled.

6.4.5 Sleep Primitives

More expected behavior can be observed in fig. 6.14, confirming that in our virtualized environment, TPAUSE and UMONITOR/UMWAIT both successfully put the processor into a low sleep state and measurably reduce power consumption. The regular spikes in power consumption are likely also a result of us performing our testing in a virtual machine, coming from system noise of our host machine. However, even accounting for the spikes in power consumption, we still mostly stay below the average power draw of the busy-waiting approach.

Executing one of the user wait instructions may also possibly incur a VMEXIT, as they rely on the TSC to know when to wake up. Disabling interception here, however, is significantly more difficult, as no default interface for this exists and interception of the time stamp counter may be necessary to account for time spent in the host. As most configurations, e.g., of cloud instances, will not allow to disable this interception, we see this as the default, acknowledging that there may be a slight improvement in energy efficiency. Thus, we have not considered this approach.

The difference made by including a PAUSE instruction, which is a recommended practice for spin-loops [98], was within margin of error. This makes sense, as the PAUSE instruction only helps when the duration spent in the busy-wait loop is very short, which does not apply to our scheduler. We later realized that the PAUSE instruction was being intercepted by KVM, which can be disabled via KVM_CAP_X86_DISABLE_EXITS similar to disabling interception for IA32_MPERF and IA32_APERF. This ultimately did not matter, as the change between interception and non-root execution was also within the margin of error, suggesting that this is not something of concern and does not require explicit configuration. For longer waits, the manual recommends operating system mechanisms, which would defeat the purpose of wanting to bypass the kernel as much as possible, and in the case of seL4 we can only use seL4_Yield for this, which yields the thread to the scheduler.

On the PKG RAPL plane, we see a drop from 7,457 J using busy-waiting to 5,019 J when using TPAUSE and a drop to 5,352 J with UMONITOR/UMWAIT. For PP0 the change is more significant, dropping from 5,327 J to 2,639 J with TPAUSE and 3,107 J with UMONITOR/UMWAIT. The slightly higher power consumption of UMONITOR/UMWAIT likely comes from the additional overhead from allowing the hardware to wake up early when a write occurs [8]. As this allows us to react quicker to signals from threads, we see this as a worthy tradeoff between energy efficiency and responsiveness, as the drop in power consumption is still significant compared to a busy-wait. Compared to related work [36], this is a higher drop in power consumption than expected, as they report a drop of $\sim$20-30% compared to our

∼40-50%, but not wholly outside the realm of possibility, as our scheduler is an outlier example of a program that spends the majority of its time in an idle state.

In our case, we are not concerned with the latency of `UMONITOR/UMWAIT` and `TPAUSE` at the cycle level, as we wait for durations that cause this latency to be negligible. However, we did notice that despite not setting a maximum value that the two instructions may sleep before waiting, we did consistently wake up after ∼100,000 cycles. A quick check of the `IA32_UMWAIT_CONTROL` MSR on our host, which contains the maximum value that these instructions may sleep for before being preempted, and comparing to the value reported inside seL4 shows that this MSR is not passed through to guest virtual machines. Whether this is intentional behavior or an oversight, we are unsure, but this led us to also test `UMONITOR/UMWAIT` and `TPAUSE` with this value set to unrestricted on our host, where the changes were within margin of error.

Putting the processor to sleep also inherently comes with an increase in latency, causing oversleep, as the core has to be woken up to be operational again, which can take a variable amount of time depending on the frequency of the processor and the requested sleep state, as reported in [36]. A busy-wait loop that also checks for writes similar to the wake up property of `UMONITOR/UMWAIT` in this case has the advantage of being able to respond much more quickly than either `UMONITOR/UMWAIT` or especially `TPAUSE`, as here in the worst-case we may sleep for an additional ∼100,000 cycles before being able to exit again. For our scheduler example, this is no issue, as the time spans we concern ourselves with are far greater than the worst possible latency that any userspace sleep primitive may cause, but for latency-critical programs, where deadlines down to the cycle count are important, a balance between response time and energy efficiency must be taken.

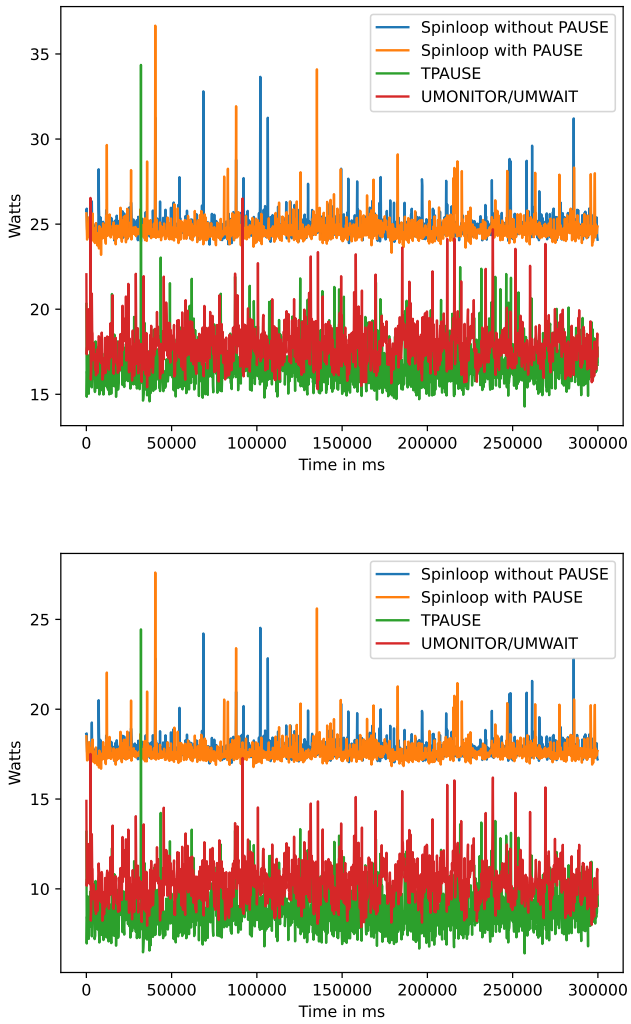

Figure 6.14: Average power consumption over 5 minutes. For clarity only every 10th sample shown.

Chapter 7

Conclusion

In this chapter, we first discuss our findings, conclusions, and takeaways. Then wrap up with possible starting points for future work.

7.1 Conclusion

With *energy efficiency* becoming more important year over year, innovative technologies that allow saving energy with relatively little additional effort, like the user wait extensions and Thread Director, are always welcome. In this thesis, we investigated whether these technologies actually prove an uplift to energy efficiency on a microkernel, and in the case of Thread Director, we are, to our knowledge, the first to investigate it on such a minimal system. For this, we present a way to implement userspace scheduling on seL4 with hardware assistance and implemented most of what we have designed, showing that not only is it possible to implement such components on top of seL4, but also relatively straightforward, e.g., compared to more complex operating systems. In our evaluation, we set out to test Thread Director under a variety of different workloads, alongside other metrics like the classification speed of Thread Director and the energy savings offered by the user wait extensions.

Across all our benchmarking runs we were unable to find Thread Director itself providing any increase in energy efficiency compared to an HMP-aware scheduler, with Thread Director sometimes being less efficient than a non-HMP-aware scheduler. Outside of the focus on energy efficiency, we were able to find anomalous behavior in the classification speed of Threads and a possible hardware bug on E-Cores, where class two is never reported, which may partly explain why we were unable to meaningfully increase energy efficiency.

Our findings in energy efficiency line up with similar results reported in previous works [48, 50], which is unfortunately both expected and disappointing,

but also confirm that the mixed results from Thread Director are more dependent on the running application instead of any sort of underlying implementation in an operating system. If we were able to know which programs were used for the training of Thread Director, we may be able to draw a more definitive conclusion here instead of speculating where the reported gains from Thread Director originate from.

Our findings might be more interesting, if we would have been able to test more realistic workloads, e.g., interactive or graphical programs. However, such complex programs are currently not publicly available for seL4. Maybe we would have seen Thread Director make more of a difference, as this would likely map more to the dataset Thread Director was trained on, as it appears to be predominantly trained to be used on Windows.

We were mostly limited in our benchmarking by the fact that programs are not easily portable to seL4 without also implementing other complex system services, which can quickly spiral into the development of an entire operating system. One may take for granted how many conveniences a monolithic kernel like Linux provides, which we realized very quickly during our time working on this thesis. Being small does not mean being simple, is what we wish to convey and offer as an explanation to our benchmarking approach.

Despite these limitations, though, we see merit in working and building on top of seL4. The addition of new features was, after a learning period, rather straightforward and convenient, and the open and free nature of seL4 lends itself to an easy testing ground for new operating system features that then could be adapted to work in more established operating systems. The issues we encountered were not because of a poor design or feature documentation, but of a lack of supporting drivers and kernel documentation, conveniences we take for granted in larger projects like Microsoft Windows and Linux. Given enough motivation and interest by developers, this could also be the case for seL4.

However, we were able to implement a userspace scheduler that is capable of scheduling threads based on user-defined policy, can migrate threads between cores for load balancing, gather data from hardware to assist in scheduling decisions, if present, and allows for the system to recover from a failed userspace scheduler without taking down the system. Theoretically, our scheduler is not as optimal as it could be, but we are aware of several approaches that can be used to further improve it, see section 7.2.

7.2 Future Work

Here, we describe possible starting points for future work, starting with two major extension points and wrapping up with a variety of more minor improvements, we either could not implement in time, or were out-of-scope for this thesis, or we discovered late during testing, thus, it was far too late to adjust to these possibilities. Finally, we conclude with further benchmarks that we came up with, but did not pursue for reasons of feasibility or time.

7.2.1 Integration Of Userspace Interrupts

While we were successfully able to shift the kernels role in scheduling to a simple dispatcher, we still need to interact with the kernel more than we prefer. When interacting with a core other than the one we are running on, we need to send an IPI, which must currently be done in kernel-space, which furthermore is not optimized for SMP in the recent implementation of seL4. Additionally, when engaging in IPC, we also need the kernel to send the messages from one thread to another.

With userspace interrupts [8, 92], both of these can be done in userspace. Userspace interrupts allow, after some configuration in the kernel, interrupts to be received in userspace. This opens up several avenues of extension for our scheduler.

For one, we could redirect a periodic timer interrupt to userspace, allowing for a design with per-core userspace schedulers, instead of being restricted to the centralized scheduling model we designed for this thesis. Furthermore, userspace interrupts can also be used for more performant IPC that can, after configuration, be fully handled in userspace, which would allow threads to communicate both with the scheduler and each other while avoiding entry into the kernel.

This has already been done in Linux, showing up to a *16x* improvement compared to mechanisms like Pipes or Signals [99]. Their approach, however, relies on file descriptors, which is rather awkward to port to seL4.

Recently, Gurres implemented Userspace Interrupts in seL4 for his Bachelor thesis [92], detailing the process and relative performance increases that such an approach can yield for IPC. We therefore see great potential in merging the changes done to facilitate userspace interrupts into our userspace scheduler, but this fell outside our scope, as we focus on scheduling improvements with hardware information.

With two of the three major abstractions provided by a microkernel removed, leaving only memory management, we see that the kernel need not concern itself with running beyond one core, as other cores could reasonably be fully managed by a single core still remaining with the kernel. The reason this is important,

as seL4, being concerned with formal verification above all else, proposed the concept of a partitioned multikernel [67], whereby a single-core version of the kernel runs on every core and negotiation between kernels regarding resources is handled via IPIs. This has the benefit of supporting multiple cores, while retaining the formal verification of the single-core variant. Therefore, we propose a hypothetical system with userspace interrupts and user-managed kernel threads as an alternative solution to the partitioned multikernel approach, where the kernel runs on only one core and hands off control of the remaining cores to userspace. In this system, interaction with the kernel would only extend to initial configuration of the userspace operations and services that cannot or cannot yet be provided by userspace. The main reason for entering the kernel in this system would be the configuration and reading of various MSRs at the request of userspace services with the privileged `(RD/WR)MSR` instruction. In a future system, this could be reduced to a single entry, which configures the desired MSRs to be read from and written to by userspace with a future unprivileged `U(RD/WR)MSR` instruction [33].

7.2.2 Integration In seL4 Ecosystem

As we had no previous experience regarding both working with real operating systems and especially microkernels, we implemented our userspace scheduler in the most minimal environment that allowed us to modify the seL4 kernel, which ended up being the "Hello world" tutorial as could be found on the seL4 website [100]. During development we were aware that further tooling for seL4 in userspace existed, notably the `CAmkES` and `Microkit` [76] frameworks, but could not allocate the time to learn how to take advantage of them. Now that development has finished, we realize that implementation of more complex schedulers would become rather difficult without these frameworks, and suggest integrating the features we added and make them accessible to programs taking advantage of these frameworks. More complex schedulers would also allow more detailed testing and benchmarking of userspace scheduling in general.

As these frameworks both build upon the seL4 API and defining one's own scheduler was not a possibility until now, we expect little to get in the way of integration into these frameworks. We reason that, while there may be some assumptions about scheduling currently present in these frameworks, these should not be blockers for a possible integration of userspace scheduling, as we tried to maintain a high degree of compatibility with existing seL4 tooling by only touching the areas of the kernel we had to and following a capability-based approach.

7.2.3 Extending The Current Implementation

Like we stated before, we never worked with a microkernel-based system, leaving open the possibilities of minor improvements to our prototype implementation. In its current implementation, both Thread Director and Userspace scheduling are united in one capability and allow control over all cores reserved for userspace. Instead, we suggest the approach taken by the *IOPort* and *IOPortControl* capabilities, whereby the control capability manages enabling and disabling Thread Director, setting the reservation mask and issuing scheduling capabilities, which allow a userspace scheduler to only manage a set range of cores. This subdivision of reserved cores is already possible in our current implementation, as copies of the *ITD* capability can be created, but userspace schedulers need to take care not to overlap.

Furthermore, we could not determine how our implementation would have to be adapted to work with the MCS version of the seL4 kernel, particularly how timer interrupts are handled. One of the benefits of the MCS kernel is the conversion to a tickless kernel [21], which brings the advantage of timer interrupts that need to be regularly rearmed, instead of being fired periodically.

Currently, we do not disable the timer interrupt on reserved cores, instead just acknowledging it without doing anything. This is a current work-around, as we do not want to interfere with every compartment of the seL4 kernel. But in the future it would be possible to disable and enable the periodic timer interrupt.

With the deadline timer this would not be an issue as the kernel does not rearm the timer on reserved cores.

As a final point, we also wonder if our implementation is sufficient enough that it can be adapted from a centralized scheduling model to a distributed scheduler without needing significant changes.

7.2.4 Further Evaluation

Furthermore, we see that more opportunities for evaluation options exist, which have the potential to give more interesting insights into the workings of Thread Director. Some of them were skipped due to a lack of time, lack of resources in seL4, or creating them in time for properly testing.

For one, we were only able to test fairly static workloads where not much changed during execution. A more dynamic or even real-world application may have yielded more interesting results, but knowing the limitations of working with a microkernel, see section 6.2, this could prove to be quite difficult.

Likewise, we could only evaluate a power constrained system, where we defined the power constraint before runtime, as we worked with very limited tooling, where dynamically setting power limits at exact points in time would have currently been impossible. Future work could perhaps perform the same testing, but vary both thermal and power constraints during execution, which could also invoke Thread Director to perform more dynamic behavior.

Another dimension to test would be testing with a more advanced scheduler. We only had time to develop a simple RR scheduler, but future work could look into a port of a more advanced scheduler, like CFS, to seL4 and see how it performs and what overheads come with that implementation.

Code Snippets

This chapter contains various code snippets referred to in the thesis.

```
for(int i = 0; i < loopCnt; ++i) {
    asm volatile (
    "xor %%rax, %%rax\n"
    "inc %%rax\n"
    "add %%rax, %%rax\n"
    ::: "rax");
}
```

Figure 1: Code for invoking class 0

```
for(int i = 0; i < loopCnt; ++i) {
    asm volatile (
    "vxorps       %%ymm0, %%ymm0, %%ymm0  \n"
    "vxorps       %%ymm1, %%ymm1, %%ymm1  \n"
    "vxorps       %%ymm2, %%ymm2, %%ymm2  \n"
    "vfmadd132ps %%ymm0, %%ymm1, %%ymm2  \n"
    "vfmadd213ps %%ymm1, %%ymm2, %%ymm0  \n"
    "vfmadd231ps %%ymm2, %%ymm0, %%ymm1  \n"
    ::: "ymm0", "ymm1", "ymm2"
    );
}
```

Figure 2: Code for invoking class 1

```
//class_test.c
for(int i = 0; i < loopCnt; ++i) {
    peakflops_avx(8, buffer);
}
//peakflops_avx.S
...
1:
vmovapd ymm1, [rsi + rax * 8 ]
vmulpd ymm0, ymm0, ymm1
vaddpd ymm2, ymm2, ymm1
vmulpd ymm3, ymm3, ymm1
vaddpd ymm4, ymm4, ymm1
vmulpd ymm5, ymm5, ymm1
vaddpd ymm6, ymm6, ymm1
vmulpd ymm7, ymm7, ymm1
vaddpd ymm8, ymm8, ymm1
vmulpd ymm9, ymm9, ymm1
vaddpd ymm10, ymm10, ymm1
vmulpd ymm11, ymm11, ymm1
vaddpd ymm12, ymm12, ymm1
vmulpd ymm13, ymm13, ymm1
vaddpd ymm14, ymm14, ymm1
vmulpd ymm15, ymm15, ymm1
add rax, 4
cmp rax, rdi
jl 1b
...
```

Figure 3: Code for invoking class 2. `peakflops_avx.S` was not written by us, instead taken from Likwid, as we already had the files.

```
for(int i = 0; i < loopCnt; ++i) {
    asm volatile("pause" ::: "memory");
}
```

Figure 4: Code for invoking class 3

```
   for (int i = 0; i < 1024 * 256; ++i) {
     seL4_X86_ITD_ResetHistory(itd); //Wrapper around HRESET
instruction
     start = read_timestamp(); // RDTSC surrounded by LFENCE
instructions
     for (;;) {
       class_id_workloads(class, 100); //Runs class-specific
workload for 100 iterations.
       seL4_Word value = seL4_X86_ITD_GetClassRaw(itd).
rawClass; //Returns raw value of MSR
       if (value & BIT(63)) {
         if ((value & MASK(8)) == class) {
           stop = read_timestamp();
           break;
         }
       }
     }
     double diff = stop - start;
     //get_tsc_frequency returns the frequency of the TSC,
for conversion to us
     sprintf(scratch, "%.0f,%f\n", diff, diff /
get_tsc_frequency());
     send_over_ioPort(scratch, port); //Print out results in
CSV to host.
   }
```

Figure 5: Code for obtaining class readings

```
   for (int i = 0; i < 1024 * 1024; ++i) {
     seL4_Word before = read_timestamp(); //LFENCE + RDTSC +
LFENCE
     seL4_X86_ITD_TestWRMSRLatency_t result =
seL4_X86_ITD_TestWRMSRLatency(itd);
     seL4_Word after = read_timestamp();
     sprintf(scratch, "%lu, %lu\n", after - before, result.
latency);
     send_over_ioPort(scratch, port); //Print in CSV to host
   }
```

Figure 6: Code for obtaining latency readings, userland

```
asm volatile("lfence");
word_t before = x86_rdtsc();
asm volatile("lfence");
x86_rdmsr(IA32_THREAD_FEEDBACK_CHAR_MSR);
asm volatile("lfence");
word_t after = x86_rdtsc();
asm volatile("lfence");
tcb_t *thread = NODE_STATE(ksCurThread);
//Returns the difference in TSC cycles to userland
if (call) {
    setRegister(thread, badgeRegister, 0);
    unsigned int length = setMR(thread,
                                buffer,
                                0,
                                after - before);
    setRegister(thread,
                msgInfoRegister,
                wordFromMessageInfo(
                    seL4_MessageInfo_new(0, 0, 0, length)
                ));
}
```

Figure 7: Code for obtaining latency readings, kernel

Further Data

Other data collected on Thread Director, which we did not discuss in the thesis. If necessary, captions contain additional information relevant to the presented data.

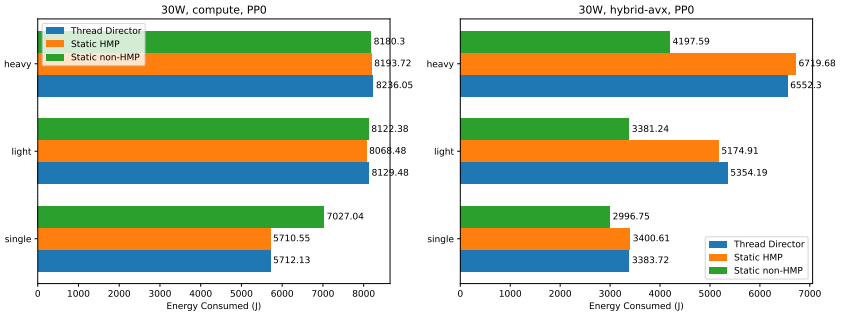

Figure 8: Total average Power consumption for the 30W_XT_C_X and 30W_XT_HA_X workloads

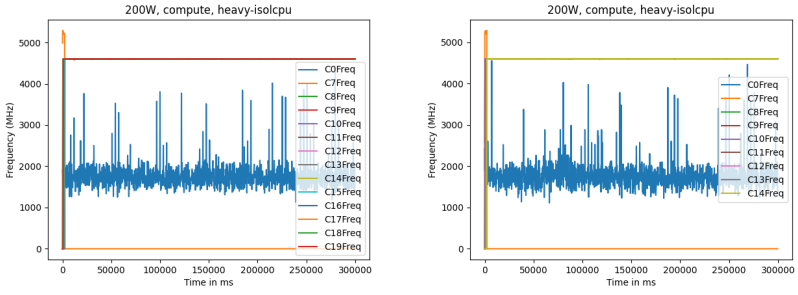

Figure 9: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core. Cores 1-19 isolated from host scheduling.

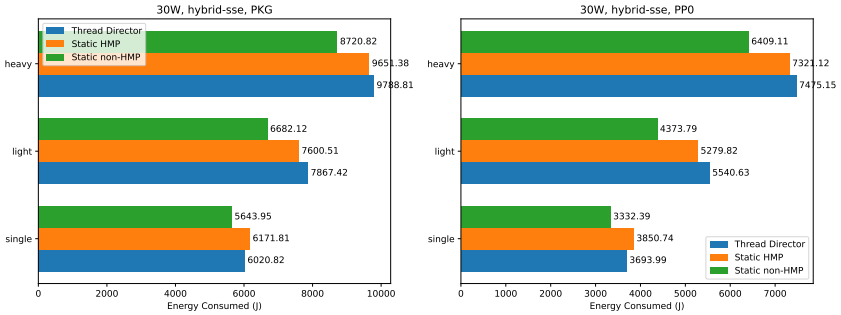

Figure 10: Total average Power consumption for the 30W_XT_HS_X workloads

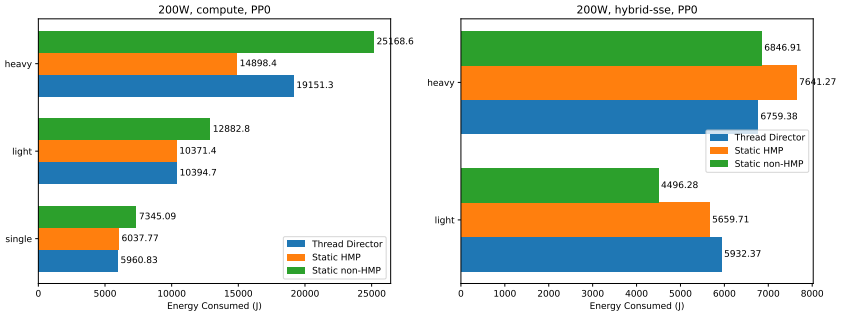

Figure 11: Total average Power consumption for the 200W_XT_C_X and 200W_XT_HS_X workloads. In the case of the compute benchmark, Thread Director scheduled on more cores, thus increasing power consumption.

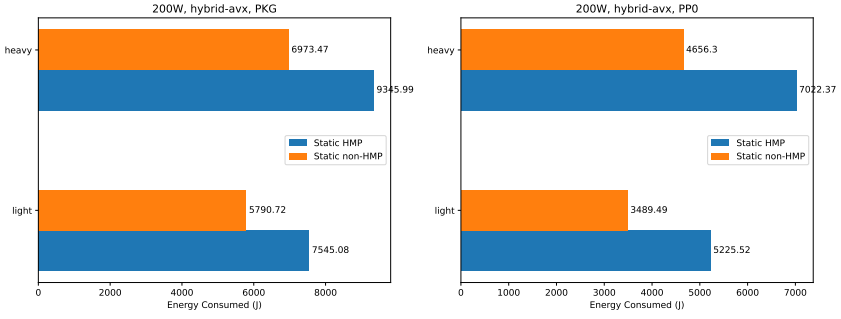

Figure 12: Total average Power consumption for the 200W_XT_HA_X workloads. Missing results for Thread Director, due to a defect in our benchmarking script. Colors not the same as in previous graphs

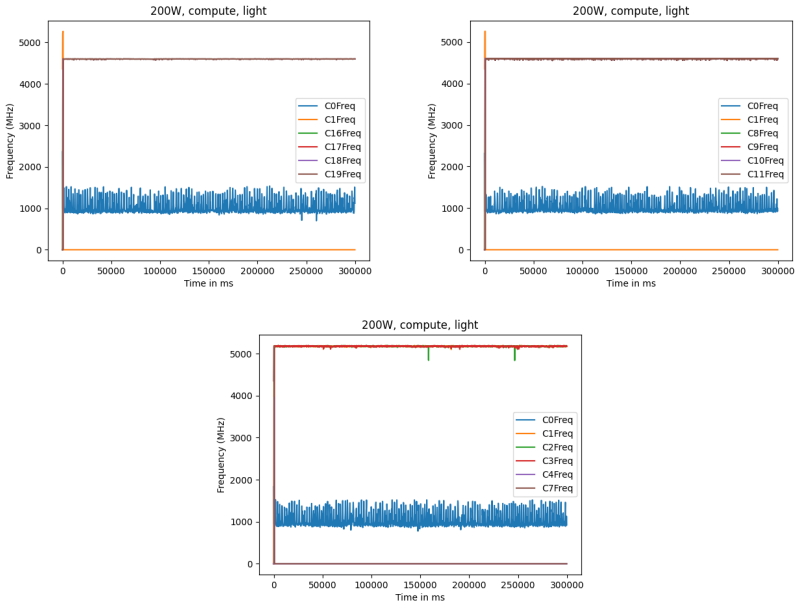

Figure 13: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

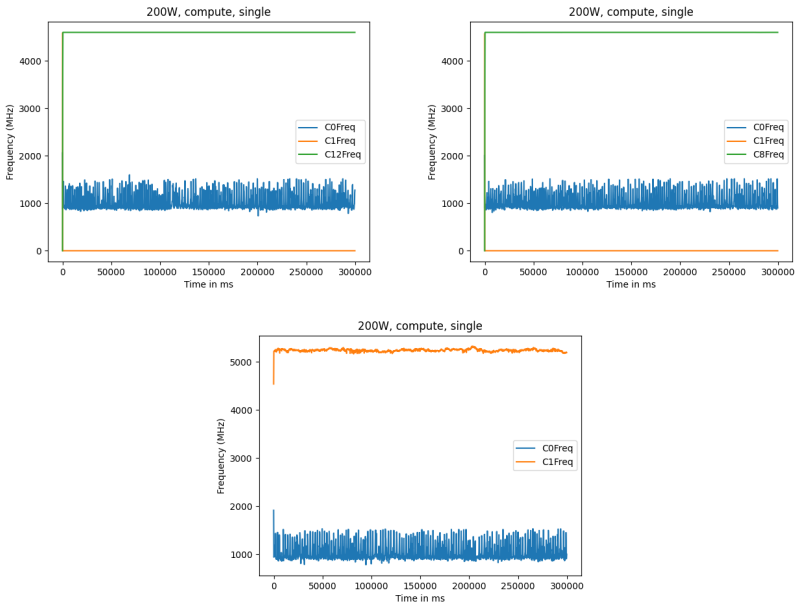

Figure 14: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

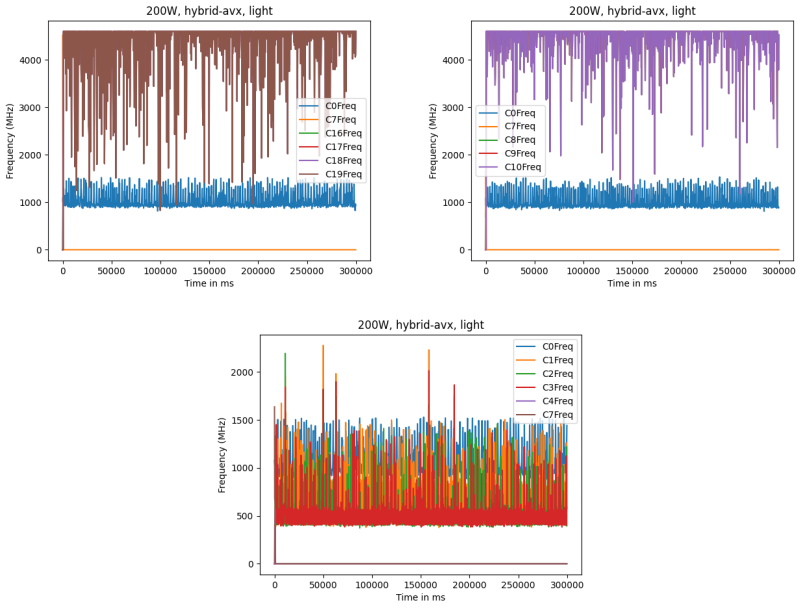

Figure 15: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

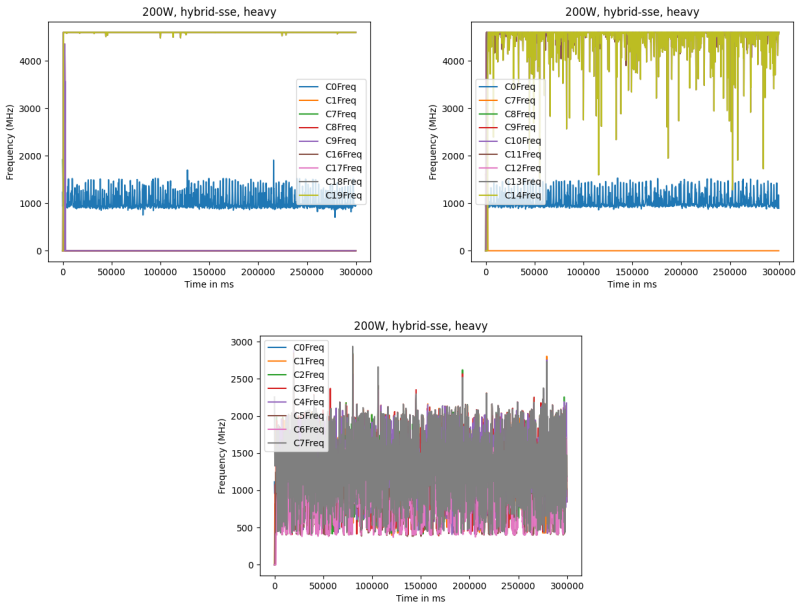

Figure 16: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

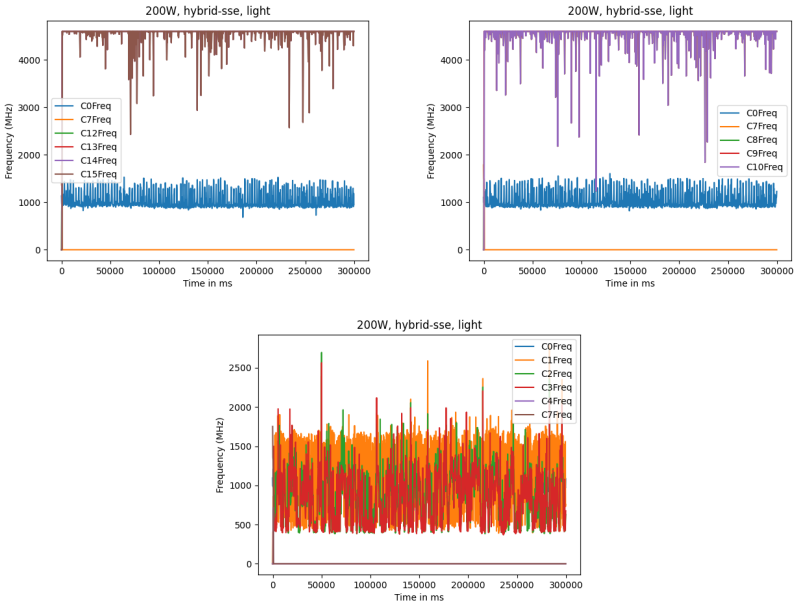

Figure 17: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

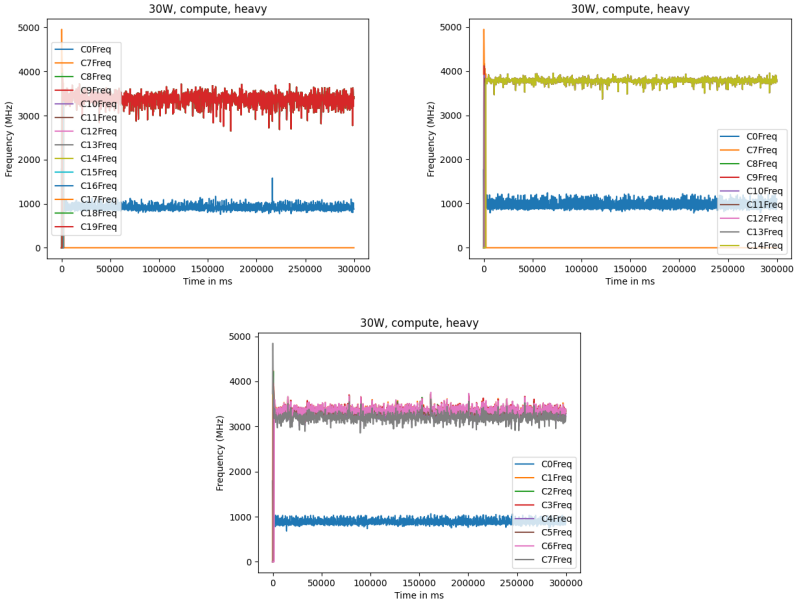

Figure 18: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

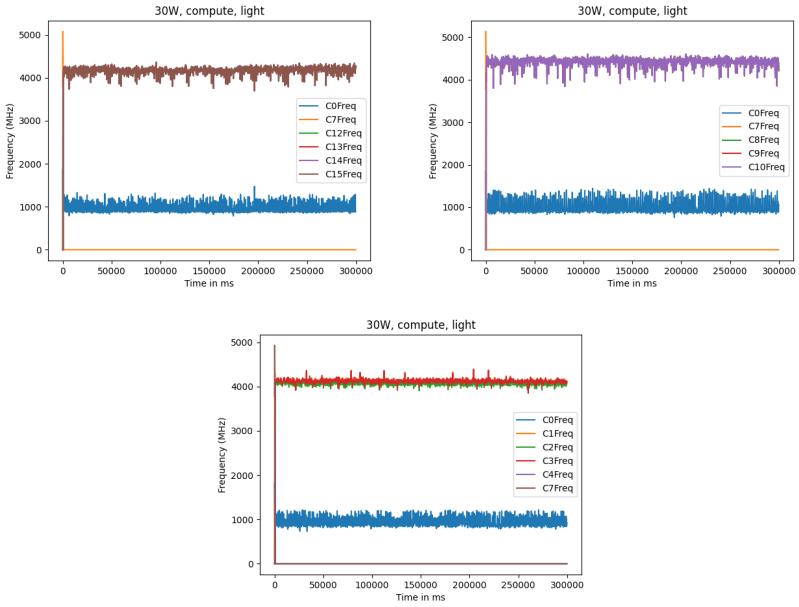

Figure 19: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

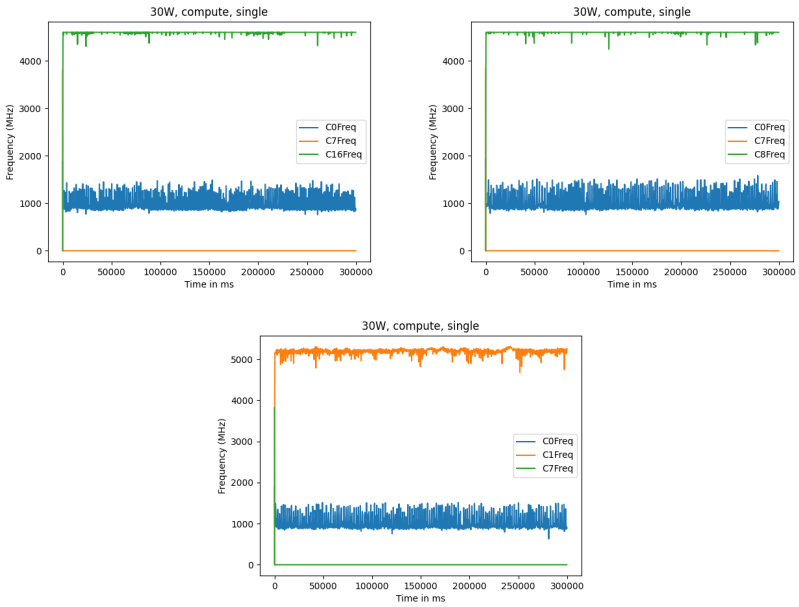

Figure 20: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

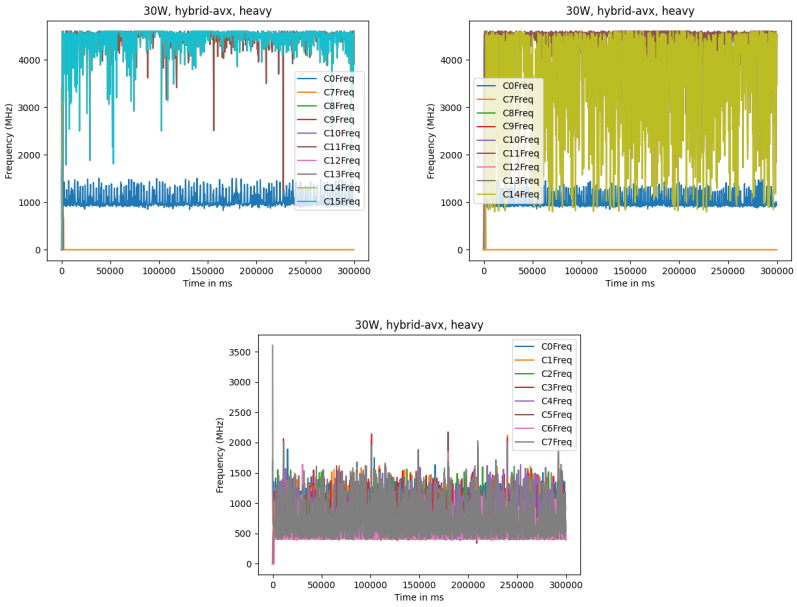

Figure 21: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

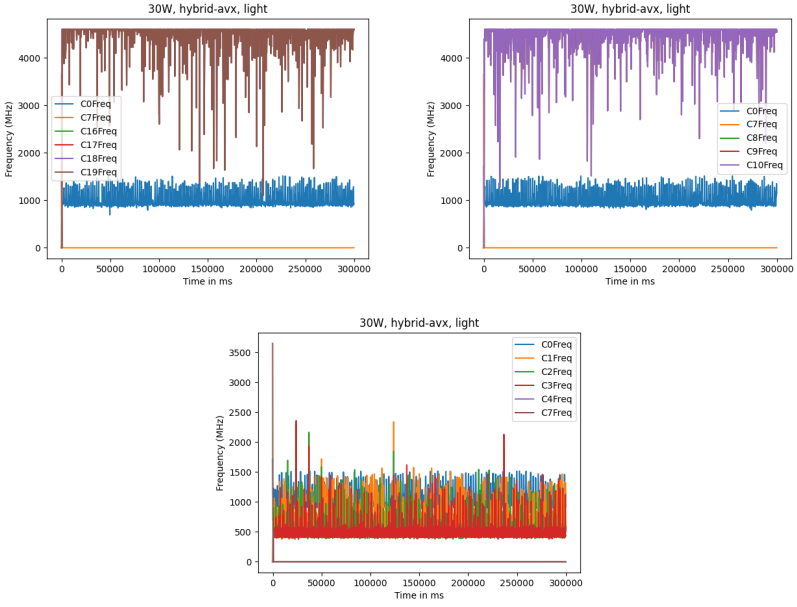

Figure 22: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

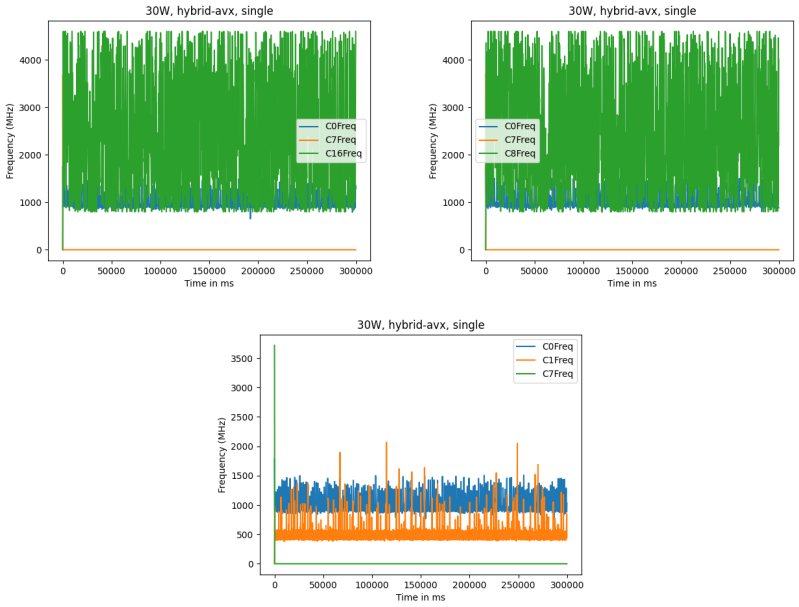

Figure 23: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

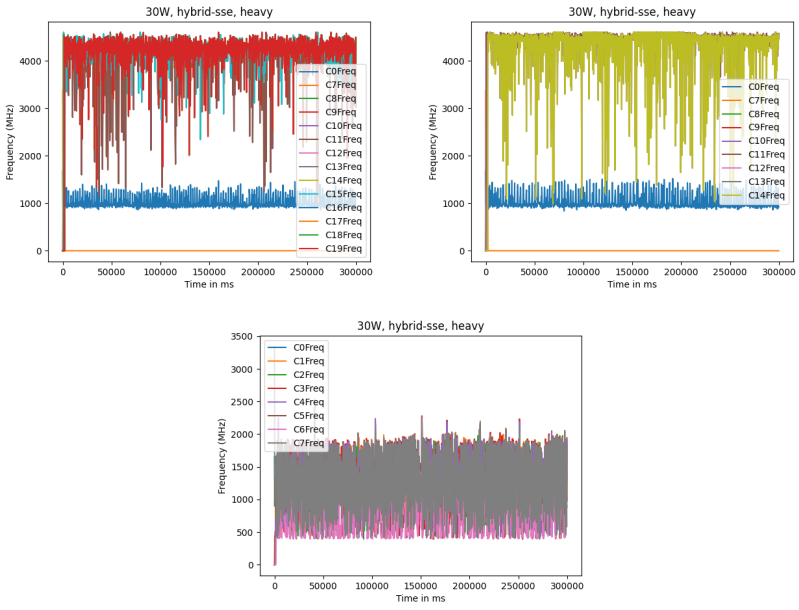

Figure 24: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

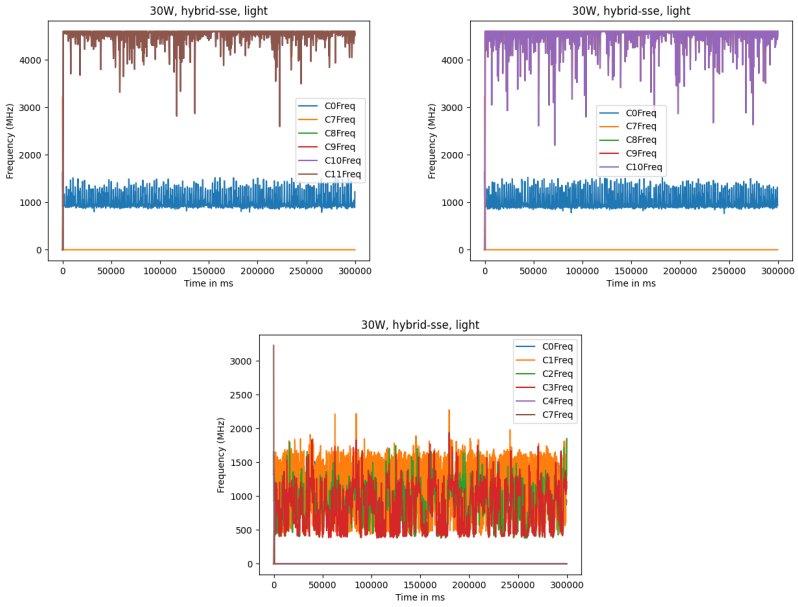

Figure 25: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

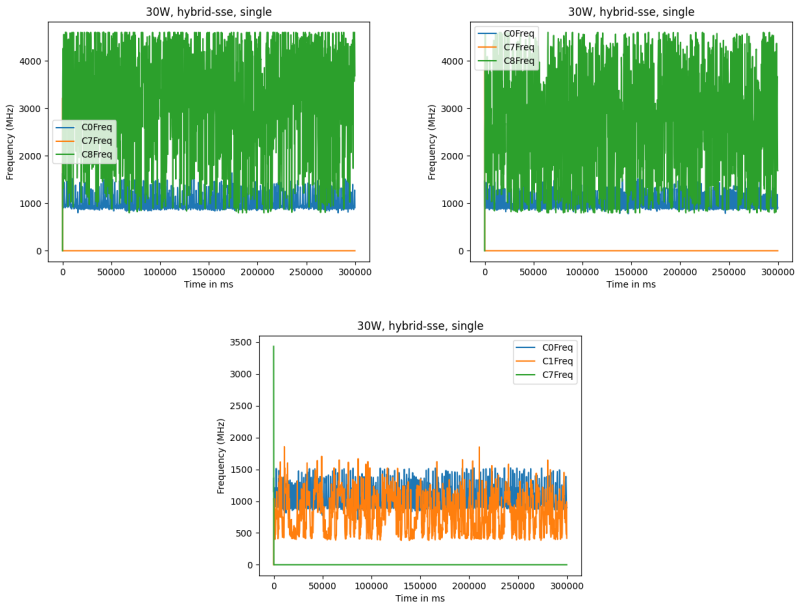

Figure 26: Average frequency over one test run. Only active cores plotted. Core zero is the scheduler core.

Thread Director tables

Here, we include some Thread Director tables we found interesting enough to mention.

Do note that capabilities for E-cores are always reported as clusters of four, meaning, for example, the capabilities of core eight are the same as core nine, ten, and eleven, which is why we only recorded the values for each cluster here.

Core	Class	Performance	EnergyEfficiency
0	0	86	31
0	1	105	32
0	2	197	45
0	3	48	17
…	…	…	…
6	0	88	31
6	1	107	32
6	2	200	45
6	3	49	17
7	0	88	31
7	1	107	32
7	2	200	45
7	3	49	17

Figure 27: Thread Director table with default values for our hardware, P-Cores. Core 0-5 reported identical values.

Core	Class	Performance	EnergyEfficiency
8	0	64	151
8	1	64	119
8	2	64	92
8	3	46	108
…	…	…	…
12	0	64	154
12	1	64	121
12	2	64	94
12	3	46	110
…	…	…	…
16	0	64	155
16	1	64	122
16	2	64	95
16	3	46	111
…	…	…	…

Figure 28: Thread Director table with default values for our hardware, E-Cores

Core	Class	Performance	EnergyEfficiency
0	0	86	37
0	1	105	38
0	2	197	53
0	3	48	21
…	…	…	…
6	0	88	37
6	1	107	38
6	2	200	53
6	3	49	21
7	0	88	37
7	1	107	38
7	2	200	53
7	3	49	21

Figure 29: Table for the 30 W_4T_HS_SH workload for P-Cores after two iterations. Here, Core 0-5 reported identical values.

Core	Class	Performance	EnergyEfficiency
8	0	64	63
8	1	64	49
8	2	64	38
8	3	46	45
…	…	…	…
12	0	64	65
12	1	64	51
12	2	64	40
12	3	46	46
…	…	…	…
16	0	64	155
16	1	64	122
16	2	64	95
16	3	46	111
…	…	…	…

Figure 30: Table for the 30 W_4T_HS_SH workload for E-Cores after two iterations.

Bibliography

[1] M. Pawlish, A. S. Varde, S. A. Robila, and A. Ranganathan, "A call for energy efficiency in data centers," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 45–51, 2014.

[2] R. Muralidhar, R. Borovica-Gajic, and R. Buyya, "Energy efficient computing systems: Architectures, abstractions and modeling to techniques and standards," *ACM Comput. Surv.*, vol. 54, no. 11s, 2022-09, ISSN: 0360-0300. DOI: 10.1145/3511094. [Online]. Available: https://doi.org/10.1145/3511094.

[3] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007. DOI: 10.1109/N-SSC.2007.4785534.

[4] Arm Holdings, *Arm unveils its most energy efficient application processor ever; redefines traditional power and performance relationship with big.little processing*, https://web.archive.org/web/20120603021524/http://www.arm.com/about/newsroom/arm-unveils-its-most-energy-efficient-application-processor-ever-with-biglittle-processing.php, Archived; Online; Accessed: 19. November, 2025.

[5] Apple Corporation, *Apple unleashes m1*, https://www.apple.com/de/newsroom/2020/11/apple-unleashes-m1/, Online; Accessed: 19. November, 2025.

[6] D. I. Cutress, *Intel alder lake: Confirmed x86 hybrid with golden cove and gracemont for 2021*, https://web.archive.org/web/20200814201629/https://www.anandtech.com/show/15979/intel-alder-lake-confirmed-x86-hybrid-with-golden-cove-and-gracemont-for-2021, Archived; Online; Accessed: 19. November, 2025.

[7] D. Patel and G. Wong, *Zen 4c: Amd's response to hyperscale arm & intel atom*, https://newsletter.semianalysis.com/p/zen-4c-amds-response-to-hyperscale, Online; Accessed: 19. November, 2025.

[8] *Intel 64 and ia-32 architectures software developer's manual volumes 1-4*, Intel Corporation, 2025.

[9] M. Larabel, *Amd hardware feedback interface "HFI" driver updated for heterogeneous cpus*, https://www.phoronix.com/news/AMD-HFI-Linux-Driver-v2, Online; Accessed: 17. November, 2025.

[10] E. Rotem et al., "Intel alder lake cpu architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19,

[11] R. Neri, *Thermal: Intel: Hfi: Enable the intel thread director*, https://lore.kernel.org/lkml/20230613042422.5344-18-ricardo.neri-calderon\spacefactor\@m{}linux.intel.com/, Online; Accessed: 19. November, 2025.

[12] Intel Corporation, *Monitor and umonitor performance guidance*, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/monitor-umonitor-performance-guidance.html, Online; Accessed: 11. March, 2026.

[13] J. Stoess, "Towards effective user-controlled scheduling for microkernel-based systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 4, pp. 59–68, 2007.

[14] J. Liedtke, "Improving IPC by kernel design," in *Proceedings of the fourteenth ACM Symposium on Operating systems principles (SOSP)*, ACM Press, 1993-12, pp. 175–188.

[15] J. Liedtke, "On μ-kernel construction," in *Proceedings of the fifteenth ACM Symposium on Operating systems principles (SOSP)*, ACM Press, 1995-12, pp. 237–250.

[16] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, no. 9, pp. 70–77, 1996-09.

[17] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-based systems," in *Proceedings of the sixteenth ACM Symposium on Operating systems principles (SOSP)*, ACM Press, 1997-10, pp. 66–77.

[18] K. Elphinstone, "Future directions in the evolution of the L4 microkernel," in *Proceedings of the NICTA workshop on OS verification 2004*, National ICT Australia, 2004-10.

[19] G. Klein et al., "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[20] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified os kernel," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 471–482.

[21] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–16.

[22] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," *ACM Transactions on Computer Systems (TOCS)*, vol. 34, no. 1, pp. 1–29, 2016.

[23] *Sel4 reference manual version 13.0.0*, The seL4 Foundation, 2024.

[24] The seL4 Foundation, *Supported platforms*, https://docs.sel4.systems/Hardware/, Online; Accessed: 19. November, 2025.

[25] A. Shan, *Heterogeneous processing: A strategy for augmenting moore's law*, https://www.linuxjournal.com/article/8368, Online; Accessed: 25. November, 2025.

[26] B. Rau, D. Yen, W. Yen, and R. Towle, "The cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs," *Computer*, vol. 22, no. 1, pp. 12–35, 1989.

[27] D. I. Cutress and A. Frumusanu, *Intel disabled avx-512, but not really*, https://web.archive.org/web/20211104154046/https://www.anandtech.com/show/17047/the-intel-12th-gen-core-i912900k-review-hybrid-performance-brings-hybrid-complexity, Archived; Online; Accessed: 19. November, 2025.

[28] I. Cutress and A. Frumusanu, *Gracemont microarchitecture (e-core) examined*, https://web.archive.org/web/20250615210819/https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/4, Online; Accessed: 26. November, 2025.

[29] G. Wathan, *Where does big.little fit in the world of dynamiq?* https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/where-does-big-little-fit-in-the-world-of-dynamiq, Online; Accessed: 25. November, 2025.

[30] A. Frumusanu, *The exynos 9820 soc - a new tri-cpu design*, https://web.archive.org/web/20190807123540/https://www.anandtech.com/show/14072/the-samsung-galaxy-s10plus-review/4, Online; Accessed: 26. November, 2025.

[31] M. Shedd, *Snapdragon 820 and kryo cpu: Heterogeneous computing and the role of custom compute*, https://www.qualcomm.com/news/onq/2015/09/snapdragon-820-and-kryo-cpu-heterogeneous-computing-and-role-custom-compute, Online; Accessed: 26. November, 2025.

[32] *Intel architecture instruction set extensions and future features*, Intel Corporation, 2021.

[33] *Intel architecture instruction set extensions and future features*, Intel Corporation, 2025.

[34] I. Chun, I. Siu, and R. Paccagnella, "Scheduled disclosure: Turning power into timing without frequency scaling," in *2025 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2025, pp. 3617–3635.

[35] *Amd64 architecture programmer's manual volumes 1-5*, Advanced Micro Devices, Inc., 2024.

[36] M.-C. Kuns, H. Tröpgen, and R. Schöne, "An analysis of user-space idle state instructions on x86 processors," in *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, 2025, pp. 232–239.

[37] R. Schöne, T. Ilsche, M. Bielert, M. Velten, M. Schmidl, and D. Hackenberg, "Energy efficiency aspects of the amd zen 2 architecture," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 562–571.

[38] C. Gough, I. Steiner, and W. Saunders, *Energy efficient servers: blueprints for data center optimization*. Springer Nature, 2015.

[39] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, 2018-03.

[40] D. Salomon and M. E. Russinovich, *Win2k architecture*, https://web.archive.org/web/20050323090649/http://mipagina.cantv.net/jjaguilerap/w2k_arq.html, Archived; Online; Accessed: 23. November, 2025.

[41] J. Liedtke and H. Wenske, "Lazy process switching," in *Proceedings of the eighth IEEE workshop on Hot Topics in Operating Systems (HotOS)*, IEEE Computer Society Press, 2001-05, pp. 15–20.

[42] Microsoft Corporation, *Managing authorization and access control*, https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457115(v=technet.10), Online; Accessed: 26. November, 2025.

[43] A. Gruenbacher, *Acl(5) file formats manual*, 2002-03.

[44] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, 1988-10.

[45] T. Close, "Acls don't," HP Laboratories Technical Report, Tech. Rep., 2009.

[46] The Linux Foundation, *The definitive kvm (kernel-based virtual machine) api documentation*, `https://docs.kernel.org/virt/kvm/api.html`, Online; Accessed: 17. March, 2026.

[47] Intel Corporation, *Errata details*, `https://edc.intel.com/content/www/us/en/design/products/platforms/details/arrow-lake-s/core-ultra-200s-series-processors-specification-update/errata-details/`, Online; Accessed: 20. March, 2026.

[48] J. C. Saez and M. Prieto-Matias, "Evaluation of the intel thread director technology on an alder lake processor," in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2022, pp. 61–67.

[49] C. Bilbao, J. C. Saez, and M. Prieto-Matias, "Flexible system software scheduling for asymmetric multicore systems with pmcsched: A case for intel alder lake," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 25, e7814, 2023.

[50] T. Smejkal, R. Khasanov, J. Castrillon, and H. Härtig, "Harp: Energy-aware and adaptive management of heterogeneous processors," in *Proceedings of the 26th International Middleware Conference*, ser. Middleware '25, Vanderbilt University, Nashville, TN, USA: Association for Computing Machinery, 2025, pp. 270–284, ISBN: 9798400715549. DOI: `10.1145/3721462.3770774`. [Online]. Available: `https://doi.org/10.1145/3721462.3770774`.

[51] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne:{core-aware} thread management," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 145–160.

[52] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high {cpu} efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 361–378.

[53] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 345–360.

[54] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 281–297.

[55] J. T. Humphries et al., "Ghost: Fast & flexible user-space delegation of linux scheduling," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 588–604.

[56] Y. Jia, K. Tian, Y. You, Y. Chen, and K. Chen, "Skyloft: A general high-efficient scheduling framework in user space," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*, 2024, pp. 265–279.

[57] Y. Li et al., "Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 922–936.

[58] S. Miller, A. Kumar, T. Vakharia, A. Chen, D. Zhuo, and T. Anderson, "Enoki: High velocity linux kernel scheduler development," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 962–980.

[59] J. Fried et al., "Making kernel bypass practical for the cloud with junction," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 55–73.

[60] L. Guo, D. Zuberi, T. Garfinkel, and A. Ousterhout, "The benefits and limitations of user interrupts for preemptive userspace scheduling," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 1015–1032.

[61] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," in *Proceedings of the thirteenth ACM Symposium on Operating systems principles (SOSP)*, ACM Press, 1991-10, pp. 95–109.

[62] sched-ext, *Sched_ext schedulers and tools*, `https://github.com/sched-ext/scx`, Online; Accessed: 19. November, 2025.

[63] E. Li et al., "Spdk+: Low latency or high power efficiency? we take both," in *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*, 2025, pp. 17–23.

[64] Y. Zhou, E. Xu, A. Su, J. Harris, A. Manzanares, and S. Swanson, "Sleeping with one eye open: Fast, sustainable storage with sandman," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, 2025, pp. 496–511.

[65] C. Li et al., "Aeolia: A fast and secure userspace interrupt-based storage stack," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, 2025, pp. 479–495.

[66] M. Åsberg and T. Nolte, "Towards a user-mode approach to partitioned scheduling in the sel4 microkernel," *ACM Sigbed Review*, vol. 10, no. 3, pp. 15–22, 2013.

[67] K. McLeod, *Sel4 multikernel ipi api*, https://sel4.atlassian.net/browse/RFC-17, Online; Accessed: 5. March, 2026.

[68] D. Greenaway, *Syscall_stub_gen.py*, https://github.com/seL4/seL4/blob/master/libsel4/tools/syscall_stub_gen.py, Online; Accessed: 9. March, 2026.

[69] The Linux Foundation, *Cpu hotplug in the kernel*, https://docs.kernel.org/core-api/cpu_hotplug.html, Online; Accessed: 20. March, 2026.

[70] The seL4 Foundation, *Constants.h*, https://github.com/seL4/seL4/blob/efd9426e1771b6cdbcc7e58d3fd58d3df77c337e/libsel4/include/sel4/constants.h, Online; Accessed: 6. March, 2026.

[71] The seL4 Foundation, *14.0.0*, https://docs.sel4.systems/releases/sel4/14.0.0.html, Online; Accessed: 6. March, 2026.

[72] The Linux Foundation, *Uaccess.h*, https://github.com/torvalds/linux/blob/c612261bedd6bbab7109f798715e449c9d20ff2f/include/linux/uaccess.h#L228, Online; Accessed: 20. March, 2026.

[73] Microsoft Corporation, *Processor groups*, https://learn.microsoft.com/en-us/windows/win32/procthread/processor-groups, Online; Accessed: 17. March, 2026.

[74] The seL4 Foundation, *Ipi.h*, https://github.com/seL4/seL4/blob/master/include/smp/ipi.h, Online; Accessed: 9. March, 2026.

[75] The seL4 Foundation, *Rust support for sel4 userspace*, https://github.com/seL4/rust-sel4, Online; Accessed: 24. February, 2026.

[76] The seL4 Foundation, *Sel4 microkit*, https://github.com/seL4/microkit, Online; Accessed: 24. February, 2026.

[77] G. Klein, *Support mcs + smp config on imx8mm*, https://github.com/seL4/seL4/issues/929, Online; Accessed: 11. March, 2026.

[78] G. Klein, *Odroid_c4 fails sched_context_0014 on mcssmp*, https://github.com/seL4/seL4/issues/928, Online; Accessed: 11. March, 2026.

[79] G. Klein, *Boot printing leads to assertion failure on smpmcs*, https://github.com/seL4/seL4/issues/513, Online; Accessed: 11. March, 2026.

[80] Z. Liu, *Intel thread director virtualization*, https://lore.kernel.org/lkml/20240203091214.411862-1-zhao1.liu\spacefactor\@m{}linux.intel.com/, Online; Accessed: 24. February, 2026.

[81] Z. Liu, *Intel thread director virtualization support in qemu*, https://patchew.org/QEMU/20240203093054.412135-1-zhao1.liu\spacefactor\@m{}linux.intel.com/, Online; Accessed: 24. February, 2026.

[82] S. Gruszka, *Enable hfi feature only when required*, https://lore.kernel.org/all/20240131120535.933424-1-stanislaw.gruszka\spacefactor\@m{}linux.intel.com/, Online; Accessed: 24. February, 2026.

[83] The QEMU Project Developers, *Rapl msr support*, https://www.qemu.org/docs/master/specs/rapl-msr.html, Online; Accessed: 24. February, 2026.

[84] Red Hat, Inc., *Privileged rapl msr helper commands for qemu*, https://github.com/qemu/qemu/blob/afe653676dc6dfd49f0390239ff90b2f0052c2b8/tools/i386/qemu-vmsr-helper.c, Online; Accessed: 24. February, 2026.

[85] I. Z. Gerwin Klein Rafal Kolanski, *Fpu switching*, https://github.com/seL4/rfcs/blob/main/src/implemented/0180-fpu-switching.md, Online; Accessed: 17. March, 2026.

[86] The Linux Foundation, *Common.c*, https://github.com/torvalds/linux/blob/f338e77383789c0cae23ca3d48adcc5e9e137e3c/arch/x86/kernel/cpu/common.c#L2350, Online; Accessed: 16. March, 2026.

[87] The seL4 Foundation, *Machine.h*, https://github.com/seL4/seL4/blob/master/include/arch/x86/arch/machine.h, Online; Accessed: 17. March, 2026.

[88] The seL4 Foundation, *Apic.c*, https://github.com/seL4/seL4/blob/master/src/arch/x86/kernel/apic.c#L59, Online; Accessed: 17. March, 2026.

[89] The Linux Foundation, *Cfs scheduler*, https://docs.kernel.org/scheduler/sched-design-CFS.html, Online; Accessed: 17. March, 2026.

[90] R. Love, *The linux process scheduler*, https://web.archive.org/web/20180302112759/https://www.informit.com/articles/article.aspx?p=101760&seqNum=2, Online; Accessed: 16. March, 2026.

[91] H. Hagedoorn, *Confirmed again: Intel arrow lake desktop processors to launch without hyper-threading technology*, https://www.guru3d.com/story/confirmed-again-intel-arrow-lake-desktop-processors-to-launch-without-hyperthreading-technology/, Online; Accessed: 9. March, 2026.

[92] M. L. Gurres, *Applying modern processor features to l4 microkernels*, Bachelor Thesis, 2025-1017.

[93] Trustworthy Systems, *The lions operating system*, https://lionsos.org/, Online; Accessed: 29. March, 2026.

[94] C. I. King, *Stress next generation*, https://github.com/ColinIanKing/stress-ng, Online; Accessed: 9. March, 2026.

[95] J. Furgala, S. Jero, A. Lin, and R. Skowyra, "Porting nasa's core flight system to the formally verified sel4 microkernel," *Workshop on Security of Space and Satellite Systems (SpaceSec)*, 2026. DOI: https://dx.doi.org/10.14722/spacesec.2026.23003.

[96] The Linux Foundation, *Power capping framework*, https://docs.kernel.org/power/powercap/powercap.html, Online; Accessed: 11. March, 2026.

[97] The Linux Foundation, *Kernel parameter*, https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html, Online; Accessed: 26. March, 2026.

[98] *Intel 64 and ia-32 architectures optimization reference manual: Volume 1*, Intel Corporation, 2024.

[99] J. Corbet, *User-space interrupts*, https://lwn.net/Articles/871113/, Online; Accessed: 24. March, 2026.

[100] The seL4 Foundation, *Hello, world!* https://docs.sel4.systems/Tutorials/hello-world.html, Online; Accessed: 5. March, 2026.