School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Core Management in LionsOS

by

Michael Mospan

Thesis submitted as a requirement for the degree of

Bachelor of Advanced Computer Science (Honours)

Submitted: December 2025

Supervisor: Prof. Gernot Heiser

Student ID: z5419614

Abstract

Energy efficiency has become a first-order concern across all classes of computing systems. As transistor densities continue to increase and workloads grow more complex, careful management of energy use is especially important for battery powered and thermally constrained embedded platforms. In parallel, mainstream processors have shifted towards multicore designs that deliver performance through parallelism, yet can still draw substantial energy even when many cores are lightly loaded or idle.

This thesis investigates the design and implementation of an intelligent core management system for LionsOS that dynamically consolidates work onto fewer cores and parks unused cores in deep idle states, with the aim of reducing CPU energy use during periods of low computational demand.

Acknowledgements

I would like to express my sincere gratitude to everyone at Trustworthy Systems for their work on the seL4 Microkit and LionsOS. Their years of engineering effort, careful analysis, and sleepless nights provided the foundations on which this thesis is built. In particular, I am deeply grateful to my supervisor, Professor Gernot Heiser, and my assessor, Dr Peter Chubb, for their guidance, feedback, and encouragement throughout this project.

I would also like to thank Ivan Velickovic, Julia Vassiliki, and Krishnan Winter for their generous help and support in the design and implementation of core management in LionsOS. My thanks as well to Lesley Rossouw for her assistance with the evaluation and interpretation of the core management energy benchmarking results.

Finally, and most importantly, I thank my friends and family for their constant support, patience, and belief in me throughout this thesis and my time at UNSW.

Abbreviations

ACPI Advanced Configuration and Power Interface

API Application Programming Interface

CMP Chip Multiprocessor

CPU Central Processing Unit

DVFS Dynamic Voltage and Frequency Scaling

ECALL Environment Call

EID Extension ID

ELF Executable and Linkable Format

FID Function ID

IPI Inter-Processor Interrupt

ITTD Intelligent Timer Tick Distribution

MCS Mixed-Criticality System

MMU Memory Management Unit

MR Memory Region

OS Operating System

PD Protection Domain

PPC Protected Procedure Call

PPM Processor Power Management

PSCI Power State Coordination Interface

RPC Remote Procedure Call

SDF System Description File

SMC	Secure Monitor Call
SMP	Symmetric Multiprocessing
TCB	Trusted Computing Base
SBI	Supervisor Binary Interface
sDDF	seL4 Device Driver Framework

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern computing increasingly relies on multi-core processors to deliver high performance through parallelism. However, these systems often consume significant energy even when underutilised. This thesis explores the development of an intelligent core management system for LionsOS that minimises central processing unit (CPU) energy consumption during periods of low computational demand.

1.1 The Problem

Energy consumption has emerged as a critical concern across all computing platforms. As transistor densities climb and workloads intensify, efficient power management becomes vital — particularly in extending the life of battery powered devices such as embedded systems.

1.1.1 Keeping all cores active on a multi-processor wastes energy

With the rapid increase in embedded system power draw comes the urgent need for energy-conscious design at all levels of computing infrastructure.

Modern embedded systems are built around multi-core processors, which account for the bulk of a system's dynamic power draw. Yet, even an idle embedded system can utilise up to 40% of its peak power [Lorenzon, Cera, and Beck 2015], making low-utilisation embedded systems extraordinarily inefficient from an energy perspective. Furthermore, the heat generated by under-loaded cores necessitates robust cooling infrastructures, which themselves incur significant additional energy overhead, creating a compounding inefficiency problem.

Figure 1.1 illustrates the relationship between CPU power draw and the number of active cores. With correlation coefficients exceeding 0.9, this graph demonstrates a strong, positive linear relationship, underscoring the potential energy savings from strategic core management.

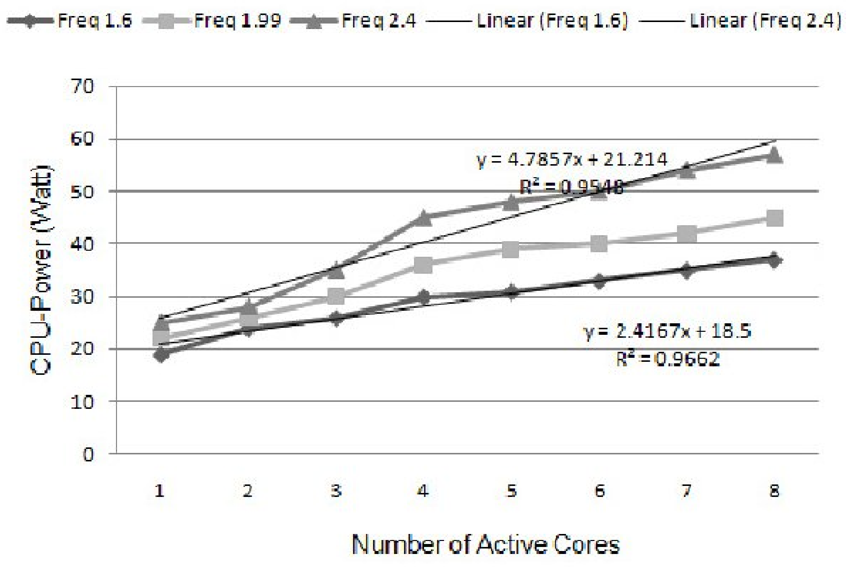


Figure 1.1: PU power draw relationship with number of active cores [Takouna, Dawoud, and Meinel 2011].

The fundamental inefficiency stems from a key characteristic of modern processors: each core consumes energy, whether performing useful computation or merely idling — unless it enters a true "deep-sleep" state. Static power draw represents a significant portion of overall system consumption during idle periods [Carroll and Heiser 2010] and can only be effectively addressed through deep-sleep states that power-gate sections of the chip.

1.1.2 Local kernel core management strategies are energy inefficient

Many existing core-management solutions employ local strategies that work on a per-core basis. As we explore further in Chapter 3, these approaches often prove inefficient due to microprocessor heterogeneity — whether by design or caused by manufacturing defects — and the difficulty of specifying and distributing a chip-wide power budget across all cores in a chip-multiprocessor (CMP) system [Sharkey, Buyuktosunoglu, and Bose 2007, J. A. Winter, Albonesi, and Shoemaker 2010].

Furthermore, these solutions typically reside within the operating system's trusted computing base (TCB) and run in privileged mode, making it impractical to develop custom core-management algorithms at runtime [Palmur, Z. Li, and Zadok 2013]. Moreover, power-gated (sleep) states — capable of significantly reducing energy consumption — are only entered when cores remain idle for extended periods. In today's OSs, however, stray interrupts and kernel threads often keep cores active even when no application code is executing [Bortolotti, Tinti, Altoé, and Bartolini 2016].

In contrast, accelerator-equipped computing nodes experience phases of computation and communication that are poorly balanced for parallel scaling, creating idle slack within the execution flow. Current OS power-management infrastructures cannot detect or exploit this slack [Bortolotti, Tinti, Altoé, and Bartolini 2016].

To address these limitations, we investigate the design of a global power manager that dynamically redistributes protection domains among processor cores and thereby maximising power efficiency under changing workloads. We also analyse the trade-offs inherent in this global approach compared to purely local techniques.

1.2 The approach to energy efficiency

To address CPU energy consumption challenges, processor manufacturers have implemented various power-management mechanisms. Two predominant approaches exist for reducing energy consumption in multi-core processors:

1. Dynamic voltage and frequency scaling (DVFS)
2. Low-power (idle) states

DVFS enables the OS to dynamically reduce processor frequency and voltage during periods of low computational demand, thereby lowering energy consumption at the cost of reduced performance. Alternatively, low-power states allow CPU cores to operate at peak performance to complete tasks quickly and then transition into deep sleep states, dramatically reducing energy consumption during idle periods. This latter strategy, often referred to as *race-to-halt*, prioritises task completion speed to maximise idle time.

Despite both mechanisms being widely implemented, recent research suggests that DVFS has become less effective on newer processor architectures and can even increase total energy use [Le Sueur 2011, Le Sueur and Heiser 2010], including for memory-bound workloads where a reduced CPU frequency should theoretically have no effect on performance. This view is reinforced by a newer study on energy models for data centres, which found that DVFS rarely reduces the energy consumption of under-loaded servers by more than 5%, while still causing performance degradation. In contrast, low-power states were able to reduce the energy use of under-loaded servers by up to 7%, with performance losses kept under control through suitable heuristics [Krzywda, Ali-Eldin, Carlson, Östberg, and Elmroth 2017].

Consequently, while acknowledging the potential role of DVFS, this thesis primarily focuses on strategic core management and the optimal utilisation of low-power states to minimise energy consumption. The interplay between these approaches and potential integration with DVFS strategies represents a promising direction for future work.

1.3 The Solution

Rather than attempting to patch the fundamental shortcomings of today's core management schemes, this thesis proposes a global, high-performance core manager built on top of a minimal microkernel. This core manager should:

1. Utilise low-power states to power-gate various components of the CPU when they are not needed, minimising energy use
2. Maintain a global view of all CPU cores in the system so it can make optimal decisions
3. Remain outside the kernel and operate in its own address space for security and ease of extensibility

By design, the microkernel exposes only the bare essentials required to securely abstract hardware, keeping our core manager out of the kernel's trusted computing base and in user space, where it can be tailored or replaced on a per-user basis without modifying the TCB.

In Chapter 2, we introduce the seL4 microkernel and its companion Microkit layer, which together provide the lightweight abstractions that underpin our core manager. Once we finalise the API and implement our initial policy set, we port the entire system to LionsOS — a secure operating system built on top of seL4 and Microkit — to demonstrate real-world integration.

The core manager itself — detailed in Chapter 4 — runs as its own protection domain (PD), similar to a UNIX process in LionsOS, effectively functioning as a privileged background service with full visibility over system resources. It performs rapid, system-wide power distribution decisions, migrates protection domains between CPU cores, and places idle cores into deep-sleep states using the heuristics from Section 4.7. Crucially, it also exposes a clean, extensible API, enabling future developers to introduce bespoke core management policies and build on this solid foundation.

Chapter 2

Background

There have been numerous efforts to address the energy inefficiencies that arise from underutilised multi-core processors. To understand these solutions, assess their limitations, and draw our own conclusions, we begin by examining the fundamental power-management mechanisms provided by processor manufacturers — particularly idle state management. Next, we introduce seL4, the seL4 Microkit, and LionsOS, laying the groundwork for Chapter 4, where we show how these components can be integrated to implement a global core manager.

2.1 Processor core power management

With the growing emphasis on managing energy consumption, modern CPUs have incorporated various power-management mechanisms, some of which enable the OS to balance performance with reduced power usage. Although a detailed comparison between DVFS and idle-state management falls outside the scope of this thesis, Section 2.1.1 provides an overview of these two mechanisms and their potential impact on energy consumption.

2.1.1 To slow down or to sleep?

Given that a processor's total power P, is a function of both dynamic and static components, it can be expressed as

$$\begin{aligned} P &= P_{dynamic} + P_{static} \\ &= \alpha C f V^2 + V I_{leakage} \end{aligned}$$

where α is the activity factor (the fraction of gates switching per cycle), C is the capacitance of the transistor gates (which depends on feature size), f is the operating frequency, and V is the supply voltage [Al-Khalili 2015, Le Sueur 2011, Le Sueur and

Heiser 2010]. Importantly, $I_{leakage}$ is the unwanted current that flows through transistors even when they are in the off state, representing imperfections in the transistor's ability to completely block current flow [Roy, Mukhopadhyay, and Mahmoodi-Meimand 2003].

As a result, we can state various proportionality ($\propto$) properties, such as

$$P_{dynamic} \propto fV^2 \tag{2.1}$$

and

$$P_{static} = VI_{leakage}$$

allowing us to draw several key conclusions.

Because software alone cannot directly reduce the intrinsic leakage current I_{leakage}, we instead lower the operating frequency f and supply voltage V. Dynamic voltage and frequency scaling is one of the most common mechanisms for achieving this: by reducing the processor's clock rate, we can drop the voltage to the minimum level required for stable operation [Le Sueur and Heiser 2010]. Thanks to the quadratic relationship in Equation 2.1, even modest voltage reductions yield disproportionately large energy savings.

However, this strategy only reduces *dynamic* power; *static* (leakage) power remains largely unaffected, since the minimum voltage required for stability scales linearly with frequency. Furthermore, lowering the frequency also degrades processor performance, minimising potential energy savings — especially on modern cores, where the voltage-scaling window is very narrow [Le Sueur and Heiser 2011, Le Sueur 2011, Le Sueur and Heiser 2010]. This is also shown in Figure 2.1 where decreasing the CPU frequency of a server running an underloaded application reduces the power draw by only approximately 5%, where they claim this is because lower frequency increases the time needed to process each request, and therefore the CPU spends more time in an active state [Krzywda, Ali-Eldin, Carlson, Östberg, and Elmroth 2017]. As a result, DVFS cannot match the energy savings achieved through aggressive power-gating in deep-sleep idle states.

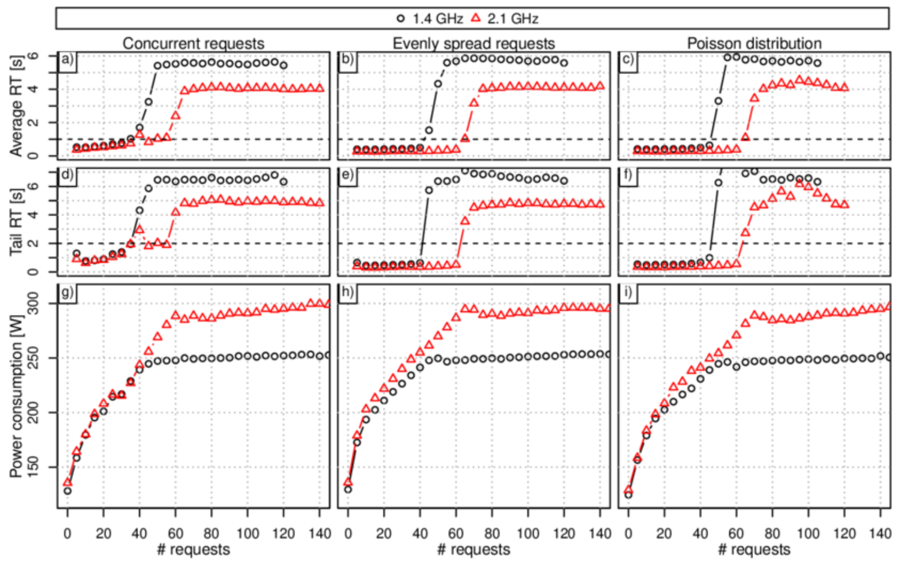


Figure 2.1: Impact of DVFS on response time (RT) and average electrical power for AMD servers [Krzywda, Ali-Eldin, Carlson, Östberg, and Elmroth 2017].

When an OS has no tasks to schedule on a core, it will usually invoke the idle thread which puts that core into an idle state. As opposed to reducing the frequency and then matching that frequency with a voltage, idle states power-gate the processor core, effectively disconnecting it from the power supply. This approach eliminates both dynamic and static energy consumption for that core. Modern processors implement a hierarchy of idle states with progressively deeper energy savings and longer entry and exit latencies. Figure 2.2 demonstrates the possible idle state transitions for an Intel Core i7 processor.

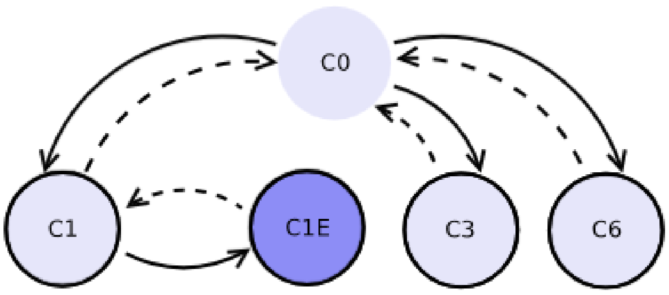


Figure 2.2: Possible idle state transitions for an Intel Core i7 processor.

Shallower idle states like C1 might simply halt the CPU clock while maintaining voltage, whereas deeper states like C6 can shut down caches, memory controllers, and other subsystems. The OS power management subsystem must carefully balance the potential energy savings against the latency cost of entering and exiting these states, especially for workloads with intermittent activity patterns. Whenever a CPU core is put into a lower power state and wants to return into an active state, the time taken to do so is referred to as the “exit latency”. The deeper the low-power state a core is in, the higher its exit latency becomes.

From Table 2.1 we can see the typical power draw and corresponding exit latency for an Intel Xeon 5,600 processor [Palmur, Z. Li, and Zadok 2013].

C-State	**Power Draw**	**Exit Latency**
C0 (Active)	80W	0 μs
C1	40W	3 μs
C3	33W	20 μs
C6	12W	200 μs

Table 2.1: Power draw and exit latency of Intel Xeon 5,600 series processor states.

These low-power or idle states are also commonly referred to as C-states.

2.1.2 Why not always turn the CPU core completely off?

Unlike completely powering a core off, most C-states only power-gate portions of the processor core, with the exception of the deepest C-state. Just as exiting from a deeper low-power state incurs a longer latency, powering a CPU core completely off and then back on imposes an even greater delay. Complete core shutdown produces the highest exit latency by a wide margin: whereas a typical low-power C-state entry–exit cycle takes tens to hundreds of microseconds, Carroll measured up to 22.6 ms to restart a single Arm core after full power-off [Carroll 2017].

If the idle interval is short, powering a core off completely can take hundreds or even thousands of times longer than a shallow C-state transition. In systems with strict performance or real-time requirements — such as mixed-criticality environments — that additional wait time may be unacceptable.

2.2 Heterogeneous Architectures

Heterogeneous processors use different types of cores to perform various tasks, optimising both performance and efficiency. This contrasts with homogeneous systems, which rely on a single type of processor. As the semiconductor industry continues to increase transistor density, enabling hundreds of cores on a single chip, these "many-core" processors are now commonly designed to be heterogeneous. In many cases, such designs deliver orders of magnitude improvements in energy efficiency and performance compared with homogeneous chips, making them a fundamental component of modern computing systems ranging from embedded platforms to supercomputers [Fang, Huang, T. Tang, and Z. Wang 2020]. As the number of on-chip cores continues to scale, achieving power–performance efficiency becomes an increasingly complex optimisation challenge for the runtime manager.

2.2.1 Arm big.LITTLE

One example of a heterogeneous system is Arm big.LITTLE. This system uses two types of cores in a coherent multicore architecture: big cores optimised for high performance and little cores optimised for energy efficiency. Figure 2.3 illustrates a typical big.LITTLE configuration with varied clusters.

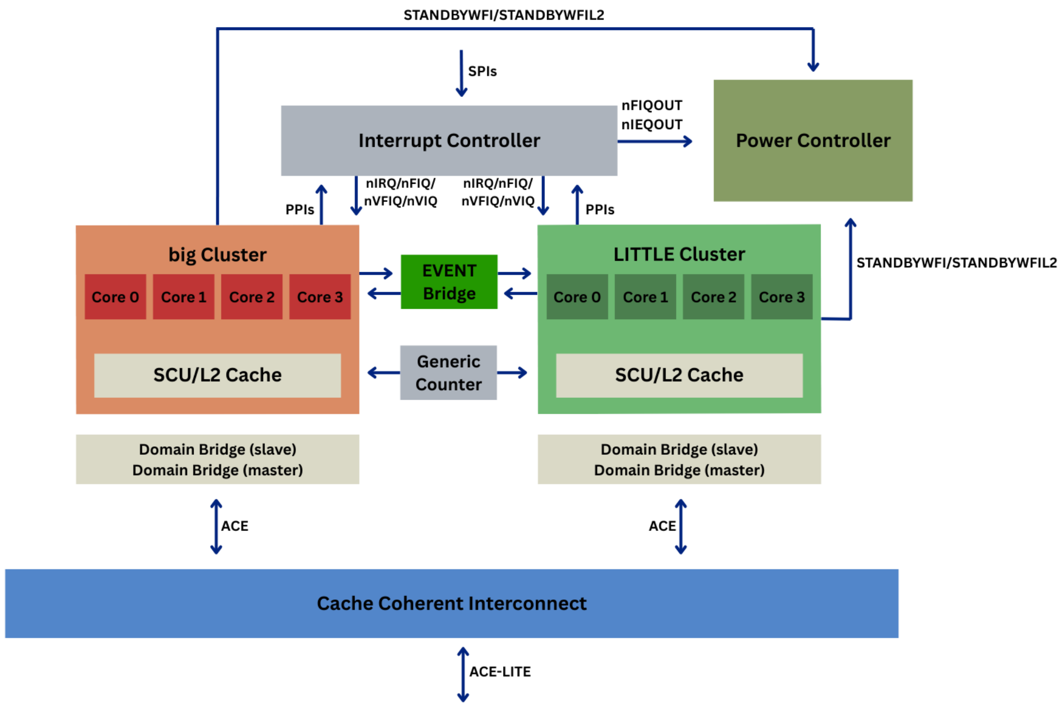


Figure 2.3: Typical Arm big.LITTLE architecture with distinct big and LITTLE clusters [Arm Limited 2025b].

By intelligently migrating protection domains between clusters without modifying the processor's low-power states, and simply directing tasks to either a big or little cluster, significant energy savings can be realised. Arm's measurements on an example platform demonstrate that dynamically moving threads between big and little cores as their workload demands shift yields noticeable energy saving gains (Figure 2.4).

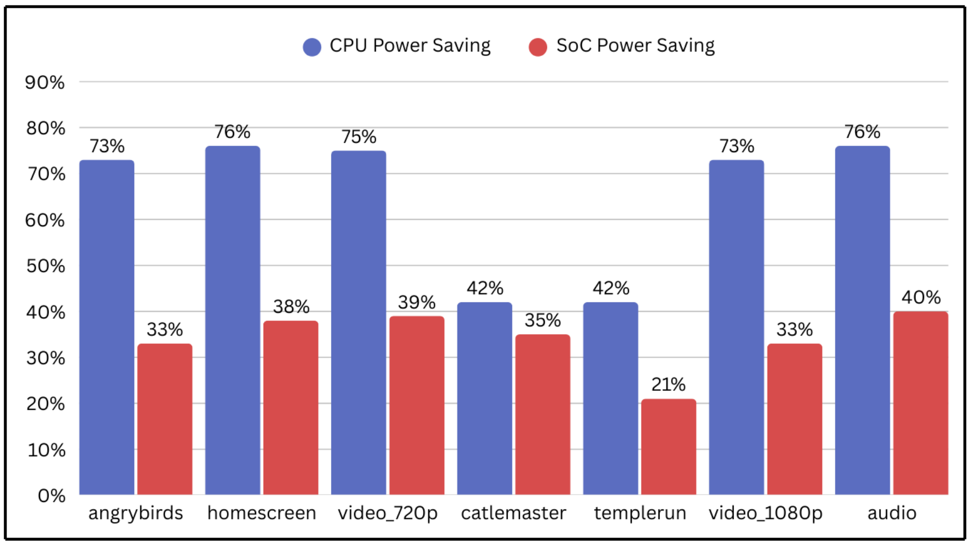


Figure 2.4: Energy savings when running on two big cores only, compared to dynamically switching between two big and two little cores [Arm Limited 2025b].

Recently, Arm extended the concept of processor heterogeneity beyond merely placing different cores in separate clusters to include mixing core types within the same cluster, a technology it calls DynamIQ. These mixed-core clusters can contain up to eight heterogeneous cores that share a single L3 cache and coherent interconnect [Arm Limited 2025c]. As a result, migration latencies decrease and energy savings improve, with single-threaded performance rising by over 40% [Singh 2018]. With the growing popularity of such architectures, global core-management solutions are essential to account for the varying power and performance characteristics of each core.

2.2.2 NUMA and Memory Heterogeneity

Many multi-core processors do not exhibit uniform memory access. Non-Uniform Memory Access (NUMA) is beneficial for workloads with high memory locality of reference and low lock contention, as processors can operate on data mostly or entirely within their own cache node, reducing traffic on the memory bus. However, this means the cores on which our tasks run have significant implications for both performance and energy consumption.

This is also the case with memory heterogeneous systems such as the Intel Xeon Phi processor (codenamed Knights Landing). This processor uses on-package Hybrid Memory Cubes (HMC), which Intel calls MCDRAM, with up to 16GB capacity [Park, Rho, Kim, and Nam 2019]. With the HMC market expected to experience exponential growth in the next few years — projected at a compound annual growth rate of 24.5% until 2029 [The Business Research Company 2025] — future core managers need to be aware of the global hardware and memory resources within their system to make optimal scheduling decisions.

2.3 Power management standards and interfaces

Modern CPU architectures provide standardised interfaces to coordinate low-power modes across hardware and firmware. In the following subsections, we review three key frameworks: Arm's Power State Coordination Interface (PSCI), the x86 Advanced Configuration and Power Interface (ACPI) C-state model, and the RISC-V Supervisor Binary Interface (SBI). In this section we aim to highlight how each defines and controls processor idle and suspend states.

2.3.1 Power state coordination modes

The Arm PSCI, RISC-V SBI, and x86 ACPI define two primary power-state coordination modes: platform-coordinated and OS-initiated. In both modes, the operating system decides when an individual core may enter an idle or low-power state. The key difference is who chooses the composite power state for higher-level topology nodes such as clusters. By default, the system operates in platform-coordinated mode, where the platform firmware derives the deepest permissible power state for each topology node from its children's requests and implementation-specific constraints.

In OS-initiated mode, the operating system also selects the target power state for the affected topology nodes and passes this choice to the platform, which validates and applies the request. The OS can, for example, request that a cluster-level node enter a low-power state once all of its child cores are idle. This mode offers better scalability and precision especially in multi-cluster SMP systems or heterogeneous big.LITTLE configurations, because the OS knows each core's next wake-up event and can finely balance entry, exit, and wake-up latencies when choosing composite power states [Shah and W. Li 2023].

A further advantage of OS-initiated mode concerns so-called last man activity: OS-level housekeeping that should occur when the last core in a power domain is about to enter a low-power state (for example, migrating cluster-local interrupts and timers to another domain before the cluster powers down). In platform-coordinated mode, the platform decides internally when the last core at a given power level becomes idle and chooses the composite state for the domain. The OS only sees independent per-core idle transitions and therefore cannot easily tie its last man activity to the precise moment when the last core goes idle without implementation-specific callbacks or side channels from the platform.

In OS-initiated mode, the OS already tracks how many cores in each power domain are active in order to select an appropriate composite power state. When this count drops to zero, the OS knows it is about to idle the last core in that domain. It can then perform its last man activity (such as redistributing cluster-local interrupts and timers, or updating OS-managed data structures) immediately before issuing the PSCI/SBI/ACPI request. This removes the need for non-standard side channels and relies solely on the standard, well-documented API between the OS and the platform.

Figure 2.5 illustrates an example power-domain topology from the Arm PSCI handbook, with each node representing an exclusive or shared power domain.

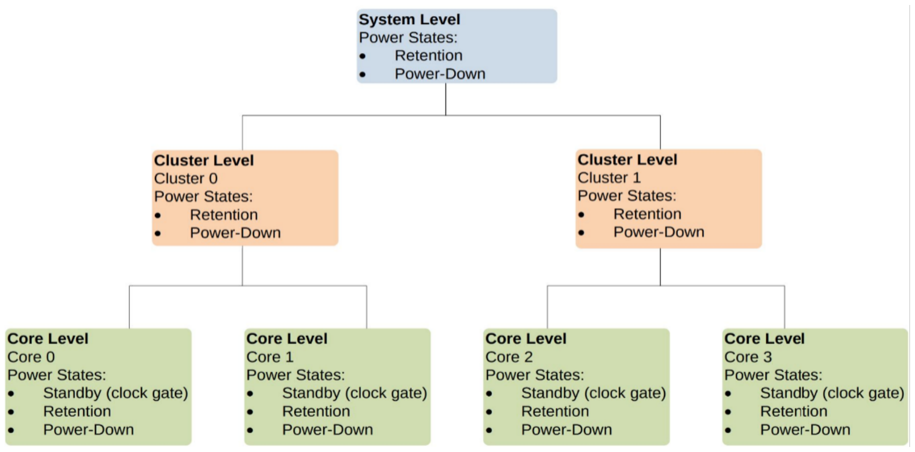


Figure 2.5: Example power-domain topology [Arm Limited 2025b].

However, OS-initiated mode introduces challenges — particularly race conditions — when the OS's view of the system state diverges from the hardware implementation's view. For instance, the OS might request a power-down for a node, while the hardware observes another core in that node powering up simultaneously [Arm Limited 2025b]. Addressing these issues requires substantial engineering effort.

Furthermore, [Sudheendra and Prabhakar 2020] states that, despite OS-initiated mode providing a far greater range of control over the hardware's power usage, it can come with the drawbacks of higher latencies and less accurate power state observations compared to platform-coordinated mode. This is because aggregation occurs at the platform level, which is closer to the actual hardware implementation. Consequently, this thesis focuses on platform-coordinated mode, leaving OS-initiated mode for future work.

2.3.2 Arm Power State Coordination Interface

While the seL4 microkernel supports other architectures such as x86 and RISC-V, this thesis focuses on core management for Arm, which is the subject of the current progress discussed in Chapter 4. To control a CPU core's power state, Arm provides the Power State Coordination Interface (PSCI) [Arm Limited 2025b], which defines three primary functions, as shown in Table 2.2 below.

Function	**Arg 1**	**Arg 2**	**Arg 3**	**Function ID**
`CPU_SUSPEND`	`power_state`	`entry_point_address`	`context_id`	`0xC4000001`
`CPU_OFF`	—	—	—	`0x84000002`
`CPU_ON`	`target_cpu`	`entry_point_address`	`context_id`	`0xC4000003`

Table 2.2: PSCI core power-management calls and their parameters in SMC64.

These functions are invoked via a Secure Monitor Call (SMC). The SMC instruction triggers a synchronous exception that is handled by Secure Monitor code running at Exception Level 3 (EL3). Function arguments and return values are passed in registers according to the SMC calling convention. After the Secure Monitor processes the exception, control — and any resulting calls — can be forwarded to a trusted OS or another component in the secure software stack. The Arm architecture defines four privilege tiers called Exception Levels (EL0 – EL3), where EL0 is the least privileged (user applications) and EL3 is the most privileged (firmware and Secure Monitor) level. Secure Monitor handlers always execute at EL3.

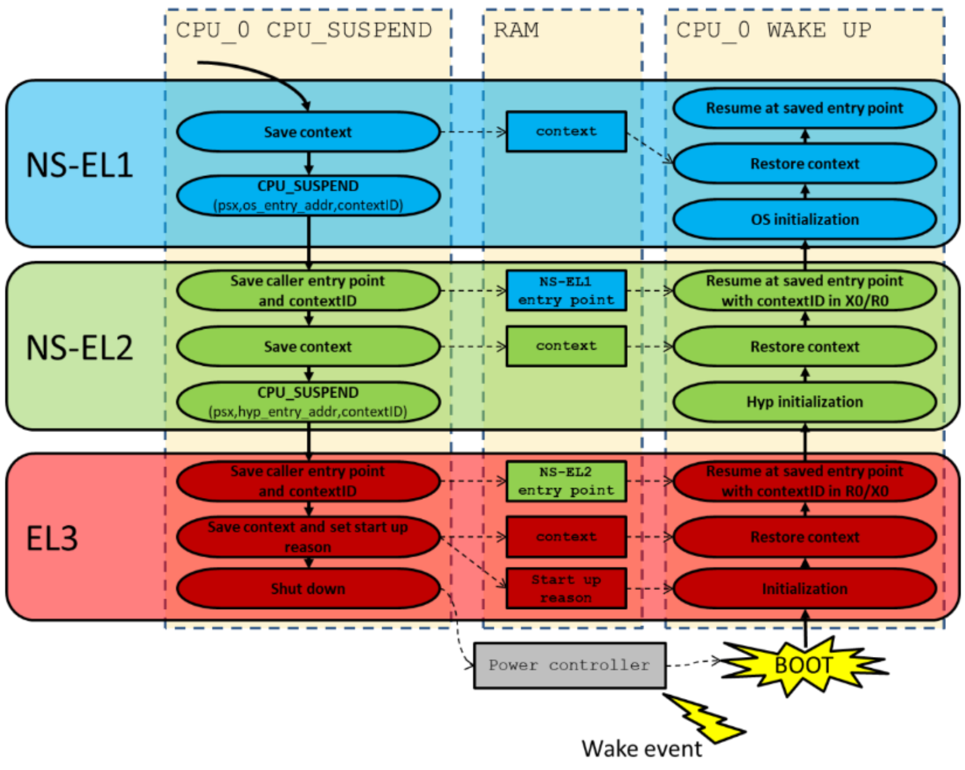
The `CPU_SUSPEND` function suspends execution of a core or higher-level topology node. It is intended for idle subsystems, where the core resumes execution upon a wake-up event.

Arm PSCI defines several power states:

1. **Run**: The core is powered and fully operational.
2. **Standby**: The core remains powered but enters a low-energy mode (typically a Wait-For-Interrupt state), clock-gating most logic until an interrupt occurs.
3. **Retention**: To the OS's idle manager, standby and retention appear identical. The distinction only matters to external debuggers and in hardware implementation details. Throughout this document, we use "standby" to refer to both standby and retention modes unless otherwise specified.
4. **Powerdown**: The core is fully powered off. Software must save all architectural state beforehand so it can be restored after power-down.

In platform-coordinate mode (see Section 2.3.1), calling `CPU_OFF` is equivalent to invoking `CPU_SUSPEND` for a power-down at the maximum power level. The difference lies in intent: `CPU_OFF` is used for hot-plugging to indicate the core will not be used again until explicitly re-onlined. Consequently, while `CPU_OFF` and `CPU_ON` are exposed in the user API, they are not automatically invoked by the background core-manager until leaving platform-coordinate mode.

Below, Figure 2.6 demonstrates an example call flow for a `CPU_SUSPEND` call, showing the platform specific instructions executed upon the suspension and wake-up of a particular processor core.



CPU_0 CPU_SUSPEND
RAM
CPU_0 WAKE UP
NS-EL1
Save context
CPU_SUSPEND (psx,os_entry_addr,contextID)
context
Resume at saved entry point
Restore context
OS initialization
NS-EL2
Save caller entry point and contextID
Save context
CPU_SUSPEND (psx,hyp_entry_addr,contextID)
NS-EL1 entry point
context
Resume at saved entry point with contextID in X0/R0
Restore context
Hyp initialization
EL3
Save caller entry point and contextID
Save context and set start up reason
Shut down
NS-EL2 entry point
context
Start up reason
Resume at saved entry point with contextID in R0/X0
Restore context
Initialization
Power controller
BOOT
Wake event

Figure 2.6: Example call flow for a `CPU_SUSPEND` call [Arm Limited 2025b].

2.3.3 RISC-V Supervisor Binary Interface Specification

Similar to the Arm PSCI, RISC-V architectures provide a Supervisor Binary Interface (SBI) specification for managing idle-states. However, unlike the Arm architecture which operates on a processor core granularity, RISC-V architecture operates on hardware threads which they call a *hart*. The Hart State Management (HSM) Extension of the SBI specification introduces a set of hart states and a set of functions which allow the supervisor-mode software to request a hart state change [RISC-V Platform Specification Task Group 2022]. The relevant function equivalents to Arm are defined in Table 2.3. Each of these function calls put the hart into either a **started**, **stopped**, or **suspended** (lower-power) state.

Function	**Arg 1**	**Arg 2**	**Arg 3**	**FID**
`sbi_hsm_hart_start`	`target_hartid`	`entry_point_address`	`priv_data`	0x0
`sbi_hsm_hart_stop`	—	—	—	0x1
`sbi_hsm_hart_suspend`	`suspend_type`	`resume_addr`	`priv_data`	0x3

Table 2.3: RISC-V SBI HSM functions and their parameters.

Each function call within the SBI specification is a combination of its function ID (FID), and its extension ID (EID). The EID is a 32-bit identifier used to specify different SBI extensions, where the EID of the HSM Extension is 0x48534D.

Unlike Arm which performs a SMC instruction to trigger power management function executions, RISC-V uses something called an Environment Call (ECALL). All SBI functions share a single binary encoding, which facilitates the mixing of SBI extensions. An ECALL is used as the control transfer instruction between the supervisor and the Supervisor Execution Environment (SEE) [Borza 2021]. Just like in Arm, these are basically requests made by a lower privileged code (user mode) to execute higher privileged code (kernel).

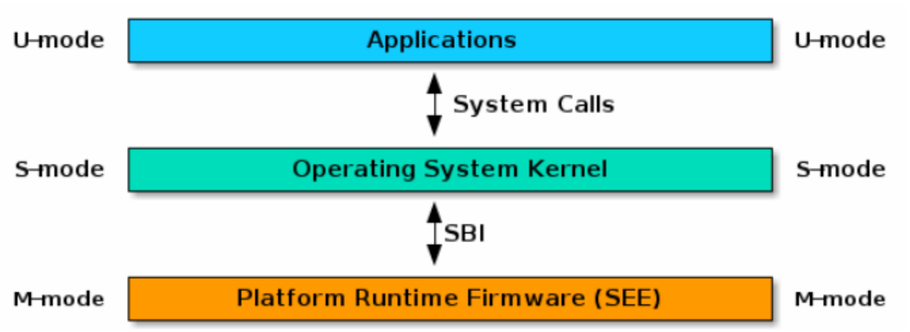


Figure 2.7: A typical RISC-V system stack [RISC-V Platform Specification Task Group 2022].

2.3.4 x86 Advanced Configuration and Power Interface

On x86 architectures, power management of processor cores is handled through a combination of the Advanced Configuration and Power Interface (ACPI), hardware-specific features, and operating system (OS) mechanisms. Unlike Arm's PSCI or RISC-V's SBI, x86 does not have a single standardised firmware interface for core power management. Instead, functionality is distributed across hardware, firmware, and OS layers.

ACPI specifies a range of low-power idle states, commonly referred to as C-states, which processor cores can enter to save energy. In addition, ACPI allows for the creation of optional objects for advanced core management. The primary mechanism for transitioning processor cores into lower power states is the `MWAIT` instruction. This instruction is used by a logical CPU to signal that it is idle, enabling the processor to potentially transition some of its functional blocks into low-power states [UEFI Forum 2024, Wysocki 2020].

The `MWAIT` instruction takes two arguments (passed in the EAX and ECX registers of the target CPU):

- **EAX (Hint):** Indicates the desired level of power-saving configuration. The specific interpretation of this hint depends on the processor model and platform configuration.
- **ECX (Extensions):** Provides additional context or flags that modify the behaviour of the idle state transition.

The mapping of `MWAIT` hint values to specific C-states is model-specific and may depend on platform configuration or ACPI tables. To determine the available idle states for a given processor, two primary sources of information can be used:

1. **Static Tables:** Predefined tables listing idle states for supported processor models.
2. **ACPI Tables:** System-specific configuration tables, used primarily for unrecognised processor models or server processors.

The combination of ACPI and `MWAIT` provides a flexible and extensible mechanism for managing CPU idle states, allowing for both energy efficiency and responsiveness in modern x86 architectures. For further details on `MWAIT` and its arguments, refer to the Intel Software Developer's Manual [Intel Corporation 2023].

2.4 The seL4 Microkernel

seL4 is an open-source, capability-based microkernel that delivers both rigorous security and high performance. A capability-based microkernel organises all access rights as unforgeable "capabilities" — small protected tokens that grant a thread the authority to invoke operations on specific kernel objects [Dennis and Van Horn 1966]. By confining every access to an explicit capability, the kernel enforces fine-grained, least-privilege policies: if a component has no capability for a resource, it cannot accidentally or maliciously touch it.

seL4 implements only the bare minimum needed to multiplex and securely abstract hardware, exposing a low-level API upon which all traditional OS services — file systems, network stacks, device drivers, and resource managers — run as unprivileged user-level servers. Every service and application must hold capabilities to access kernel objects, ensuring that no component can exceed its granted rights. This capability discipline, together with formal verification of the kernel's binary for functional correctness and confidentiality, integrity, and availability guarantees, confines any failure to the faulty component alone [Klein, Andronick, Elphinstone, Murray, Sewell, Kolanski, and Heiser 2014].

seL4 relies on explicit message passing for both inter-thread communication and interaction with the kernel [Heiser 2020]. Threads communicate by invoking capabilities, which represent the authority to access a particular communication endpoint. In practice, three system calls form the basic IPC interface:

- `seL4_Send()` transmits a message via an endpoint capability.
- `seL4_Recv()` blocks the calling thread until a message is received via the capability.
- `seL4_Yield()` voluntarily gives up the remainder of the thread's current timeslice.

These calls are typically issued by user-space threads running within their own `VSpace` kernel object, which represents a virtual address space. Possession of a capability to a `VSpace` confers the right to use that address space. All capabilities held by a thread are stored in its `CSpace`, which determines which kernel objects the thread is authorised to access [Heiser 2020]. seL4 provides two main classes of kernel objects for communication across threads: *Notifications* and *Endpoints*.

A Notification acts as a bitfield of binary semaphores. A thread signals a Notification by issuing `seL4_Send()` with a bitmask identifying which semaphores to set, and a waiting thread can block using `seL4_Recv()` until one or more selected bits are signalled. When a notification event wakes a waiting thread, the scheduler attempts to run it immediately. On multiprocessor systems, this includes cross-core notifications: if the waiting thread's affinity is on a different core, the kernel delivers the wakeup via an interprocessor interrupt (IPI) to that core, prompting its local scheduler to reschedule and, if necessary, pre-empt the current thread in favour of the newly woken one. From user space, this still appears as a pure notification send/receive; the use of IPIs is entirely hidden inside the kernel.

Endpoints provide synchronous IPC with an attached message payload. A sender calling `seL4_Send()` on an endpoint blocks until a matching receiver issues `seL4_Recv()` on the same endpoint capability. Once both sides are blocked, a rendezvous occurs and the message payload is transferred. After the transfer, both threads are unblocked. Endpoint capabilities may be restricted to send-only, receive-only, or full bidirectional access.

Although microkernel architecture can add overhead because many services interact via IPC and shared memory, leading to more frequent context switches than in monolithic kernels, seL4 nevertheless delivers competitive performance and operates within 25% of hardware-imposed limits [Mi, D. Li, Yang, X. Wang, and Chen 2019].

2.4.1 Mixed-criticality Systems

seL4 includes a mixed-criticality system (MCS) kernel variant designed to provide strong temporal guarantees, enabling seL4 to support real-time applications. This thesis focuses exclusively on the MCS kernel, as it is the configuration most commonly used with the seL4 Microkit discussed in Section 2.5.

The key feature of MCS seL4 is its scheduling model. On multiprocessor hardware, a separate instance of the MCS scheduler executes on each core and manages only the threads assigned to that core. A capability that grants authority to modify scheduling parameters on a given core is known as a *scheduling control* capability. Execution rights are conveyed by a *scheduling context*, a kernel object that defines a thread's period and budget. The budget represents the maximum execution time, measured in microseconds, that a thread may consume within each period. After the budget is exhausted, the thread is suspended until the next replenishment. At any given time, the scheduler selects the highest-priority runnable thread on that core that still has budget available. If multiple runnable threads share the same priority, they are executed in a round-robin fashion [seL4 Foundation 2025].

Understanding this kernel variant is especially important when discussing core management on top of seL4, because migrating a thread from one core to another requires updating its state by invoking operations on both its scheduling context and the scheduling control capability of the destination core. Cross-core notifications and IPC, delivered via IPIs as described above, are the mechanism by which those migration requests and wakeups propagate between schedulers on different cores.

2.5 The seL4 Microkit

Despite its strong security and performance guarantees, seL4's bare-metal API can present a steep learning curve for developers. This is because seL4 exposes only minimal, low-level primitives, requiring developers to manually manage capabilities and kernel objects rather than relying on the higher-level OS libraries and services to which they are accustomed. The seL4 Microkit framework was created to streamline system and application development by abstracting away architecture-specific details and seL4's low-level calls, albeit with major loss of generality [Velickovic 2025]. Its primary abstractions are:

1. **Protection Domain (PD)**: a process abstraction enforcing an event-driven programming model.
2. **Channel**: a communication link between two PDs, supporting notifications or protected procedure calls.
3. **Memory Region (MR)**: a contiguous physical memory range that can be mapped into one or more PDs.
4. **Notification**: a semaphore-like primitive for asynchronous signalling.
5. **Protected Procedure Call (PPC)**: a synchronous RPC mechanism between PDs.

By adopting an event-driven model — where events arrive via notifications or PPCs on channels — the Microkit guides developers toward correct seL4 usage. Figure 2.8 illustrates the Microkit's end-to-end flow.

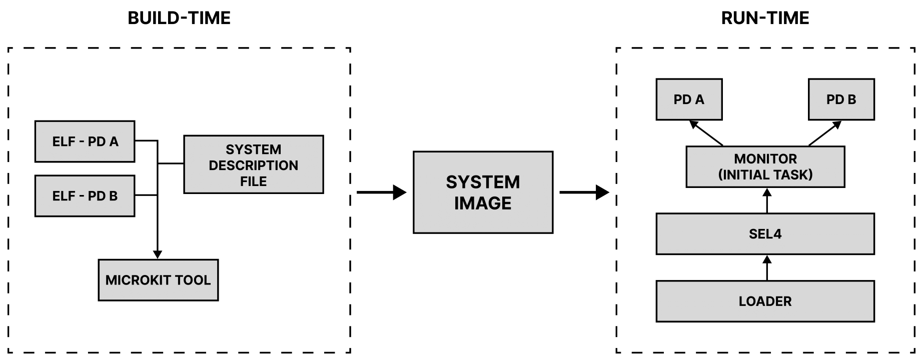


Figure 2.8: Microkit workflow from system description to runtime execution [Velickovic 2025].

The user supplies a system description file (SDF) and the ELF binaries for each PD to the Microkit tool, which packages everything into a single bootable image. At build time, the tool "emulates" the target environment to determine which seL4 system calls to make and with which arguments.

At runtime, the loader first unpacks the image into memory and performs essential hardware initialisation — switching exception levels, configuring the interrupt controller, and enabling the MMU. Once control passes to seL4, the kernel's initialisation completes and invokes the Monitor task. The Monitor sets up all system resources (memory regions, channels, interrupts) and launches the user's PDs. It then remains active to handle any faults or exceptions generated by the PDs.

Each protection domain defines three entry points: `init`, `notified`, and `protected`. The first two are mandatory: `init` serves as the domain's initialiser or constructor, while `notified` handles notifications from other domains or hardware interrupts. The third entry point, `protected`, is optional and is used for PPC. These entry points and communication channels are illustrated in Figure 2.9.

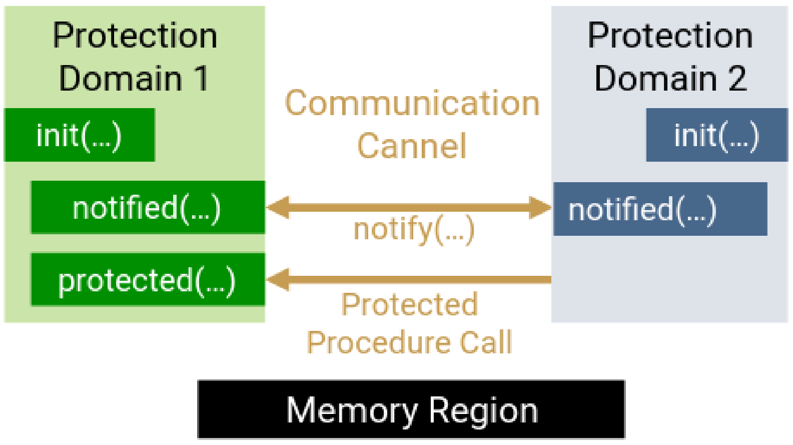


Figure 2.9: Microkit runtime protection-domain communication [Heiser 2020].

2.6 LionsOS

By combining the Microkit framework with the seL4 Device Driver Framework (sDDF) — a collection of user-space driver abstractions and helper routines that manage capabilities, shared memory, and IPC for isolated device drivers — we obtain a fully functional operating system: LionsOS. Although our primary development focuses on the Microkit layer (where the core-manager PD resides) we will ultimately integrate this protection domain into the broader OS, ensuring it remains one of LionsOS's central components. Figure 2.10 depicts the complete system stack, from hardware up through LionsOS and its core subsystems [Heiser 2024].

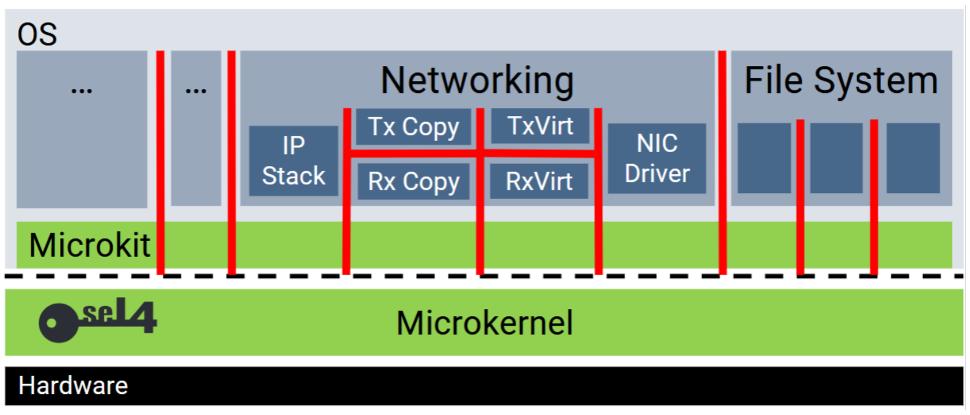


Figure 2.10: LionsOS system stack with modular components [Heiser 2024].

As shown in the figure, fine-grained modularity is a key property of LionsOS. Individual modules execute sequentially on a single core and communicate via shared memory, synchronised with semaphores. This design allows each policy within the OS to be highly specialised for its task, and the system achieves use-case diversity by rewriting policies as needed. Such a radically simple, use-case-specific approach enables system designers to select a policy from an existing library or to adapt one with minimal effort [Heiser, Velickovic, Chubb, Joshy, Ganesh, Nguyen, C. Li, Darville, Zhu, Archer, Zhou, K. Winter, Parker, Duchniewicz, and Bai 2025].

LionsOS components communicate via lock-free, single-producer single-consumer queues in shared memory, with semaphores used for signalling between producer and consumer. Crucially, the semantics of these queues do not depend on where the communicating components execute. A component is unaware of whether its peer runs on the same core or on a different one; it simply enqueues or dequeues buffers and uses the agreed signalling protocol. In this sense, the component model is location transparent [Heiser, Velickovic, Chubb, Joshy, Ganesh, Nguyen, C. Li, Darville, Zhu, Archer, Zhou, K. Winter, Parker, Duchniewicz, and Bai 2025].

Location transparency compensates for the fact that LionsOS components are strictly sequential. Rather than requiring multi-threaded implementations to exploit multicore hardware, the system gains concurrency by distributing components across cores while keeping each component single-threaded. Most LionsOS code is therefore free of explicit concurrency control, with correctness depending only on the proper use of semaphores and a small set of queue operations.

This property also has direct benefits for core management. Because components do not encode assumptions about their physical placement, they can be migrated between cores without changing their behaviour. If a core needs to be taken offline, its components can be moved to other cores with no changes to their code and only temporary impacts on latency, which is essential for dynamic core offlining and energy-aware scheduling on top of LionsOS.

2.7 Thesis Problem Statement

LionsOS exploits multicore hardware by placing different components on different cores, while keeping each component strictly sequential. Because communication is performed only through shared-memory queues and semaphores, a component cannot tell whether its peer is running on the same core, on another core, or has recently been moved.

This thesis aims to design and implement core-migration and core-management mechanisms in the seL4 Microkit — the framework underpinning LionsOS — that preserve this location transparency. Modules should not need to know, or to be reconfigured, when a communicating peer is moved to a different core, temporarily parked in a low-power state, or resumed elsewhere. Instead, the core manager will use a chip-wide view of power budgets and overall CPU utilisation to decide where components execute and which cores can safely be idled, while keeping the LionsOS component interfaces unchanged.

Chapter 3

Related Work

The wider community is well aware of the challenges associated with maintaining all cores in the running state. Numerous attempts have been made to save energy by transitioning these cores into low-power states when they are not in use. In Section 3.1, we explore an existing open-source framework, specifically Linux's CPUIdle subsystem. In Section 3.2, we examine research that developed a tool built on top of CPUIdle to export the idle state decision-making infrastructure to user space. Finally, in Section 3.4, we review a research initiative focused on creating a global core manager, which makes heterogeneous, design-conscious decisions to outperform local power management techniques. These insights are then applied to guide our own design in Chapter 4.

3.1 The Linux CPUIdle framework

CPUIdle is a module in the Linux kernel responsible for executing power-saving routines on a core when there are no tasks in its run queue [Pallipadi, S. Li, and Belay 2007]. These routines attempt to transition the core into a low-power state. As mentioned in Section 2.1, every idle state has associated entry and exit latencies. The trade-off is that while deeper idle states result in greater energy savings, they also come with increased entry and exit latencies.

The CPUIdle module has the following components:

- The `sysfs` interface
- The CPUIdle governors
- The core CPUIdle logic
- The platform specific driver functions

most of which is found in Figure 3.1.

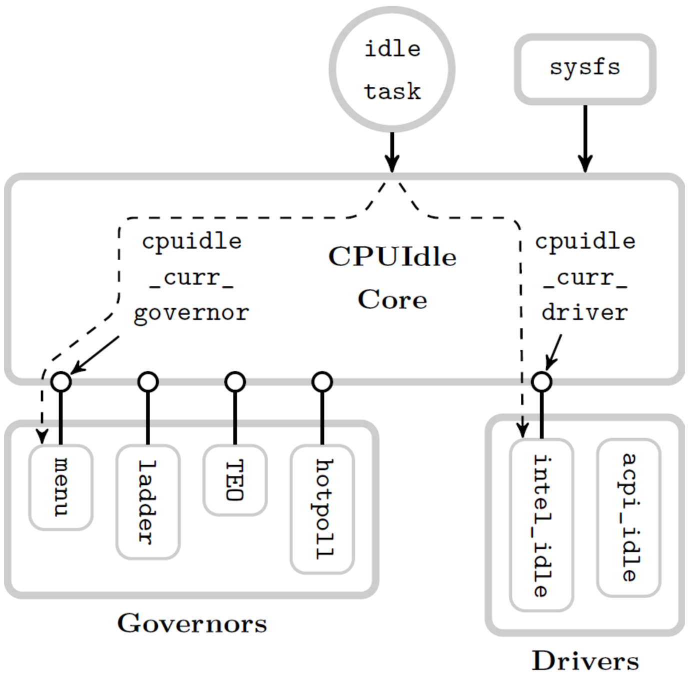


Figure 3.1: The CPUIdle subsystem and architecture [Liu and X. Li 2024].

Once the processor core is designated as idle through its empty run queue, Linux dispatches a special idle task to do the CPU idle time management. The idle task is also referred to as the idle loop because it repeats finding opportunities to put the idle core into deep sleep states when there are no other t asks to run. It completes C-state entering with the help of the CPUIdle subsystem.

3.1.1 Idle Task

The idle task is effectively an infinite loop containing the core logic of the CPUIdle system, which is run on every core. In each loop cycle, it calls the CPUIdle subsystem to choose a C-state appropriate for the current CPU's running state and then invokes the CPUIdle subsystem to drive the CPU into the selected sleep state.

When the system receives an interrupt, it exits the sleep state and begins handling the interrupt. The interrupt could originate from another CPU and be triggered by the scheduler because the idle CPU needs to handle new tasks, or by other means such as a completed I/O command [Liu and X. Li 2024].

After interrupt handling, the idle task resumes executing the instructions following the C-state entry call. The scheduler rechecks the core's run queue, and if no tasks are found, another loop cycle begins.

3.1.2 CPUIdle Governors

CPUIdle governors are responsible for selecting the optimal C-state for each core whenever it becomes idle. Their goal is to maximise energy savings without violating latency constraints imposed by the system's power management quality of service (PM QoS) [The Linux Foundation 2024]. Each C-state is characterised by three parameters:

1. **Exit latency**: the time required to return the core to C0,
2. **Target residency**: the minimum time the core must remain in that state to justify the transition overhead,
3. **Approximate energy savings**: relative to the active state.

A governor must therefore predict how long a core will remain idle and choose the deepest C-state whose target residency does not exceed this prediction and whose exit latency falls below the PM QoS latency budget [Liu and X. Li 2024]. Since true idle duration is unknown at decision time, governors typically use two heuristics:

- The interval until the next timer event (the maximum guaranteed idle window), and
- The length of the previous idle period (to capture recent workload patterns).

Different governor algorithms balance responsiveness and energy efficiency in various ways — some favour conservative, low-latency states for interactive workloads, while others aggressively enter deep sleep to maximise savings under longer idle gaps. This diversity is why multiple CPUIdle governors coexist in the Linux kernel [The Linux Foundation 2024].

Governor	**Algorithm**
`menu`	Default governor for tickless systems. Picks the deepest C-state whose target residency $\leq$ time to next timer event and whose exit latency $\leq$ PM QoS latency.
`ladder`	Default governor for non-tickless systems. Performs a binary search through the sorted C-state list to find the optimal state, reducing search overhead when many C-states exist.
`TEO`	Same as the `menu` governor, it always tries to find the deepest idle state suitable for the given conditions. However, it does so solely using timer events as system timer interrupts are typically two or more orders of magnitude more frequent than other interrupt types.
`haltpoll`	A special governor that is used for virtualisation.

Table 3.1: Linux CPUIdle governors and their core algorithms/use cases.

3.1.3 CPUIdle Drivers

CPUIdle drivers discover and register the C-states supported by the hardware, then integrate those states into the kernel's CPUIdle framework [Liu and X. Li 2024, The Linux Foundation 2024]. Their lifecycle comprises three main phases:

1. **State Discovery and Initialisation**
   The driver enumerates C-states via platform firmware (ACPI tables) or device-tree bindings, populating a `struct cpuidle_state` array [Wysocki 2018]. Each entry records:

   - Name (e.g. "C1", "C6")
   - Estimated power usage
   - Entry and exit latencies
   - Callback pointers for `enter`
   - Flags and residency requirements

   Entries are sorted by increasing `target_residency` so deeper states appear later in the array [The Linux Foundation 2024].

2. **Driver Registration**
   Once initialised, the driver registers its `struct cpuidle_driver` with the CPUIdle core:

   ```
   cpuidle_register_driver(&my_cpuidle_driver);
   ```

   The first registered driver becomes the CPUIdle driver for the entire system [The Linux Foundation 2024].

3. **Per-CPU Device Registration**
   The driver then registers each logical CPU as a CPUIdle device:

   ```
   cpuidle_register_device(&cpuidle_device_for_cpu);
   ```

   After this, the CPUIdle core invokes the driver's callbacks to perform the actual hardware transition (e.g. `MWAIT` or power-gate) to the selected C-state whenever the CPU enters or exits idle [Linus Torvalds and Linux Kernel Developers 2024].

With the C-state metadata and callbacks in place, the CPUIdle core can select and invoke the optimal idle state on every loop iteration, balancing energy savings against entry/exit latencies and PM QoS constraints [The Linux Foundation 2024, Wysocki 2020].

3.1.4 The `sysfs` interface

The CPUIdle `sysfs` interface exposes both read-only and read-write attributes.

Read-only entries include:

- `current_driver`: the active CPUIdle driver.
- `current_governor_ro`: the governor algorithm currently in use.

For every core read only information like energy consumed while in a particular idle state, latency to exit out of a particular idle state is also exposed [Palmur, Z. Li, and Zadok 2013].

Writable entries allow dynamic reconfiguration:

- `current_driver`: switch the active CPUIdle driver at runtime.
- `current_governor`: change the governor algorithm at runtime.

With these interfaces, administrators and user-space tools can monitor idle-state behaviour and adjust governors on the fly [Pallipadi, S. Li, and Belay 2007].

3.1.5 Energy-Aware Scheduling

Beyond simply placing idle cores into low-power states, the Linux scheduler also needs to make energy-aware placement decisions that consolidate runnable threads onto fewer CPUs, giving the CPUIdle governor an opportunity to select deep sleep states. The kernel does this in two main ways: a *power-saving load balancer* and *energy-aware scheduling.*

By default, Linux is not built with the power-saving load balancer enabled, but it can be compiled in and tuned at run time. When configured, the user controls its behaviour via the multi-core scheduling knobs (for example `sched_mc_power_savings`). A value of 1 instructs the scheduler to prefer filling one core or package with long-running threads before using others, while a value of 2 also biases task wake-ups toward a partially loaded or "semi-idle" package. In effect, the scheduler attempts to pack work onto a subset of cores so that the remaining cores or packages can remain completely idle for longer intervals and are more likely to reach deep sleep states [Chiang 2009].

In addition to this heuristic load balancer, Linux provides an *Energy Aware Scheduler* (EAS). Unlike the simple consolidation scheme described above, EAS operates only on heterogeneous CPU topologies, such as Arm big.LITTLE systems, where different cores have different performance and energy characteristics. EAS uses an energy model of the platform together with per-CPU utilisation signals to estimate the energy cost of running a task on each candidate CPU, then selects the placement that minimises estimated energy while still respecting performance constraints [The kernel development community 2025]. This often results in running most work on a small set of efficient cores and leaving other cores idle, again increasing the opportunities for deep idle states and lower overall energy use.

3.1.6 Takeaways

The CPUIdle framework is highly scalable and modularised: each component lives in its own well-defined container, allowing new drivers to be added without impacting other code — particularly the governors. Likewise, new algorithms and policies can be integrated by creating a new in-kernel governor, without changing drivers, the `sysfs` interface, or other framework logic.

However, this design has some drawbacks. Although adding governors in-kernel is straightforward, there is no equivalent mechanism in user space, so any new algorithm requires kernel changes and expands the TCB. Furthermore, because CPUIdle operates on a per-core basis (via each core's idle thread), it only sees its own local heuristics and has no knowledge of global system state. As a result, it may make suboptimal C-state transition decisions in favour of lower latency by avoiding coordination overhead such as cross-core communication. Linux tries to mitigate some of these limitations with energy-aware scheduling, using an in-kernel energy model to influence task placement, but those decisions are still tightly coupled to the kernel and its local view of the system.

In contrast, LionsOS leverages its highly modular architecture to enable use-case diversity. By using the user-space Microkit framework, power-management policies can be implemented as isolated modules with a global view of system state. This approach keeps the kernel small — limiting the TCB — while enabling coordinated, use-case-specific C-state and core-management policies that optimise energy efficiency across platforms and workloads.

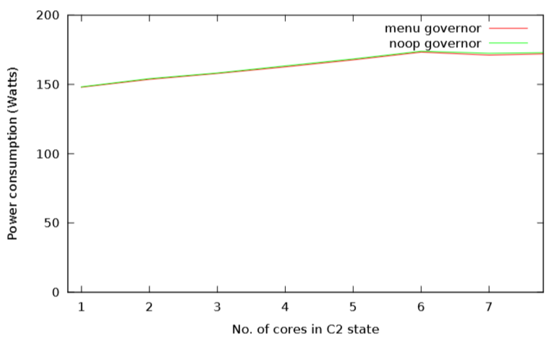
3.2 CPUIdle from user space

To address some of the shortcomings of the existing CPUIdle framework in Linux, Madhu Palmur, Zhichao Li, and Erez Zadok proposed migrating part of its functionality to user space in their paper [Palmur, Z. Li, and Zadok 2013], based off prior work from [Snowdon, Le Sueur, Petters, and Heiser 2009].

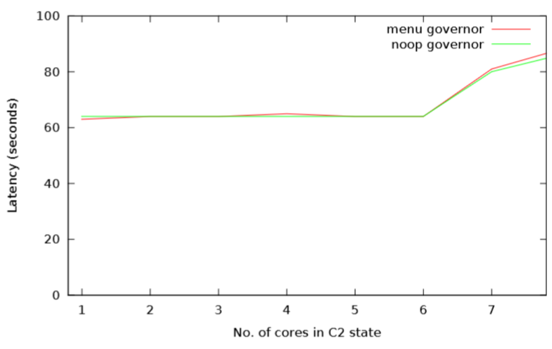
This approach is necessary because all current governor algorithms operate entirely in kernel space, making the implementation of new, customised algorithms for various workloads impractical. Each time a new governor algorithm is developed, the process requires writing a new patch, modifying the kernel configuration file, recompiling the kernel, and reinstalling it [Pallipadi, S. Li, and Belay 2007, Palmur, Z. Li, and Zadok 2013]. Additionally, writing kernel code is significantly more complex than developing programs in user space, making the creation of multiple customised governors cumbersome and inefficient.

The goal was to build a system that grants the user complete control over the idle states of the cores. One of the simplest approaches, and the first method explored in the paper, was to create a custom governor. This governor (they called the Noop governor) merely reads the user-requested C-state and sets the next C-state value for the core accordingly. However, stray interrupts or tasks scheduled on that core caused the idle thread to yield, transitioning the core state back to C0, even when the user had requested a deeper C-state.

To address this issue, the CPU hot-plugging infrastructure was used. This infrastructure makes the core completely invisible to the OS and kernel scheduler, migrating all tasks, interrupts, timers, and tasklets from the "victim" core to other online cores [Mwaikambo, Raj, Russell, Schopp, and Vaddagiri 2004]. By leveraging the `MWAIT` instruction pair, the system was able to halt the cores and place them in any desired idle state deeper than C0.



(a) Average energy draw



(b) Performance (latency)

Figure 3.2: Noop vs Menu `cpuidle` governors for a CPU-bound workload: (a) average energy draw and (b) performance (latency) [Palmur, Z. Li, and Zadok 2013].

Figure 3.2 compares the power draw and latencies of the user-space governor with the default menu governor. Under a fully CPU-bound workload, the user-space governor performs comparably to the kernel-mode menu governor. However, when tested with a workload that includes intermittent I/O operations, the latency graphs remain nearly identical, but the power draw graph shows a significant difference, as illustrated in **??**. The `make -jX` command was used to spawn X threads to complete the workload.

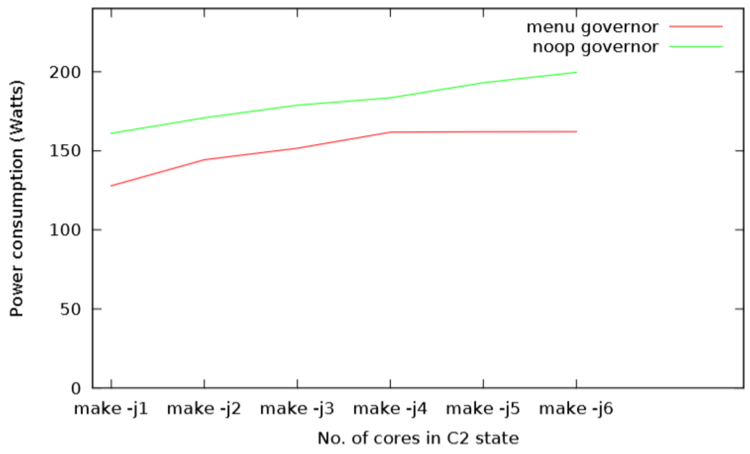


Figure 3.3: Comparison of average power draw for the Noop and Menu governors under a `make -jX` workload [Palmur, Z. Li, and Zadok 2013].

The primary reason for the user-space governor's inefficiency lies in its handling of intermittent I/Os. During these periods, the core lacks sufficient work, creating an opportunity to save power. Unlike the menu governor, which identifies this as an ideal moment to transition the core into a low-power state, the Noop governor keeps the cores active, failing to enter any idle states [Palmur, Z. Li, and Zadok 2013].

3.2.1 Takeaways

The authors identify a key drawback in the Linux CPUIdle framework: keeping the entire system in-kernel severely limits extensibility from userland. This not only makes creating and testing many different workload-specific algorithms cumbersome — because each change requires kernel modification, recompilation and reboot — but also raises the skill floor, since writing kernel code is considerably more difficult than user-space programming.

That said, several enhancements could improve their prototype. In particular, employing more intelligent predictors to first learn a workload's I/O patterns and then forecast imminent I/O events would allow the tool to exploit short idle windows for energy savings as explored in prior aforementioned work [Snowdon, Le Sueur, Petters, and Heiser 2009]. In doing so, its energy efficiency could match or even surpass that of the existing in-kernel governors.

3.3 Windows Processor Power Management

In this section, we examine Windows processor power management (PPM), using the white paper [Microsoft Corporation 2017] as our primary reference. Although Windows 7 reached end of life on 14 January 2020, this document remains a valuable case study and one of the few detailed, publicly available descriptions of the Windows PPM architecture. Newer versions of Windows (10 and 11) are described mainly through higher-level online documentation rather than an equivalent, in-depth white paper [Microsoft Corporation 2024].

The responsibilities for processor power control are shared between the platform firmware and the operating system components:

- System firmware (ACPI BIOS),
- the Windows kernel power manager, and
- Windows processor drivers.

Figure 3.4 shows how these components relate to one another.

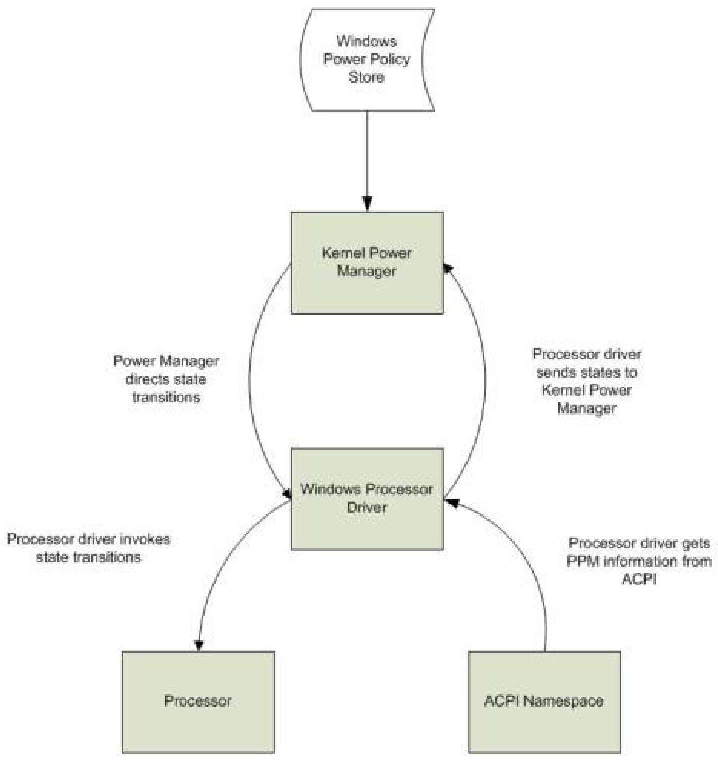

Windows Power Policy Store
Kernel Power Manager
Power Manager directs state transitions
Processor driver sends states to Kernel Power Manager
Windows Processor Driver
Processor driver invokes state transitions
Processor driver gets PPM information from ACPI
Processor
ACPI Namespace

Figure 3.4: Main components of the Windows processor power management stack [Microsoft Corporation 2017].

3.3.1 System firmware

The system firmware is responsible for providing the ACPI tables that describe the platform's processor power-management features and controls. These ACPI processor objects specify the available performance and idle states, and may also include optional objects that describe multiprocessor domain dependencies or support for cooperative performance state management (for example, platform co-ordinated P-states via the Processor Clocking Control interface) [Microsoft Corporation 2017]. The firmware can modify the exposed features in response to system events, such as transitions between AC and DC power sources.

Processor C-states can be declared in firmware by using one of the following ACPI processor idle-state control interfaces [Microsoft Corporation 2017]:

- the ACPI 2.0 `_CST` object in the processor's object list; or
- the legacy ACPI 1.0 interface, which uses the Processor Register Block (`P_BLK`) `P_LVL2` and `P_LVL3` registers together with the FADT `P_LVLx_LAT` latency values.

3.3.2 Windows kernel power manager

The Windows kernel power manager is responsible for:

- managing and applying processor power policy,
- determining the required transitions between processor power states, and
- enforcing constraints arising from thermal conditions or other system limits.

Processor power policy is owned by the kernel power manager, which selects an appropriate target state based on CPU utilisation and other inputs, depending on the PPM features available on the platform. Once a target state has been chosen, the power manager issues a request through a direct-call interface to the relevant processor driver. The driver then performs the hardware transition, for example by programming model-specific registers on the processor or other platform control registers [Microsoft Corporation 2017].

3.3.3 Processor drivers

Processor drivers in Windows hide processor-specific details from the rest of the operating system. They typically contain routines to detect the presence of PPM features on a particular CPU implementation and may use model-specific registers (MSRs) or other vendor-defined interfaces to invoke power-state transitions directly on the processor.

These drivers are responsible for:

- enumerating the system's PPM features,
- validating that these features are correctly described in firmware and present in hardware,
- reporting the supported features to the kernel power manager, and
- carrying out processor power-state transitions on the appropriate hardware as directed by the power manager.

Processor objects appear in the ACPI namespace and are enumerated by the Windows ACPI interpreter. During enumeration, Windows uses information from the `CPUID` instruction to derive the correct hardware device ID for each processor, which allows the system to load an appropriate vendor-specific processor driver. If no specific match is found, a generic processor driver is used instead [Microsoft Corporation 2017].

For idle-state control, the processor driver first checks for ACPI 2.0-style C-state support in the system firmware, indicated by the presence of `_CST` and `_CSD` objects in the ACPI namespace. If ACPI 2.0 C-states are not available, the driver falls back to the legacy ACPI 1.0 C-state interface based on the `P_BLK P_LVLx` registers and associated latency fields [Microsoft Corporation 2017].

3.3.4 Core Parking

Core Parking in Windows is a co-operative mechanism between the processor power management (PPM) engine and the kernel scheduler that adjusts, at run time, how many logical processors are allowed to run threads. Instead of spreading work evenly across all CPUs, the scheduler is encouraged to concentrate runnable threads on a subset of logical processors so that the remaining ones can stay completely idle and allow their cores to enter deep idle states with very low power draw [Microsoft Corporation 2017, Microsoft Corporation 2024].

When the Core Parking policy decides that fewer logical processors are needed, it marks some of them as *parked*. The scheduler then prefers unparked processors when placing any non-affinitised threads, avoiding parked CPUs except when additional capacity is required. Parked logical processors therefore see no normal thread execution and can remain in deeper C-states for longer intervals, which improves overall energy efficiency without necessarily reducing throughput [D. Tang 2021]. Below is an example of two threads that depend on each other to complete the previous phase of work to be able to start the next phase. Figure 3.5 shows the spread of this work across CPU 0 and CPU 1 with Core Parking disabled, whereas Figure 3.6 demonstrates the spread of this work with Core Parking enabled.

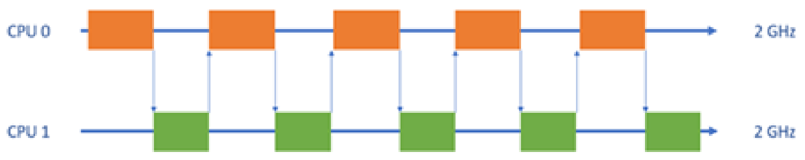


Figure 3.5: Windows Core Parking setting with Core Parking disabled [D. Tang 2021].

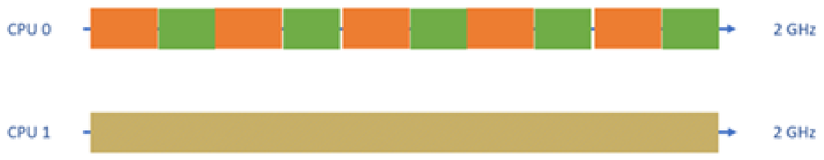


Figure 3.6: Windows Core Parking setting with Core Parking enabled [D. Tang 2021].

Core Parking is particularly effective on systems that offer deep, low-leakage idle states. Windows complements this with Intelligent Timer Tick Distribution (ITTD), a mechanism that reduces unnecessary periodic timer interrupts to idle processors so they are not woken solely to service clock ticks. Instead, timekeeping is updated when a processor is next woken for useful work, allowing long uninterrupted idle periods.

3.3.5 Takeaways

Windows processor power management is built around a clear separation of responsibilities. Firmware exposes the available performance and idle states through ACPI tables, processor drivers translate vendor-specific mechanisms into a uniform interface, and the kernel power manager applies policy by selecting target states and requesting transitions [Microsoft Corporation 2017]. This division allows the same power-management logic to operate across a wide range of processor implementations and platforms, with most hardware idiosyncrasies confined to the processor drivers.

On top of this foundation, Windows combines power management with scheduling to improve energy efficiency. Core Parking uses policy from the PPM engine and the kernel scheduler to concentrate load on a subset of logical processors so that the remaining cores can stay in deep idle states, while timer-related mechanisms reduce unnecessary wake-ups and extend idle intervals [Microsoft Corporation 2024, D. Tang 2021]. However, as with Linux's in-kernel power-management framework, these policies live inside a large, proprietary kernel and are not easily replaced or specialised by user-space components. In contrast, the LionsOS design in this thesis aims to achieve similar co-ordination between scheduling and power management through user-space core-management services with a global view of system state, without enlarging the trusted computing base.

3.4 A global power manager

In this section, we examine a global power manager proposed by IBM's T.J. Watson Research Center [Sharkey, Buyuktosunoglu, and Bose 2007]. The authors contend that, as multicore chips become ever more aggressively scaled out, it is essential to adopt a holistic view of both per-core and chip-wide power–performance demands. By leveraging this information, a shared power budget can be allocated among the cores to maximise overall throughput while staying within the chip's thermal and power envelope.

Power-management schemes generally fall into two categories: local and global.

- Local techniques delegate power decisions entirely to each core, with no inter-core communication. This approach is simple to implement but cannot exploit opportunities for cross-core optimisation.
- Global techniques, in contrast, rely on a central power manager that interacts with every core. Armed with a comprehensive view of both aggregate and per-core metrics, it can make allocation decisions that optimise performance and energy efficiency at the chip level. Global managers themselves are further divided into:
  - Homogeneous approaches, which treat all cores identically and give each core an equal power budget,
  - Heterogeneous approaches, provide a dynamic redistribution of the power budget among the cores.

Figure 3.7 illustrates the maxBIPS (Billions of Instructions per Second) algorithm, which optimises overall system throughput by evaluating every voltage–frequency pair for each core and selecting the configuration that satisfies the power budget while maximising performance [Isci, Buyuktosunoglu, Cher, Bose, and Martonosi 2006]. The authors also implemented a global homogeneous power manager using a chip-wide DVFS scheme. Both approaches were evaluated at multiple time granularities.

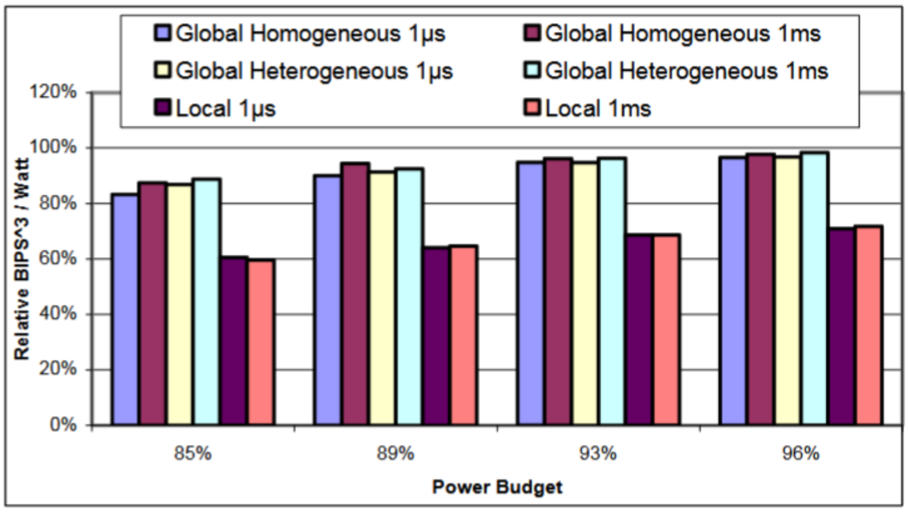


Figure 3.7: BIPS^3/W relative to the baseline system across power-management strategies and power budgets [Sharkey, Buyuktosunoglu, and Bose 2007].

As shown in Figure 3.7, all global solutions significantly outperform the local ones across both time granularities. This improvement arises because, even if one core temporarily exceeds its individual power allotment, the total chip power remains within the overall budget as long as the other cores draw less power. A local power manager cannot detect this redistribution and will therefore throttle the high-consuming core unnecessarily, reducing its performance and power efficiency [Sharkey, Buyuktosunoglu, and Bose 2007].

Importantly, the authors also observe that, although the power manager's time granularity has little impact on application performance per watt, the average power-budget overshoot across all benchmarks increases substantially at a 1ms granularity. At this fine granularity, the power manager cannot react quickly enough to changing workload behaviour to prevent the large budget overshoots shown in Figure 3.8.

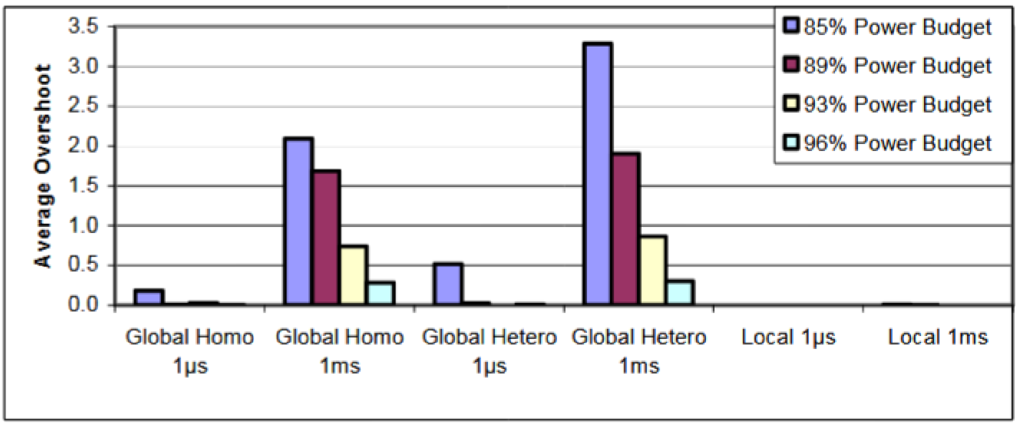


Figure 3.8: Average overshoot across all benchmarks for several power-management solutions [Sharkey, Buyuktosunoglu, and Bose 2007].

However, this paper is now somewhat dated, and techniques such as DVFS are no longer as widely used as discussed in Chapter 2. As a result, the opportunities for power management have narrowed: due to high leakage currents, approaches that place processor cores in idle states — such as race-to-halt — are almost always the most effective strategy [Le Sueur and Heiser 2011, Le Sueur and Heiser 2010]. Furthermore, although the authors compared a heterogeneous core manager with a homogeneous one, their experiments were conducted on a homogeneous, four-core processor. The heterogeneous manager does allow more dynamic power redistribution across cores — adapting more effectively to workloads with significant inter-thread variation — but the difference between the two approaches would be far more pronounced on an actual heterogeneous processor.

3.4.1 Takeaways

One of the key takeaways from this paper is that global core managers significantly outperform local core managers, primarily because the global manager has access to chip-wide information. This allows it to track heuristics and manage power budgets across the entire system, rather than being limited to a single core. However, such systems are prone to power budget overshoots at coarse granularities, making it crucial to balance time granularity to achieve a mix of responsiveness while avoiding excessive CPU overhead.

Chapter 4

Design and Implementation

Before delving into the implementation details of this thesis, it is important to refine the broad title and problem statement into a more focused scope. To achieve a low-latency and highly energy-efficient core manager in LionsOS, we extend the current SMP version of the seL4 Microkit to enable a new privileged protection domain. Section 4.1 outlines the key objectives of the final design, which will be explored in detail throughout the rest of this chapter, including both the current implementation and future enhancements.

4.1 Design Goals

Before outlining the methodology of our design, we establish a set of design goals to serve as guiding principles, which we aim to uphold as we extend the seL4 Microkit and LionsOS:

1. **Fine-grained modularity:** As discussed in Chapter 2, modularity is a core principle of LionsOS. Since seL4 is intentionally policy-agnostic, all energy-aware scheduling and protection-domain migration logic must be fully encapsulated within the core-manager module.
2. **Use-case diversity:** The core manager must support an arbitrary number of policies — both provided and user-defined. Adding a new policy for a specific use case should be straightforward and require minimal effort.
3. **Extensibility:** It should be easy to extend the core manager with new features or updates. For example, enabling OS-initiated mode (see Chapter 2) or integrating additional energy-saving mechanisms, such as DVFS, must involve minimal changes.
4. **Security:** The solution must leverage seL4's capability-based protection to enforce least privilege. Although the core manager requires privileged access to other domains, no privileged state or data may be exposed to unprivileged components.

5. **Performance:** The core manager should be competitive with industry-standard implementations and research prototypes in both latency and throughput (see Chapter 3).

To achieve our design goals, the following tasks must be completed:

1. Design and implement the mechanisms to offline a particular CPU core (Section 4.2).
2. Develop mechanisms to migrate protection domains between CPUs (Section 4.3).
3. Outline the necessary infrastructure to bring back online a previously offlined CPU core (Section 4.4).
4. Support intermediate power states for improved entry/exit latencies (Section 4.5).

4.2 Offlining CPU Cores

In this section, we describe the design for taking a CPU core offline to reduce energy use. The basic structure consists of two protection domains: a Core Manager, which issues requests to offline specific cores, and a Core Manager API, which receives and services these requests. Communication between these domains is performed via a protected procedure call.

To preserve key LionsOS properties such as modularity and ease of extension, the Core Manager component is intended to be implemented by the user. It can be replaced or customised to suit different policies or even different operating systems built on top of the seL4 Microkit. Figure 4.1 shows how the design is conceptually split into two parts: the *user implementation*, which currently consists of the Core Manager, and the *core manager subsystem*, which currently consists of the Core Manager API. Everything within the core manager subsystem is designed to be a drop-in, platform-agnostic implementation that accepts and executes requests from the user implementation. The user implementation, by contrast, is expected to vary with the use case and can be reimplemented or swapped out without changing the subsystem.

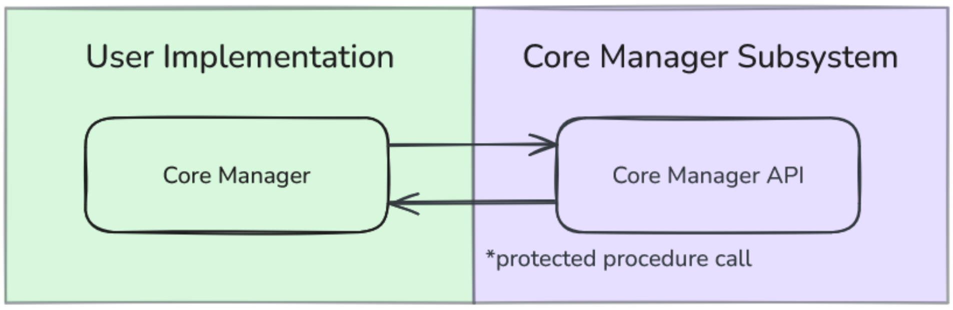


Figure 4.1: High-level structure of the core-management design, showing the user-implemented Core Manager and the platform-agnostic Core Manager API.

To offline a particular core, the Core Manager sends a request to the Core Manager API, which performs the offlining operation on its behalf without exposing any internal details. The Core Manager API must then forward this request to Arm Trusted Firmware via the `CPU_OFF` function introduced in Chapter 2. However, if the Core Manager requests to turn off core 1 while the Core Manager API is running on core 0, this creates a problem, because `CPU_OFF` must be run on the core that is being turned off.

To address this, we introduce a per-core worker protection domain, called a Core Worker. The primary role of a Core Worker is to execute operations on the target core on behalf of the Core Manager API when the API itself cannot do so directly. Figure 4.2 illustrates a simple system with three cores, where the Core Manager and Core Manager API reside on core 0, and a Core Worker runs on each core.

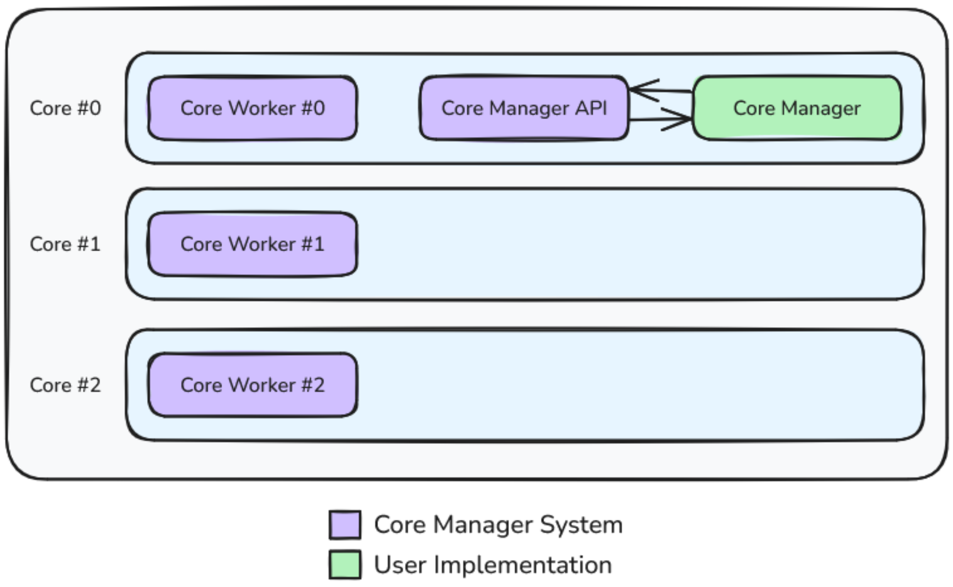


Figure 4.2: Core Manager and Core Manager API on core 0, with a Core Worker protection domain instantiated on each core.

With this design, the Core Manager can effectively turn off any chosen core by first issuing a protected procedure call to the Core Manager API, passing an encoded instruction that specifies which core to offline. The Core Manager API then validates that the instruction is safe to execute (for example, it rejects requests that attempt to turn off a core that is already offline, or the core currently hosting the Monitor protection domain) and signals the corresponding Core Worker to invoke the `CPU_OFF` function. It is important that the Core Manager API sends a notification rather than a protected procedure call to the Core Worker, because the Core Worker is not expected to reply after executing an instruction that offlines the core on which it is running.

In our implementation, the Core Manager API is realised as a Microkit protection domain that exposes a single `protected()` entry point. When a client issues a `CORE_OFF` request, the API encodes the requested instruction into a shared instruction buffer, checks that the target core is a valid candidate for offlining (for example, not already offline and not hosting the Monitor), and then notifies the Core Worker on that core via `microkit_notify`. The Core Worker's `notified()` handler reads the instruction and executes the corresponding PSCI call locally. For the `CPU_OFF` case, this reduces to issuing a single SMC, as shown in Listing 4.1, after which the core powers down and never returns to the worker code path.

```c
static void core_off(void) {
    seL4_ARM_SMCContext args = { .x0 = PSCI_CPU_OFF };
    seL4_ARM_SMCContext response = {0};

    microkit_arm_smc_call(&args, &response);
    print_error(response);  // Only reached on error
}
```

Listing 4.1: Core Worker implementation of the CPU_OFF path

4.3 Migrating Protection Domains

However, before turning a CPU core off, we must ensure that all protection domains have been migrated away from that core, including any interrupts they are subscribed to. This is a precondition imposed by Arm Trusted Firmware, because leaving runnable protection domains or pending interrupts on a core can lead to unexpected behaviour, such as the core waking up again or interrupts being lost. To satisfy this precondition, we need a mechanism to migrate a protection domain together with all of its interrupts, as shown in Figure 4.3.

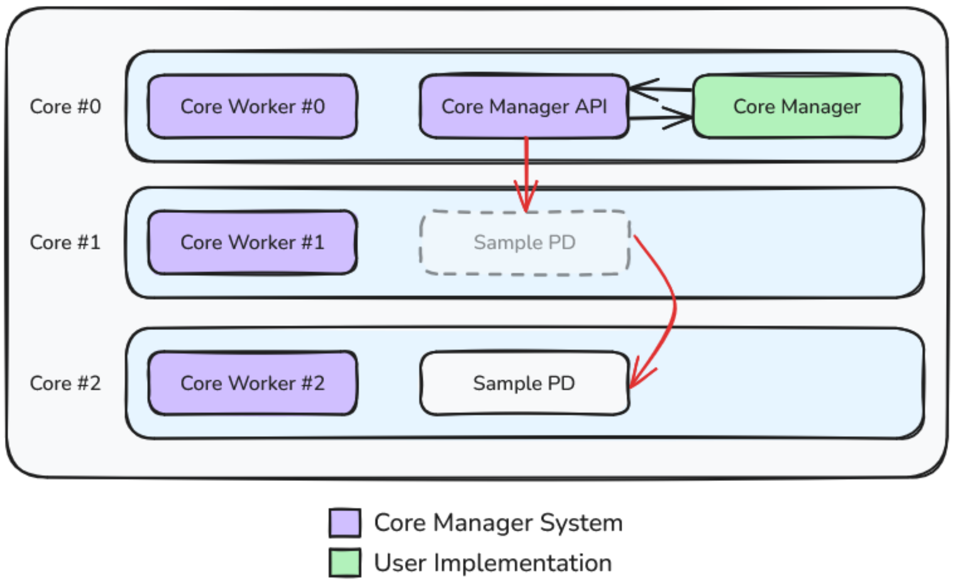


Figure 4.3: Core Manager sending a request to migrate the sample PD from core 1 to core 2.

On seL4, this can be achieved by updating the scheduling parameters of the protection domain's TCB. By changing the TCB's affinity to the target core, the MCS scheduler will subsequently run the protection domain on that core. On the next scheduler timer tick, or when the protection domain receives a notification, the scheduler is invoked again and the PD is scheduled on the core requested by the Core Manager. Similarly, to migrate interrupts from one core to another, we update the target CPU of each interrupt by writing the appropriate interrupt routing register so that the interrupt is delivered to the new core.

To implement this in LionsOS, the Core Manager API holds a set of capabilities that allow it to redirect both execution and interrupts. Scheduling-context capabilities are laid out contiguously starting at `BASE_SCHED_CONTEXT_CAP`, with one scheduling context per protection domain. Per-core scheduling control capabilities are laid out starting at `BASE_SCHED_CONTROL_CAP`, with one object per core. IRQ handler capabilities are arranged in a similar contiguous range starting at `BASE_IRQ_CAP`, with one capability for each interrupt ID.

The Core Manager API also maintains a compact description of each protection domain in a generated `pd_infos[]` array. This array is produced by the Microkit tool at build time by parsing the SDF and extracting, for every PD, its initial core assignment, scheduler period and budget, and the set of interrupts it subscribes to. Each entry stores the PD's current core (`pd_core`), a 64-bit bitmap of active IRQs (`pd_irqs`), and its period and budget in microseconds. The array is embedded into the Core Manager API's ELF and updated at runtime whenever a migration occurs.

In the first step of a migration, the Core Manager API moves the PD's scheduling context to the new core. This is implemented as a thin wrapper around `seL4_SchedControl_ConfigureFlags`, which rebinds the scheduling context to the target core while preserving the PD's period and budget.

```c
static int core_migrate(uint8_t pd, uint8_t core) {
    /* Do not migrate core worker PDs=1..NUM_CPUS. */
    if (pd >= 1 && pd <= NUM_CPUS) return 1;

    seL4_SchedControl_ConfigureFlags(
        BASE_SCHED_CONTROL_CAP + core,
        BASE_SCHED_CONTEXT_CAP + pd,
        pd_infos[pd].pd_period,
        pd_infos[pd].pd_budget,
        0,                              // Extra refills
        0x100 + pd,                     // Badge
        0                               // Flags
    );

    pd_infos[pd].pd_core = core;
    return 0;
}
```

Listing 4.2: Core Manager helper for migrating a protection domain's scheduling context

Once the PD's execution has been redirected, the second step is to move all of its interrupts. The Core Manager API iterates over the `pd_irqs` bitmap and, for each set bit, uses `seL4_IRQHandler_SetCore` to reroute that IRQ to the new core. This keeps the PD and all of its interrupt sources co-located.

```c
static void migrate_pd_irqs(uint8_t pd, uint8_t core) {
    for (int i = 0; i < MAX_IRQS; i++) {
        if (pd_infos[pd].pd_irqs & (1ULL << i)) {
            seL4_IRQHandler_SetCore(BASE_IRQ_CAP + i, core);
        }
    }
}
```

Listing 4.3: Migrating all IRQ handlers associated with a protection domain

The user-level `seL4_IRQHandler_SetCore` syscall is backed by a small kernel primitive that updates the GIC routing state for the given interrupt, shown in Listing 4.4.

```c
void invokeIRQHandler_SetIRQCore(irq_t irq, word_t core)
{
    setIRQTarget(irq, core);
}
```

Listing 4.4: Kernel entry point for migrating an IRQ handler to a different core

Together, these two steps provide a complete migration operation: the PD's scheduling context is moved to the new core, and all of its IRQ handlers are rerouted to that core as well. In the offlining path, the Core Manager API applies this mechanism to every non-worker protection domain on a core before issuing a `CORE_OFF` request, ensuring that by the time the corresponding Core Worker invokes `CPU_OFF`, no runnable protection domains or active interrupts remain on the target core.

4.4 Onlining CPU Cores

Bringing a CPU core back online is considerably more complex than turning it off. The core must be restored to a runnable state, which involves two main steps. First, its power-gated hardware components must be re-enabled, a task handled by Arm Trusted Firmware. Second, the execution state of the previously running system, including the kernel, must be reconstructed, which is the responsibility of the core-manager subsystem.

A core is reactivated via the `CPU_ON` function provided by Arm Trusted Firmware. This call takes parameters identifying the core to be brought online and a physical address of a bootstrap function. That bootstrap code is executed by Arm Trusted Firmware and is responsible for bringing the software stack back into a runnable state.

In LionsOS, the Core Manager API exposes a `CORE_ON` command through its `protected()` entry point. When this command is invoked, the Core Manager API issues a PSCI `CPU_ON` call using a helper function, passing the target core identifier and a shared bootstrap entry point (`bootstrap_entry`) that was prepared during initialisation by copying a small bootstrap stub into a region of memory visible to all cores. The helper simply wraps the PSCI call and reports any error codes returned by Arm Trusted Firmware, as shown in Listing 4.5.

```c
static void core_on(uint8_t core, seL4_Word cpu_bootstrap)
{
    seL4_ARM_SMCContext args = {
        .x0 = PSCI_CPU_ON,
        .x1 = core,
        .x2 = cpu_bootstrap
    };
    seL4_ARM_SMCContext response = {0};

    microkit_arm_smc_call(&args, &response);
    print_error(response);
}
```

Listing 4.5: Core Manager helper for turning a core back on via PSCI

Both the offlining and onlining paths rely on knowing whether a core is currently online. Conceptually, the Core Manager needs to reject attempts to power down a core that is already off, and it should not attempt to migrate protection domains or restart PDs on a core that has not yet fully joined the scheduler after a `CPU_ON` call. While this information could be tracked in user space, doing so would introduce a subtle race: the Core Manager might mark a core as "online" immediately after issuing `CPU_ON`, even though the kernel has not yet executed its secondary-core bootstrap and updated its internal data structures. In that window, the Core Manager could use the migration API to schedule a protection domain onto a core that the kernel still considers offline.

To avoid this race, core on/off status is derived from kernel state via a small syscall on the `SchedControl` object. The `seL4_SchedControl_CoreStatus` API consults the kernel's internal `isCPUOnline` flag for the core associated with a given scheduling-control capability and returns a one-bit status to the caller. Internally, the SMP kernel represents core reachability as a single 64-bit bitmap, with one bit per core. The current configuration supports up to 64 CPUs, so this bitmap precisely encodes which cores are online at any given time and allows `isCPUOnline` to answer queries with a single mask operation. The corresponding decode function is shown in Listing 4.6. Because the online flag is only updated once the secondary-core bootstrap path has completed, user-level code cannot observe a core as online before the kernel is ready to schedule work on it.

```c
static exception_t
decodeSchedControl_CoreStatus(cap_t cap, word_t *buffer)
{
    tcb_t *thread = NODE_STATE(ksCurThread);
    bool_t isOnline =
        isCPUOnline(cap_sched_control_cap_get_core(cap));

    setRegister(thread, badgeRegister, 0);
    setMR(thread, buffer, 0, isOnline);
    setRegister(
        thread,
        msgInfoRegister,
        wordFromMessageInfo(seL4_MessageInfo_new(0, 0, 0, 1))
    );

    return EXCEPTION_NONE;
}
```

Listing 4.6: Kernel implementation of `seL4_SchedControl_CoreStatus`

In LionsOS, the Core Manager API uses this syscall as the single source of truth when checking whether a core is a valid target for offlining, onlining, or migration. This ensures that core-management decisions are synchronised with the kernel's view of which cores have actually joined or left the scheduler, and prevents user space from racing ahead of the secondary-core bootstrap path.

Once `CPU_ON` succeeds and the kernel has marked the core as online, the newly powered core reaches the secondary-core bootstrap entry point, which joins it to the MCS scheduler. At this point, the core is able to run protection domains again, but any PDs that were previously migrated off the core must be restarted explicitly before they can resume useful work. To do this, the Core Manager API provides a `CORE_RESTART_PDS` command that iterates over all protection domains previously running on the target core and reinitialises their register state to a known entry point and stack pointer using the `seL4_TCB_WriteRegisters` syscall. This logic is encapsulated in a small helper function, shown in Listing 4.7.

```c
void restart_pd(microkit_child pd, seL4_Word ep, seL4_Word sp) {
    seL4_CPtr tcb = BASE_TCB_CAP + pd;

    /* Stop the PD before rewriting its user context. */
    seL4_TCB_Suspend(tcb);

    seL4_UserContext c = {0};
    c.sp = sp;
    c.pc = ep;

    seL4_TCB_WriteRegisters(
        tcb,
        seL4_True,                      // Resume after write
        0,                              // Starting register index
        sizeof c / sizeof(seL4_Word),   // Number of registers
        &c
    );
}
```

Listing 4.7: Restarting a protection domain after `CPU_ON`

Figure 4.4 shows a flow chart illustrating this process, from the Core Manager API receiving a request and invoking `CPU_ON`, through Arm Trusted Firmware executing the bootstrap code, to the point where protection domains on that core are restarted. Only the Core Worker PD should be restarted at this stage, because all other protection domains and their interrupts are expected to have been migrated off the core before it was powered down.

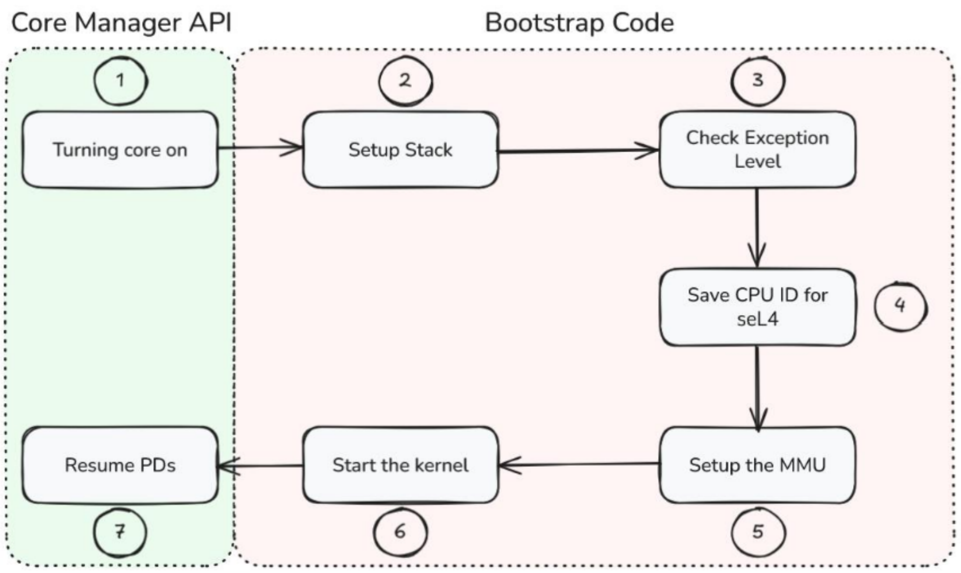


Figure 4.4: Core reactivation flow between the core manager and Arm Trusted Firmware.

Importantly, this step does not restart the kernel from scratch. Instead, execution enters the secondary-core bootstrap entry point, which marks the core as online and reinvokes the MCS scheduler.

4.5 Intermediate Power States

While being able to turn a CPU core fully off and back on is useful, this also comes with high entry and exit latencies, which we explore further in Chapter 5. As a result, these states are rarely suitable when tight timing constraints must be met. In practice, `CPU_ON` and `CPU_OFF` are not intended to be used as fine-grained energy-management primitives, but rather for hot-plugging CPU cores that are expected to remain unused for a long period of time. For shorter idle intervals, the PSCI interface instead provides `CPU_SUSPEND`, which can place cores into either a *powerdown* or *standby* state, as discussed in Chapter 2.

In the powerdown state, the core behaves very similarly to a full `CPU_OFF`. The core's architected state is lost, and it will only resume execution via the warm-entry bootstrap path described in Section 4.4. The key difference is how it is brought back: a core that has entered powerdown via `CPU_SUSPEND` resumes in response to a hardware wakeup event, such as an interrupt, rather than an explicit `CPU_ON` call. The flow from user space into Arm Trusted Firmware and back via a wakeup event is illustrated in Figure 4.5.

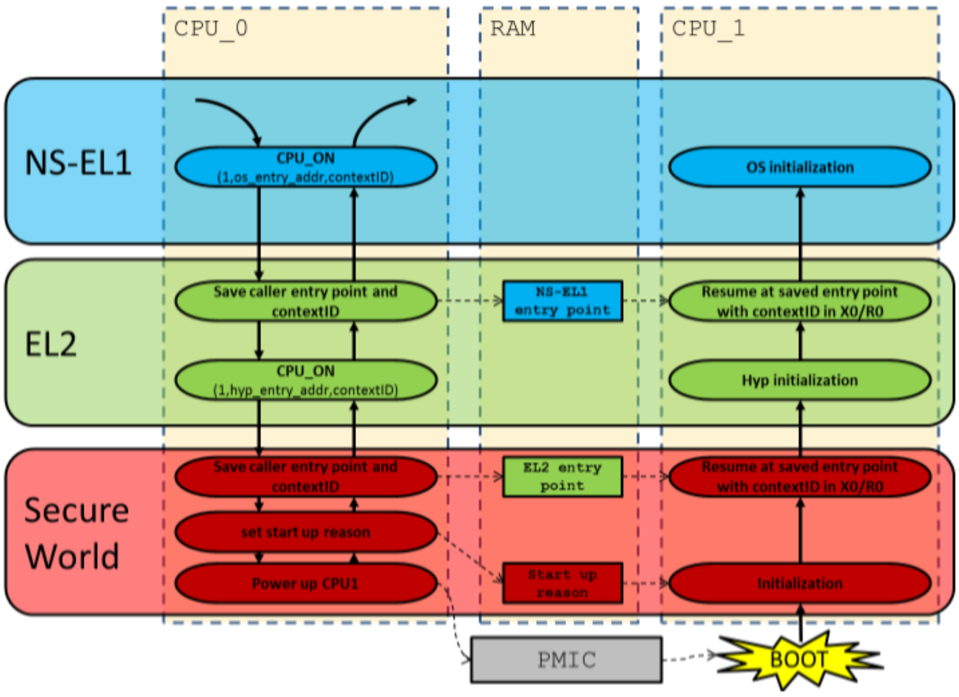

CPU_0
RAM
CPU_1
NS-EL1
CPU_ON
(1,os_entry_addr,contextID)
OS initialization
EL2
Save caller entry point and contextID
NS-EL1 entry point
Resume at saved entry point with contextID in X0/R0
CPU_ON
(1,hyp_entry_addr,contextID)
Hyp initialization
Secure World
Save caller entry point and contextID
EL2 entry point
Resume at saved entry point with contextID in X0/R0
set start up reason
Start up reason
Power up CPU1
Initialization
PMIC
BOOT

Figure 4.5: Powerdown control flow from user space to EL3 and back, where execution resumes after a wake-up event [Arm Limited 2025b].

In contrast, the standby state is a retention state. The core retains architected state and, once a wakeup event arrives, execution continues at the instruction following the `CPU_SUSPEND` call. This drastically reduces exit latency compared to powerdown or a full off/on cycle, because there is no need for a warm boot through Arm Trusted Firmware or the kernel's secondary-core initialisation. However, this benefit depends heavily on how the platform vendor implements standby. On the boards evaluated in this thesis, the standby state provided by PSCI is effectively no more than a `wfi` in exception level 3; the seL4 idle thread already executes `wfi` whenever a core has no runnable threads, so a standby state that only performs a `wfi` offers little additional advantage over remaining in the normal idle loop. Standby becomes more interesting on platforms where the vendor maps it to a deeper, implementation-specific retention mode that goes beyond what the kernel's idle thread can achieve on its own.

Regardless of whether `CPU_SUSPEND` is used to enter powerdown or standby, the kernel must ensure that no interrupts target the core while it is attempting to suspend. If a timer interrupt arrives while the core is in the process of executing `CPU_SUSPEND`, the firmware may reject the request due to an in-flight interrupt, or the interrupt may be lost. In LionsOS, this is handled inside the seL4 PSCI call path: whenever the kernel sees a PSCI call that will remove the current core from service (either powerdown or standby), it marks that core as offline in the global 64-bit bitmap of online CPUs and disables the kernel timer IRQ for that core before issuing the SMC to Arm Trusted Firmware. If the call represents a standby state, the PSCI call returns in place and the kernel then reverses this bookkeeping by re-enabling the timer IRQ and marking the core as online again. The essence of this logic is shown in Listing 4.8, where `cpu_standby` is true for retention states.

```c
cpu_id_t cpu_index = getCurrentCPUIndex();

if (a0 == PSCI_CPU_OFF || a0 == PSCI_CPU_SUSPEND) {
    setCPUOffline(cpu_index);
    setIRQState(IRQInactive,
                CORE_IRQ_TO_IRQT(cpu_index, KERNEL_TIMER_IRQ));
}

/* Issue the PSCI call in x0..x7. */
asm volatile("smc #0\n" : /* clobbers omitted */);

if (cpu_standby) {
    setIRQState(IRQTimer,
                CORE_IRQ_TO_IRQT(cpu_index, KERNEL_TIMER_IRQ));
    setCPUOnline(cpu_index);
}
```

Listing 4.8: Kernel bookkeeping around PSCI low-power calls

From the kernel's point of view, powerdown is therefore not much better than a full `CPU_OFF`/`CPU_ON` cycle: in both cases, the core is removed from the online-CPU bitmap, loses its architected state, and only re-enters the system through the warm bootstrap path. Exit latency is dominated by firmware and kernel reinitialisation, so powerdown does not significantly improve entry or exit latency compared to turning the core fully off. This motivates the use of standby as the primary intermediate state in LionsOS: it still uses the same interrupt and bitmap bookkeeping to ensure a clean transition, but the PSCI call returns in place and the kernel merely flips the core back to “online” and re-enables its timer interrupt.

At user level, LionsOS exposes these intermediate states via the Core Manager API and the per-core Core Worker protection domains. The Core Manager API accepts `CORE_POWERDOWN` and `CORE_STANDBY` instructions from the Core Manager and forwards them to the appropriate Core Worker as notifications. The Core Worker then interprets these instructions and invokes PSCI `CPU_SUSPEND` on the local core with an appropriate power-state encoding. The `power_down` parameter controls whether the worker requests a deep powerdown state (architected state lost) or a retention state (standby), as shown in Listing 4.9.

```c
static void core_suspend(seL4_Bool power_down)
{
    /*
     * Encode the power level according to the original or extended
     * PSCI format. When power_down = 0 we request a standby state;
     * when power_down = 1 we request a deeper powerdown state.
     */
    seL4_Word power_state =
        power_down << (has_ext_power_state
                       ? PSCI_EXT_STATE_MASK
                       : PSCI_ORIG_STATE_MASK);

    seL4_ARM_SMCContext args = {
        .x0 = PSCI_CPU_SUSPEND,
        .x1 = power_state,
        .x2 = bootstrap_entry, // Resume entry point (for powerdown)
        .x3 = has_ext_power_state
    };
    seL4_ARM_SMCContext response = {0};

    microkit_arm_smc_call(&args, &response);
    print_error(response);
}
```

Listing 4.9: User-level PSCI `CPU_SUSPEND` wrapper in the Core Worker

In summary, LionsOS uses three layers to manage intermediate power states. At the bottom, the kernel maintains a 64-bit bitmap of online CPUs and handles IRQ masking and unmasking around PSCI low-power calls. In the middle, the Core Manager API exposes abstract `CORE_POWERDOWN` and `CORE_STANDBY` operations, ensuring that all protection domains and interrupts have been migrated off a core before it is placed into a deeper state. At the top, user code in the Core Manager can choose between the high-latency powerdown state and the low-latency standby state based on its timing and energy requirements, while relying on the underlying layers to enforce the necessary invariants.

On platforms where standby is implemented as more than a simple `wfi`, this gives the runtime a genuine intermediate option; on platforms where PSCI standby is effectively just a `wfi` in EL3, the main benefit is to provide a portable hook for vendors that do offer deeper retention states.

4.5.1 Transparent Message Delivery in Low-Power States

So far, we have focused on how LionsOS moves cores between online, powerdown and standby states. However, from the perspective of a protection domain, this mechanism should be largely invisible. A PD that sends a notification, or a server PD that periodically polls shared memory from its `init` function, should be able to rely on the same high-level semantics regardless of which power state its core is currently in. For example, if PD A writes a request into a shared buffer and signals a server PD B, it should be able to assume that B will eventually run and service the request, even if B resides on a core that has just been placed into a low-power state.

If we simply allowed a server PD to remain on a core in standby, this transparency would break down. A standby core is typically executing a `wfi` instruction and no longer polling or making progress in user space. One option is to migrate such a server PD onto a different, fully active core before entering standby, as described in Section 4.3. However, this does not cover the case where we wish to keep the PD on its original core and still allow it to be reactivated on demand. It also does not address more dynamic scenarios, such as upgrading a core from standby to powerdown after the system has become more idle.

To handle these cases, LionsOS requires that a core in standby can be explicitly reactivated when work arrives. Concretely, the per-core Core Worker must remain reachable via the notification and scheduling paths, even while its core is in a WFI loop. This allows the Core Manager to wake a core that is in standby whenever it needs to deliver work (for example, a new request for a server PD or a command to change the core's power state), and then either let that work run or reissue a deeper `CPU_SUSPEND` request.

This behaviour is implemented by extending the scheduler's treatment of offline cores. When a core enters standby, the PSCI path marks it as offline in the global bitmap, but interrupts that can act as wakeup events (such as IPIs) remain routable to that core. The notification and remote scheduling paths in the kernel have been updated so that they treat a core that is marked offline as a candidate for rescheduling and IPIs, rather than ignoring it. In particular, the `sendSignal` fast path now wakes a thread not only when it is blocked on a receive, but also when it is bound to a core that is currently offline, and `remoteQueueUpdate` considers an offline target core a reason to trigger a reschedule IPI. For clarity, Listing 4.10 shows a simplified version of these changes; the production kernel code includes additional cases and bookkeeping that are omitted here.

```c
void sendSignal(notification_t *ntfnPtr, word_t badge)
{
    if (notification_ptr_get_state(ntfnPtr) != NtfnState_Idle) {
        return;  /* other cases omitted */
    }

    tcb_t *tcb = notification_ptr_get_ntfnBoundTCB(ntfnPtr);
    if (!tcb) {
        return;
    }

    bool_t blocked =
        thread_state_ptr_get_tsType(&tcb->tcbState)
            == ThreadState_BlockedOnReceive;
    bool_t coreOffline = !isCPUOnline(tcb->tcbAffinity);

    if (blocked || coreOffline) {
        cancelIPC(tcb);
        setThreadState(tcb, ThreadState_Running);
        setRegister(tcb, badgeRegister, badge);
        MCS_DO_IF_SC(tcb, ntfnPtr, {
            possibleSwitchTo(tcb);
        })
    }
}

void remoteQueueUpdate(tcb_t *tcb)
{
    cpu_id_t target = tcb->tcbAffinity;

    if (target == getCurrentCPUIndex() ||
        tcb->tcbDomain != ksCurDomain) {
        return;
    }

    tcb_t *targetThread = NODE_STATE_ON_CORE(ksCurThread, target);

    bool_t targetIdle =
        targetThread == NODE_STATE_ON_CORE(ksIdleThread, target);
    bool_t higherPrio =
        tcb->tcbPriority > targetThread->tcbPriority;
    bool_t targetOffline = !isCPUOnline(target);

    if (targetIdle || higherPrio || targetOffline) {
        ARCH_NODE_STATE(ipiReschedulePending) |= BIT(target);
    }
}
```

Listing 4.10: Treating offline cores as candidates for wakeup and rescheduling (simplified)

Recall from Section 4.4 and Section 4.5 that a core entering standby is marked offline in the kernel's 64-bit bitmap and placed in a low-power state using either `CPU_SUSPEND` or a local `wfi`. When a PD on another core sends a notification to a thread whose affinity is that standby core, the added `!isCPUOnline` checks cause the kernel to generate an inter-processor interrupt. That IPI acts as a wakeup event for the standby core, causing the `wfi` to return. The PSCI path then reacquires the kernel lock, re-enables the timer interrupt, marks the core online again, and the scheduler sees the pending reschedule, allowing the awakened server PD (or the Core Worker) to run and service the request.

This mechanism also allows LionsOS to *upgrade* power states. Suppose the Core Manager initially places a core into standby because it predicts a short idle period, but subsequent global information suggests that the core can safely enter a deeper powerdown state. The Core Manager can send a new instruction to the Core Worker on that core; the notification wakes the core out of standby, the Core Worker runs, and then reissues `CPU_SUSPEND` with a powerdown encoding instead of standby. In other words, the system can revise its power-management decision after the fact, without relying on a timer expiry or external interrupt to naturally wake the core.

By comparison, Linux's `cpuidle` subsystem selects a single idle state each time a CPU enters idle, based on a prediction of how long the CPU will remain idle and the exit latencies of the available states. The governor chooses the deepest state that satisfies residency and latency constraints, but that choice is made at idle-entry time; once the CPU has entered a given C-state, the governor does not attempt to move it into a deeper state until the next idle period. Any adjustment to the chosen state happens when the governor is invoked again, not while the CPU is already idle [Wysocki 2018, The Linux Kernel Developers 2010].

In practice, this means that Linux does not provide a general mechanism to "upgrade" an already-idle CPU from a shallow C-state to a deeper one purely in response to a separate management decision; it would have to wake the CPU, run some code and then let it re-enter idle so that the governor can pick a new state [Wysocki 2018]. The LionsOS design goes a step further by explicitly allowing the Core Manager to wake standby cores via kernel notifications and IPIs, then reissue a different PSCI request. This extra control is particularly useful for cluster-level policies, where one core's decision to enter standby may later be revised based on the aggregate idleness of the system, while still preserving transparent message delivery semantics for protection domains.

4.6 The Core Manager API

The core manager runs as a dedicated Microkit protection domain and exposes a small RPC-style API to the rest of LionsOS. Other components interact with it via a protected channel: they place an instruction code and up to two parameters into message registers and issue a protected call. The `protected()` entry point decodes these requests and dispatches to internal helper functions such as `core_on()`, `core_migrate()` and `manager_restart_pd()`.

Each request is identified by an `Instruction` value, with arguments interpreted as a target core identifier and/or a protection-domain (PD) identifier. The core manager returns a single status code in message register 0, where 0 indicates success and 1 signals failure or a rejected request.

4.6.1 Initialisation

Before servicing any requests, the core manager initialises its internal state in `init()`. It copies a small bootstrap routine into a shared memory region (`bootstrap_vaddr`) that is executable by secondary cores, then cleans and invalidates the corresponding cache lines to ensure that all cores observe the updated instruction stream. The physical entry point of this code is exported as `bootstrap_entry` and later passed to firmware when a core is turned on.

Per-PD metadata is stored in the `pd_infos` array. For each PD it records the currently assigned core, the IRQ bitmask, and its MCS budget and period. This table is updated whenever PDs are migrated.

4.6.2 Core control operations

The core manager provides several operations that directly affect core power state or kernel view of each core:

`CORE_ON` Powers on a secondary core via the PSCI `CPU_ON` call. The caller specifies the target core and the core manager passes `bootstrap_entry` as the bootstrap address. Arm Trusted Firmware brings the core out of reset and starts executing the bootstrap code, which eventually re-enters the kernel and marks the core online.

`CORE_OFF`, `CORE_POWERDOWN`, `CORE_STANDBY` Request that a core enter a low-power state. To preserve system liveness, the core that currently hosts the Monitor PD (`monitor_core`) cannot be powered down. For other cores, the core manager first checks whether the core is online via `core_status()`. If it is online, the manager sends a notification to the worker PD on that core, which is responsible for migrating its local workload and eventually issuing `CPU_OFF` or `CPU_SUSPEND` to firmware. If the core is already offline, the request fails and a diagnostic is printed.

`CORE_STATUS` Returns the kernel's view of a core's state. Internally, `core_status()` calls `seL4_SchedControl_CoreStatus()` on the relevant scheduling-control object and reports 0 for offline cores and 1 for online cores. If needed, the user implementation is responsible for differentiating between which power state the core is in based off the request it sent and the output of `CORE_STATUS`.

The helper `print_error()` decodes PSCI error codes and prints human-readable diagnostics when firmware rejects a `CPU_ON` request. These diagnostics are currently logged to the Microkit debug console rather than propagated back to the caller.

4.6.3 Protection-domain migration

A key responsibility of the core manager is to move PDs between cores while preserving their scheduling parameters and interrupt routing. This is exposed through two instructions:

`CORE_MIGRATE` Migrates a regular PD to a new core. The caller supplies the PD identifier and the target core. The core manager rejects attempts to move core-worker PDs, which are reserved for per-core management. For other PDs, `core_migrate()` reconfigures the PD's scheduling context using `seL4_SchedControl_ConfigureFlags()` on the destination core's scheduling-control object, preserving the existing budget and period stored in `pd_infos`. It then iterates over the PD's IRQ bitmask and updates each associated `IRQHandler` capability with the new core using `seL4_IRQHandler_SetCore()`. Finally, it updates the `pd_infos[pd].pd_core` field to reflect the new placement.

`CORE_MIGRATE_MONITOR` Migrates the Monitor PD to a different core. The Monitor uses a fixed scheduling context (identifier 63) with a period and budget of 1000 µs. `monitor_migrate()` reconfigures this scheduling context on the chosen core and updates `monitor_core` so that later power-down requests cannot accidentally switch off the core hosting the Monitor.

To support clean recovery after onlining a core, the API also includes:

`CORE_RESTART_PDS` Restarts all PDs assigned to a given core. The core manager scans `pd_infos` for entries whose `pd_core` matches the requested core and calls `manager_restart_pd()` for each one. Instead of using Microkit's `pd_restart()` primitive, this helper directly suspends the PD's TCB and overwrites its user context (PC and SP) with the initial entry point and stack pointer, then resumes the thread. This avoids a race where a PD might execute with stale register state immediately after `CPU_ON`.

These migration and restart primitives are central to maintaining location transparency. PDs themselves are unaware of their physical placement; the core manager adjusts scheduling contexts and interrupt affinities so that communication continues to work regardless of which core currently hosts a PD.

4.6.4 Monitoring and PMU support

The current API includes two simple operations that use the per-core worker PDs to collect and reset performance counters. These interfaces are placeholders and will be substantially overhauled in Chapter 6, where we introduce a richer measurement and policy framework.

`CORES_QUERY` Returns a coarse snapshot of per-core activity, such as cycle counts, aggregated from the worker PDs running on each online core. This information is intended for lightweight monitoring and to inform higher-level core-management policies.

`CORES_RESTART_PMU` Requests that all worker PDs reset their performance monitoring units, so that a new measurement interval can begin in a coordinated fashion across cores.

Together, these operations form a minimal yet expressive API for controlling core power state, migrating PDs across cores, and collecting per-core performance information. Higher-level LionsOS policies invoke this API to implement energy-aware and load-aware core management while preserving the location transparency of the underlying component model.

4.7 A Simple Core Manager

To demonstrate how the core-manager subsystem can be used in practice, the seL4 Microkit tool includes a simple user-level Core Manager protection domain that implements a concrete policy on top of the Core Manager API. This Core Manager runs as a regular Microkit PD and communicates with the Core Manager API via protected procedure calls on a dedicated channel. It provides both an interactive interface over UART and an automatic mode that periodically samples per-core utilisation and places idle cores into deeper power states.

In manual mode, the Core Manager behaves as a command-line tool. It accepts commands such as `status`, `dump`, `migrate`, `powerdown`, `standby` and `on`, parsing arguments from a small line-buffer and issuing the corresponding instructions to the Core Manager API using `send_core_command`. For example, `status <core>` queries the core's current power state, `migrate <pd> <core>` asks the API to move a PD to a different core (updating both its scheduling context and IRQ routing as described in Section 4.3), and `powerdown <core>` requests that a core be placed into the deep `CPU_SUSPEND` powerdown state rather than being fully turned off with `CPU_OFF`. In this prototype, powerdown is used as the primary deep state for energy saving, with full offlining reserved for more specialised use cases.

4.7.1 Automatic Policy Using PMU Feedback

In addition to manual control, the Core Manager can operate in an automatic mode in which a periodic timer interrupt drives a simple utilisation-based policy. When automatic mode is enabled, the timer PD delivers a notification on `TIMER_CHANNEL`; on each tick, the Core Manager issues a `CORES_QUERY` request to the Core Manager API to obtain a snapshot of per-core utilisation:

```c
void handle_timer_tick(void)
{
    /* Ask the Core Manager API for per-core cycle counts. */
    microkit_mr_set(0, CORES_QUERY);
    microkit_ppcall(API_CHANNEL, microkit_msginfo_new(0, 1));

    uint64_t util[NUM_CPUS];
    for (int i = 0; i < NUM_CPUS; i++) {
        util[i] = microkit_mr_get(i);
    }

    const uint64_t THRESHOLD = 100000000ULL; // 10% of 1e9 cycles

    for (int core = 0; core < NUM_CPUS; core++) {
        seL4_Word state =
            send_core_command(CORE_STATUS, core, 0);

        if (state == CORE_ON &&
            util[core] < THRESHOLD &&
            cores_on > 1) {

            // Migrate all PDs away from this under-utilised core.
            int target = find_best_core(core, util);
            if (target < 0) {
                continue;
            }

            migrate_all_pds(core, target);
            if (core == monitor_core) {
                migrate_monitor(target);
            }

            send_core_command(CORE_POWERDOWN, core, 0);
            cores_on--;
        }
    }
}
```

Listing 4.11: Using `CORES_QUERY` and PMU feedback to decide powerdown

The `CORES_QUERY` call closes the loop between measurement and control. From the Core Manager's perspective, these counts form a simple utilisation signal over the last timer period, which it uses to decide when a core is "cold" enough to justify migrating its workloads and powering it down.

This creates a feedback loop spanning three layers:

1. **Measurement (Core Worker).** On each `CORES_QUERY`, the Core Worker reads `PMU_CYCLE_CTR`, returns the cycle count and resets the counter. This provides a per-core measure of how many cycles were spent executing between sampling points.

2. **Aggregation (Core Manager API).** The API polls all Core Workers, aggregates their cycle counts into an array, and returns this array to the Core Manager via message registers.

3. **Policy (Core Manager).** The Core Manager interprets the cycle counts as approximate utilisation, selects under-utilised cores, migrates their PDs to more active cores, and finally sends `CORE_POWERDOWN` instructions to the affected cores via the Core Manager API. Those instructions are delivered to the relevant Core Workers, which in turn invoke PSCI `CPU_SUSPEND` in powerdown mode on their local cores.

Because the same path is used for both power-state changes and utilisation sampling, the Core Manager can easily refine its policy. For example, it could adjust the utilisation threshold, bias migration towards cores with similar load, or prefer placing frequently-woken server PDs on cores that are kept in standby rather than powerdown. The design remains modular: the Core Manager implements policy in user space, the Core Manager API provides a stable, platform-agnostic interface to PSCI and kernel features, and the Core Workers encapsulate the low-level PMU and power-state transitions on each core.

At the same time, the current mechanism for passing PMU values through this stack is intentionally minimal. Cycle counts are exported as raw per-core scalars via message registers, with no richer structure or coordination beyond what is needed for this prototype. This simple wiring is primarily intended as a starting point for follow-on work by others; in Chapter 6, we outline future directions for redesigning how PMU-derived information is represented and propagated, so that more sophisticated policies can be built on top of the same core-management infrastructure.

Chapter 5

Evaluation

In this chapter we present an experimental evaluation of our core-management mechanisms on real hardware. We first measure entry and exit latencies for low-power states on the MaaXBoard, a low-cost single-board computer based on the NXP i.MX 8M processor. We then characterise the power draw of the AMD Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit under a range of operating conditions, with a particular focus on different power states and core configurations. Finally, we discuss the limitations of our benchmarking setup, interpret the results, and highlight open issues that motivate the future work outlined in Chapter 6.

5.1 Benchmarking Latency

5.1.1 The Arm Generic Timer

All latency measurements in this chapter rely on the Arm Generic Timer as a common time base as illustrated in Figure 5.1. The Generic Timer consists of a 64-bit, monotonically increasing system counter driven by a constant-frequency clock in the always-on power domain. Its value continues to advance regardless of individual core power state, and the counter is visible from all relevant exception levels, which makes it well suited for timing transitions that cross OS and firmware boundaries.

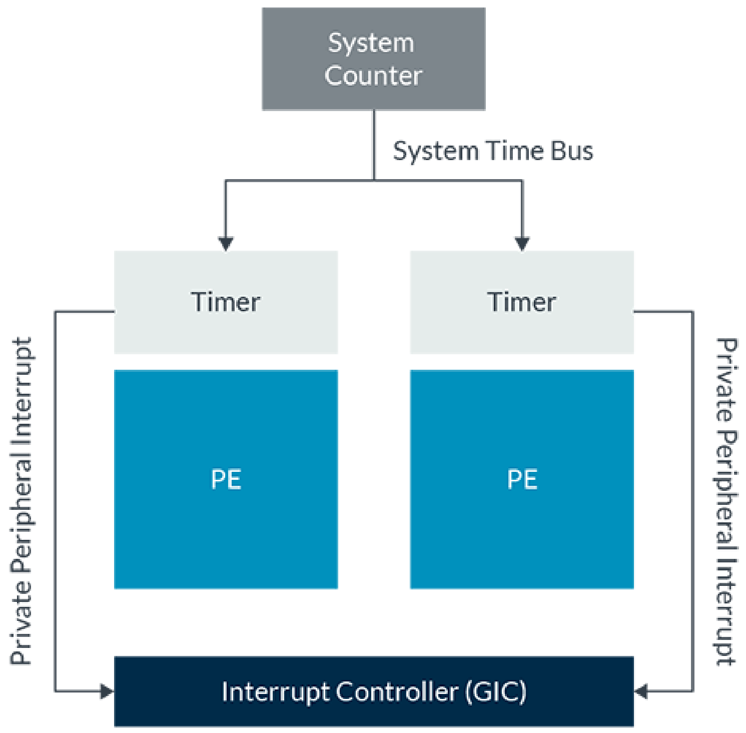

System Counter
System Time Bus
Timer
Timer
PE
PE
Private Peripheral Interrupt
Private Peripheral Interrupt
Interrupt Controller (GIC)

Figure 5.1: Arm system counter and Generic Timer view [Arm Limited 2025a].

Architecturally, the timer can be read through two main views:

- **Physical counter** (`CNTPCT_EL0`): exposes the raw system counter value. This counter is global and is not affected by virtualisation offsets.
- **Virtual counter** (`CNTVCT_EL0`): exposes the same counter plus an offset programmed in `CNTVOFF_EL2`. This allows a hypervisor to present a shifted time base to guest operating systems.

On a virtualised system the virtual counter is the natural choice for guests, since the hypervisor can independently adjust the offset per VM. In our case, however, we need timestamps that are directly comparable across EL0 (Core Manager System), EL2 (seL4 kernel) and EL3 (Arm Trusted Firmware). Any non-zero virtual offset would introduce a constant skew between readings at different exception levels. For this reason, our implementation samples the physical counter via `CNTPCT_EL0` wherever possible, and configures the system so that all participants see the same underlying system counter. This ensures that differences between timestamps reflect only real latency rather than virtualisation artefacts.

5.1.2 Instrumentation methodology

To capture end-to-end timings for power-state transitions, we reserve a small shared memory region at a fixed physical address. This region is mapped into three privilege domains: the Core Manager System at EL0, the seL4 kernel at EL2, and Arm Trusted Firmware at EL3. Each participant writes its own timestamps into this buffer using the Arm Generic Timer as a common reference. The buffer layout is fixed, and each stage writes its entries in a well-defined order, so no explicit synchronisation is required.

For core power-down (using `CPU_SUSPEND` with power-down), we instrument the following points:

- At EL0, the Core Manager System records a timestamp immediately before issuing the protected procedure call that requests a core operation. This marks the moment the core-management request enters the system.
- At EL2, the seL4 kernel records a second timestamp just before invoking the PSCI SMC call, capturing the boundary between kernel execution and firmware entry.
- At EL3, Arm Trusted Firmware records one timestamp immediately before executing the final `WFI` instruction that actually powers the core down, and another timestamp immediately after waking from `WFI`. These samples capture the firmware-visible entry and exit points of the low-power state.
- After the core completes its secondary boot path and rejoins the scheduler, the seL4 kernel records a final timestamp, indicating when the kernel has finished reinitialising the core and is ready to run user-level code again.

The resulting sequence of timestamps for each power-down cycle gives a detailed breakdown of the contribution of userspace, kernel, firmware entry and exit, and the secondary-boot path. Differences between successive timestamps correspond to distinct phases of the transition, which we analyse later in this chapter.

For standby transitions (using `CPU_SUSPEND` with standby) we follow a similar approach, but the core does not execute the full secondary boot path on wake-up. Instead, the seL4 kernel records a timestamp immediately after the PSCI SMC call returns from firmware, and the Core Manager System at EL0 records a final timestamp once its own SMC wrapper returns. In this configuration, exit latency is measured from the point where firmware resumes normal execution back into the kernel, up to the point where control returns to the core manager, allowing a direct comparison with the deeper power-off transitions.

5.1.3 Results

Running these tests on the MaaXBoard, a low-cost single-board computer based on the NXP i.MX 8M processor, yields the following results. Figure 5.2 shows the entry and exit latencies, in cycles, of the standby state, compared to a baseline path where user space invokes a core-manager call that traps into the kernel, but the SMC instruction has been removed so the path never leaves the kernel and simply returns to user space without entering standby.

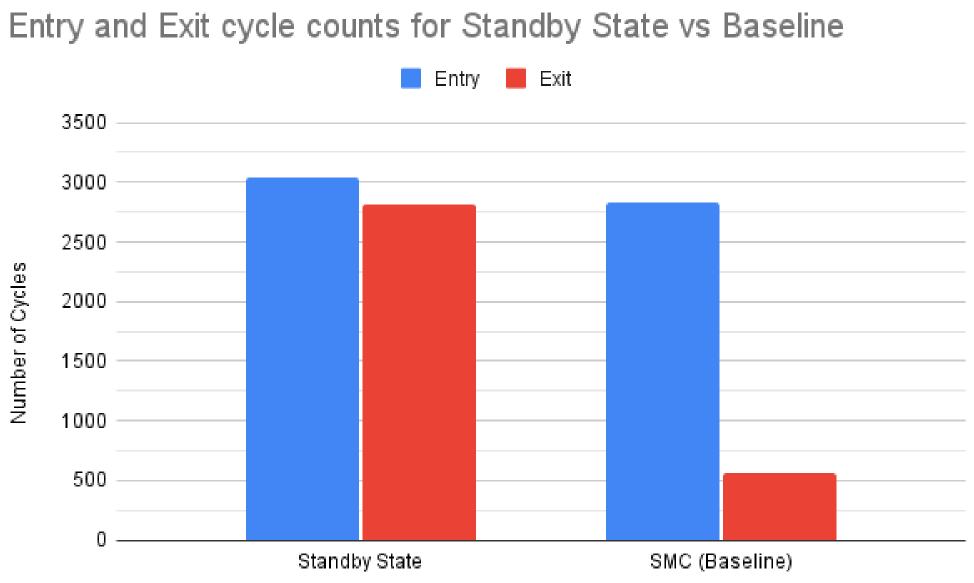


Figure 5.2: Cycle count comparison of the standby state and the default core-manager call path from EL0 to EL1/EL2 without entering EL3.

We observe that the entry latency increases by only around 200 cycles relative to the baseline, whereas the exit latency increases by more than 2000 cycles when returning from standby. This behaviour is plausible: on entry, the only additional work is executing the SMC instruction and running a small amount of Arm Trusted Firmware code to prepare for `CPU_SUSPEND` before executing `WFI`. On exit, however, the baseline path simply returns from the kernel back to user space, while the standby case must wake from `WFI`, run the secure firmware's warm-entry path, restore context at EL3, and then return via the kernel to user space, adding a much larger fixed cost to the exit path.

Table 5.1 summarises the median entry and exit latencies, in cycles, for all measured states. The MaaXBoard runs the seL4 Microkit at a fixed 1 GHz, so each cycle corresponds to 1 ns. The baseline SMC entry and exit therefore take approximately 2.88 µs and 0.60 µs respectively, while entering and exiting standby take around 3.12 µs and 2.76 µs. The powerdown state is significantly more expensive, with an entry cost of roughly 0.51 ms and an exit cost of about 3.58 ms.

	Entry Cycles	**Exit Cycles**
SMC (Baseline)	2880(364)	600(58)
Standby State	3120(147)	2760(59)
Powerdown State	511500(10273)	3582600(797)

Table 5.1: Median entry and exit latencies (in cycles) for all measured states on the MaaXBoard at 1 GHz.

Two trends are evident. First, standby adds only a small overhead to the entry path relative to the baseline SMC call (about 8% more cycles), but increases the exit cost by roughly a factor of four. This reflects the additional work required to resume the core from a low-power state, even when only core-local state is affected. Second, the powerdown state is two to three orders of magnitude more expensive than standby on both entry and exit. This suggests that powerdown is only worthwhile when the expected idle interval is long enough to amortise the additional transition cost; otherwise, remaining in standby (or even in the active state) may be preferable from an energy perspective.

Opportunities for improving powerdown latency

While the qualitative ordering of the states is as expected (powerdown > standby > baseline), the absolute exit latency of the powerdown state on this platform is higher than ideal. Exiting powerdown currently costs around 3.6 ms, which is significantly larger than a typical scheduling quantum, and therefore constrains how aggressively the core manager can use `CPU_SUSPEND` with powerdown for shorter idle intervals.

A large part of this overhead arises from the way seL4 currently brings a powered-down secondary core back online. When a core returns from `CPU_SUSPEND` with powerdown, the firmware re-enters the kernel via the same entry point that is used during initial boot. The kernel then follows the secondary-core bootstrap path (`try_init_kernel_secondary_core`), which in turn calls `init_cpu()`. This path reactivates the kernel vspace, reinitialises the kernel stack, reinstalls the exception vector table, probes for floating-point hardware and disables it, reinitialises the local interrupt controller and per-CPU timer, and re-exports selected CPU features to user mode. These steps are appropriate for a cold boot, but after powerdown the kernel image and its data structures already reside in RAM and the existing virtual memory configuration is still valid.

There is therefore room to streamline the secondary-core resume path for the powerdown case. Possible improvements include:

- avoiding redundant FPU discovery and re-disabling of the floating-point unit when the relevant control registers are known to retain state across powerdown;
- minimising repeated interrupt and timer configuration, for example by only re-enabling the per-CPU timer IRQ rather than resetting all PPI state; and

- separating the cold-boot initialisation sequence from a lightweight resume sequence that assumes the global kernel state has already been established by the primary core.

A dedicated “resume from powerdown” path that skips these redundant steps would reduce the fixed cost of `powerdown` exit and make deeper states attractive over a wider range of idle intervals.

5.2 Benchmarking Power Draw

5.2.1 Methodology

To complement the latency results from the MaaXBoard, we evaluate how different core power states affect the power draw of the AMD Xilinx Zynq UltraScale+ MPSoC on the ZCU102 evaluation board. The Zynq UltraScale+ splits the device into a processing system (PS), which contains the Cortex-A53 cores, memory controllers and on-chip interconnect, and a programmable logic (PL) region implemented in the FPGA fabric [AMD 2025a]. The ZCU102 routes both PS and PL supplies through INA226 current and power monitors via an I^2C multiplexer on the board [AMD 2023]. In this work we focus on the PS rail, since the PL is unused and remains in a dormant state.

Power draw is measured using an external microcontroller attached to the on-board INA226 via I^2C. The microcontroller runs a program that repeatedly selects the PS channel, reads the INA226 power register and streams the results over a high-speed serial link to a host. Each sample consists of a timestamp (from the microcontroller’s `micros()` counter) and the PS rail power, and is written by a Python script into CSV files.

The INA226 is configured using calibration values derived from the ZCU102 reference design so that each increment of the power register corresponds to a power step on the order of a few milliwatts on the PS rail [Texas Instruments 2024, AMD 2016]. This calibration gives sufficient dynamic range to cover the full operating envelope of the board while still resolving small changes in the PS power draw.

The ZCU102 runs the seL4 Microkit with the Core Manager System from Chapter 4. All cores execute the same simple workload: a loop that spends the vast majority of its time in `WFI` at EL2, woken periodically by the core manager’s timer to handle housekeeping and potential core-management requests. When there is no runnable work on a core, seL4 schedules that core’s idle thread, which in turn executes `WFI` as described in Section 4.5. The hypervisor runs with a fixed clock frequency and without any changes to CPU frequency. This setup is consistent with the PSCI model where per-core idle states and higher-level cluster or system states are coordinated by platform firmware [AMD 2025b].

We collect two classes of traces:

- **Baseline traces (all cores idle)**: all four Cortex-A53 cores remain online, and whenever the Core Manager System has no work for a core, seL4's per-core idle thread (Section 4.5) runs and repeatedly executes `WFI` at EL2, servicing only the minimal interrupts required by the core manager. These runs characterise the steady-state power draw of the PS rail when the cluster is fully powered but all application work has quiesced and only idle threads are runnable. Similar methodology, focusing on steady-state idle behaviour of SoCs, is common in prior work on multicore and many-core power modelling [Le Sueur and Heiser 2010, David, Gorbatov, Hanebutte, Khanna, and Le 2010].
- **Mixed traces (alternating core configurations)**: the core manager periodically transitions cores into and out of a deeper idle state via PSCI calls, alternating between an "all cores idle" configuration and a configuration in which one core remains online and idle (again running the seL4 idle thread in `WFI` when there is no work), while the remaining cores invoke `CPU_SUSPEND` with the `powerdown` parameter. Each configuration is maintained for long enough that the power draw stabilises before the next transition, and this pattern repeats throughout the run. The Core Manager System is the only significant load on the cores, so differences between the two configurations mainly reflect differences in per-core power state, while the PS uncore and memory subsystem remain dominated by background activity.

A Python script analyses the raw CSV files offline. The script parses the PS power column, discards malformed rows and obvious outliers (for example occasional corrupted lines that decode as multi-kilowatt readings), and then computes basic descriptive statistics: sample count, mean, median, sample standard deviation, minimum, maximum and range. For mixed traces, the script first computes the median PS power across the entire run and then splits samples into two groups:

- an *above-median group* containing all samples with P_{PS} greater than the median; and
- a *below-median group* containing all samples with P_{PS} less than or equal to the median.

This simple median split is sufficient in practice because the mixed traces are constructed to spend roughly half of the time in each configuration, and the PS power distribution is close to bimodal. The script reports statistics for each group separately and then compares each group to the baseline using a normalised effect size (Cohen's d), which expresses the difference in means in units of pooled standard deviation [Cohen 1988, Ellis 2010]. Our primary quantity of interest is the difference between the below-median group and the baseline, since this directly reflects the effect of placing three of the four cores into the PSCI `CPU_SUSPEND` powerdown state when they would otherwise be idle in `WFI`. The above-median group acts as a consistency check that the mixed run contains a mode that is statistically indistinguishable from the all-cores-idle baseline.

To reduce the influence of transient effects during state transitions, we optionally discard mixed samples whose PS power exceeds the maximum observed during the baseline run, treating these as entry/exit spikes rather than steady-state measurements. Because the INA226 exposes quantised power steps, this filtering and grouping effectively separate the discrete levels corresponding to different steady configurations, while leaving enough samples in each group to average out measurement noise.

5.2.2 Results

In the baseline configuration, with all four cores online and idle in `WFI` by virtue of seL4's per-core idle threads (Section 4.5), the PS rail power readings form a narrow, single-peaked distribution centred at approximately 1.05 W, with only a few milliwatts of variation across samples.

For the mixed configuration, the PS measurements separate into two clearly distinguishable bands. One band aligns closely with the baseline and corresponds to periods where all four cores are online and idle, again running the seL4 idle threads. The other band sits slightly below it and corresponds to the configuration where three of the four cores have entered the PSCI `CPU_SUSPEND` powerdown state while one remains idle in `WFI`. On a representative pair of runs, the baseline PS rail (all cores idle) has a mean power draw of about 1.05 W with a standard deviation of roughly 0.002 W, while the lower band in the mixed trace (one idle core, three suspended with `CPU_SUSPEND` + `powerdown`) has a mean of about 1.04 W with a standard deviation of around 0.004 W. In other words, the means differ by roughly 0.01 W (about 10 mW), corresponding to just under 1% of the baseline, and the lower band occupies a distinct but nearby set of INA226 quantisation steps below the baseline. The upper band in the mixed trace is concentrated at the upper end of the baseline range, around 1.056 W, consistent with a second category where all cores remain online and the monitor reports the top discrete bin within the observed baseline envelope.

Table 5.2 summarises the PS statistics for one baseline run and the two bands of a mixed run, using rounded figures that reflect the scale of the INA226 resolution and the observed spread. The baseline PS rail shows a mean of 1.05 W with a standard deviation of 0.002 W and a total range of only 0.0125 W, reflecting a very stable idle state. The lower band in the mixed trace exhibits a slightly lower mean (around 1.04 W) and a somewhat larger spread (standard deviation around 0.004 W) because it aggregates several adjacent INA226 steps in the lower band.

Run / Group	**Mean PS Power (W)**
Baseline (all cores idle)	1.051(2)
Lower band (1 core idle, 3 suspended)	1.044(4)

Table 5.2: Summary of PS rail power for the ZCU102 baseline and mixed runs. Values summarise multiple runs; absolute values depend on the experimental setup and INA226 calibration.

From a statistical perspective, the physically meaningful comparison is between the lower PS band (the below-median group) in the mixed run and the baseline run, since both correspond to well-defined, steady-state configurations of the A53 cluster. The upper band in the mixed run essentially re-samples the all-cores-idle configuration at the top INA226 bin and mainly serves to show that the grouping procedure is separating the two categories cleanly. Comparing the lower band to the baseline, the mean PS rail power draw is lower by around 0.01 W, while the standard deviations are on the order of 0.002–0.004 W. Expressed in normalised terms (for example using Cohen's d), this corresponds to a difference of a few pooled standard deviations, so the reduction is statistically clear despite being less than 1% in relative terms.

Taken together, these measurements indicate that, for **this particular board and configuration**, the benefit of using `powerdown` over leaving idle cores in `WFI` is modest in terms of PS rail power draw: placing three of four A53 cores into the `CPU_SUSPEND` powerdown state reduces the PS rail draw by just under 1% compared to an all-cores-idle baseline where all work has drained to seL4's idle threads. This modest gain is largely a consequence of the ZCU102's board- and firmware-level power architecture. The PS rail on this FPGA evaluation board feeds a substantial amount of uncore logic (interconnect, memory controllers, DDR, system peripherals) and associated support circuitry whose leakage and dynamic components dominate overall power draw even when the A53 cores are idle [AMD 2025a, AMD 2016, AMD 2025b].

In addition, the Zynq UltraScale+ PMU firmware used on ZCU102 does not fully collapse the higher-level PS domains for the `CPU_SUSPEND` state used here; instead, the PSCI implementation maps this call to a CPU-local idle state in which the core is powered down but cluster and PS infrastructure remain active [AMD 2025b]. Once the cores are already clock-gated in `WFI`, the incremental saving from suspending them is therefore small relative to this always-on baseline. Deeper states may still be important on platforms that can genuinely gate or collapse larger PS and uncore domains [Le Sueur 2011], or when the system makes more aggressive use of PSCI composite states, but on ZCU102 these results highlight that a large fraction of idle power originates from uncore and board-level components that are not affected by per-core power management, and that careful measurement is needed to quantify the true impact of core-level policies.

Chapter 6

Conclusions

In conclusion, this thesis presents a global core management system on top of the seL4 Microkit that is ready to be ported directly to LionsOS. The design introduces a small set of mechanisms for globally coordinating core states and protection-domain placement, while keeping policy in a separate, replaceable Core Manager component. By encapsulating PSCI operations and core-local workers inside dedicated Microkit protection domains, the system preserves seL4's strong isolation properties and allows LionsOS to adopt energy-aware core management without exposing platform-specific details to ordinary components.

The experimental evaluation on real hardware clarifies the practical trade-offs among the available idle states. On the MaaXBoard, measurements of PSCI-driven standby and powerdown show that standby adds only microsecond-scale latency relative to a baseline SMC call, whereas powerdown incurs millisecond-scale entry and exit times. On the ZCU102, steady-state PS-rail measurements show that placing three of four cores into the `CPU_SUSPEND` powerdown state reduces PS rail draw by just under one percent compared to an all-cores-idle baseline; although small in absolute terms, this reduction is statistically significant and is achieved on a board whose substantial uncore logic and support circuitry remain powered in the firmware.

These results highlight both the value and the limits of core-centric management on contemporary SoCs. The mechanisms developed here are deliberately simple, platform-agnostic and compact, making them practical to review and reason about, and they provide LionsOS with a concrete substrate on which richer policies can be built. Future work can extend this foundation by integrating more sophisticated predictors, coordinating core states with DVFS and peripheral gating, and evaluating the system under realistic LionsOS workloads across a broader range of hardware platforms.

6.1 Future Work

There are several natural extensions to this work. The current prototype targets Armv8-A platforms that implement PSCI, and relies on the existing seL4 Microkit SMP loader and Arm Trusted Firmware hooks. A first direction is to port the core management mechanisms to other architectures, particularly RISC-V and x86. This will require Microkit support for symmetric multicore initialisation on RISC-V and the forthcoming CapDL-based Microkit for x86. Once those pieces are available, the core-management design can be reused largely unchanged, with only the low-level bring-up and power-state interfaces replaced by the corresponding platform mechanisms.

A second direction is to integrate a more mature performance-monitoring infrastructure. The current PMU support in the Core Manager System is deliberately minimal and tailored to simple experiments, providing only basic counts of cycles and retired instructions. The Trustworthy Systems group is developing a reusable PMU library for seL4-based systems; future work will involve migrating the core manager to this library so that it can access a richer set of events, share PMU hardware safely with other components, and reduce duplication of measurement code.

Finally, it would be valuable to tie core management more tightly to LionsOS's user-level scheduling framework. One option is to embed the core manager as part of a LionsOS scheduler, letting core-state decisions be driven directly by knowledge of runnable threads rather than by indirect PMU heuristics. This could also remove the current requirement to migrate protection domains away from a core before powering it down: at present, if a protection domain remains resident and receives a notification while its core is in the `CPU_SUSPEND` powerdown state, the wake-up trap to EL3 and subsequent kernel restart will discard that notification. If core management logic lived inside the scheduler, it could track pending notifications across suspend and resume, and reissue them once the kernel and user-level scheduler state have been restored from RAM. This would allow more flexible policies that keep protection domains pinned to particular cores while still safely exploiting deeper idle states.

Bibliography

[AMD 2016] AMD (2016). *ZCU102 Evaluation Board Schematics.* Contains power tree and INA226 connections for PS and PL rails. AMD. URL: https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/73/2474.schematic.pdf.

[AMD 2023] AMD (2023). *Zynq UltraScale+ MPSoC ZCU102 Evaluation Board User Guide.* UG1182. Available from AMD/Xilinx documentation portal. AMD. URL: https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd.

[AMD 2025a] AMD (2025a). *Zynq UltraScale+ Devices Technical Reference Manual.* UG1085. Describes PS/PL partitioning, power domains and APU architecture. AMD. URL: https://docs.amd.com/v/u/en-US/ug1085-zynq-ultrascale-trm.

[AMD 2025b] AMD (2025b). *Zynq UltraScale+ MPSoC Software Developer Guide.* UG1137. Describes power states, PMU firmware and domain control. AMD. URL: https://docs.amd.com/r/en-US/ug1137-zynq-ultrascale-mpsoc-swdev.

[Arm Limited 2025a] Arm Limited (2025a). *Arm Architecture Reference Manual Supplement: Generic Timer. System Counter.* Revision 0104. URL: https://developer.arm.com/documentation/102379/0104/System-Counter (visited on 12/03/2025).

[Arm Limited 2025b] Arm Limited (Mar. 2025b). *Arm Developer Documentation.* Arm Limited. Cambridge, UK. URL: https://developer.arm.com/documentation.

[Arm Limited 2025c] Arm Limited (2025c). *Arm DynamIQ Technology.* Corporate Website. Accessed: 2025-04-22. Cambridge, UK: Arm Limited. URL: https://www.arm.com/technologies/dynamiq.

[Bortolotti, Tinti, Altoé, and Bartolini 2016] Daniele Bortolotti, Simone Tinti, Piero Altoé, and Andrea Bartolini (2016). "User-space APIs for dynamic power management in many-core ARMv8 computing nodes". In: *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 675–681. DOI: 10.1109/HPCSim.2016.7568400.

[Borza 2021] Juraj Borza (Apr. 2021). *(Mis)understanding RISC-V ecalls and syscalls.* Personal blog. URL: https://jborza.com/post/2021-04-21-ecalls-and-syscalls.

[Carroll 2017] Aaron Carroll (May 2017). “Understanding and Reducing Smartphone Energy Consumption”. PhD Thesis. Sydney, Australia. URL: `https://trustworthy.systems/publications/papers/Carroll%3Aphd.pdf`.

[Carroll and Heiser 2010] Aaron Carroll and Gernot Heiser (June 2010). “An analysis of power consumption in a smartphone”. In: NICTA
UNSW
Open Kernel Labs. Boston, MA, US, pp. 271–284. URL: `https://trustworthy.systems/publications/papers/Carroll_Heiser_10.pdf`.

[Chiang 2009] Alex Chiang (Oct. 30, 2009). *sysfs CPU scheduler power saving controls. ABI documentation for /sys/devices/system/cpu/sched_mc_power_savings and /sys/devices/system/cpu/sched_smt_power_savings.* Linux kernel documentation. URL: `https://git.zx2c4.com/linux-rng/commit/Documentation/ABI/testing/sysfs-devices-system-cpu` (visited on 11/22/2025).

[Cohen 1988] Jacob Cohen (1988). *Statistical Power Analysis for the Behavioral Sciences.* 2nd ed. Hillsdale, NJ, US: Lawrence Erlbaum Associates.

[David, Gorbatov, Hanebutte, Khanna, and Le 2010] Hadas David, Eitan Gorbatov, Udi R. Hanebutte, Rahul Khanna, and Christian Le (2010). “RAPL: Memory Power Estimation and Capping”. In: *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA).* Shows that package/uncore components form a significant fraction of total chip power. IEEE, pp. 189–197.

[Dennis and Van Horn 1966] Jack B. Dennis and Earl C. Van Horn (Mar. 1966). “Programming semantics for multiprogrammed computations”. In: *Commun. ACM* 9.3, pp. 143–155. ISSN: 0001-0782. DOI: `10.1145/365230.365252`.

[Ellis 2010] Paul D. Ellis (2010). *The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results.* Cambridge, UK: Cambridge University Press.

[Fang, Huang, T. Tang, and Z. Wang 2020] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang (May 2020). “Parallel Programming Models for Heterogeneous Many-Cores : A Comprehensive Survey”. In: *CCF Transactions on High Performance Computing*, pp. 382–400. DOI: `10.48550/arXiv.2005.04094`.

[Heiser 2020] Gernot Heiser (May 25, 2020). *The seL4 Microkernel – An Introduction.* seL4 Foundation Whitepaper. Version Revision 1.0. URL: `https://trustworthy.systems/publications/papers/Heiser_20:sel4wp.abstract.pml`.

[Heiser 2024] Gernot Heiser (Apr. 2024). “Lions OS: Secure – Fast – Adaptable”. In: Linux Australia. Gladstone, QLD, AU.

[Heiser, Velickovic, Chubb, Joshy, Ganesh, Nguyen, C. Li, Darville, Zhu, Archer, Zhou, K. Winter, Parker, Duchniewicz, and Bai 2025] Gernot Heiser, Ivan Velickovic, Peter Chubb, Alwin Joshy, Anuraag Ganesh, Bill Nguyen, Cheng Li, Courtney Darville, Guangtao Zhu, James Archer, Jingyao Zhou, Krishnan Winter, Lucy Parker, Szymon Duchniewicz, and Tianyi Bai (Jan. 2025). “Fast, Secure, Adaptable: LionsOS Design, Implementation and Performance". In: arXiv: 2501.06234 [cs.OS]. URL: https://arxiv.org/abs/2501.06234.

[Intel Corporation 2023] Intel Corporation (Mar. 2023). *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 2B: Instruction Set Reference, M-U.* Version 80. Intel Corporation. Santa Clara, CA, US. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2b-manual.html.

[Isci, Buyuktosunoglu, Cher, Bose, and Martonosi 2006] Canturk Isci, Alper Buyuktosunoglu, Chen-yong Cher, Pradip Bose, and Margaret Martonosi (2006). "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 347–358. DOI: 10.1109/MICRO.2006.8.

[Al-Khalili 2015] Asim Al-Khalili (Oct. 2015). *Power Consumption in CMOS.* Lecture slides for COEN 451. Week 7 Slides. Montreal, Quebec, Canada. URL: https://users.encs.concordia.ca/~asim/COEN%20451/Lectures/W_7/W7_Slides.pdf.

[Klein, Andronick, Elphinstone, Murray, Sewell, Kolanski, and Heiser 2014] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser (Feb. 2014). "Comprehensive Formal Verification of an OS Microkernel". In: 32.1. NICTA
UNSW, 2:1–2:70. DOI: 10.1145/2560537. URL: https://trustworthy.systems/publications/nicta_full_text/7371.pdf.

[Krzywda, Ali-Eldin, Carlson, Östberg, and Elmroth 2017] Jakub Krzywda, Ahmed Ali-Eldin, Trevor Carlson, P-O Östberg, and Erik Elmroth (Nov. 2017). "Power-Performance Tradeoffs in Data Center Servers: DVFS, CPU pinning, Horizontal, and Vertical Scaling". In: *Future Generation Computer Systems* 81. DOI: 10.1016/j.future.2017.10.044.

[Le Sueur 2011] Etienne Le Sueur (June 2011). "An Analysis of the Effectiveness of Energy Management on Modern Computer Processors".
MSc Thesis. URL: https://trustworthy.systems/publications/papers/LeSueur%3Amsc.pdf.

[Le Sueur and Heiser 2010] Etienne Le Sueur and Gernot Heiser (Oct. 2010). "Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns". In: UNSW. Vancouver, Canada, pp. 1–5. URL: https://trustworthy.systems/publications/nicta_full_text/4158.pdf.

[Le Sueur and Heiser 2011] Etienne Le Sueur and Gernot Heiser (June 2011). "Slow Down or Sleep, That is the Question". In: UNSW. Portland, Oregon, USA. URL: https://trustworthy.systems/publications/nicta_full_text/4667.pdf.

[Linus Torvalds and Linux Kernel Developers 2024] Linus Torvalds and Linux Kernel Developers (Nov. 2024). *cpuidle.c.* https://github.com/torvalds/linux/blob/master/drivers/cpuidle/cpuidle.c. Accessed: 2025-04-22.

[Liu and X. Li 2024] Peng Liu and Xiaochun Li (Apr. 2024). *Using Processor Idle C-States with Linux on ThinkSystem Servers.* Lenovo Press. URL: `https://lenovopress.lenovo.com/lp1945-using-processor-idle-c-states-with-linux-on-thinksystem-servers` (visited on 04/22/2025).

[Lorenzon, Cera, and Beck 2015] Arthur F. Lorenzon, Márcia C. Cera, and Antonio Carlos S. Beck (2015). "On the influence of static power consumption in multicore embedded systems". In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1374–1377. DOI: `10.1109/ISCAS.2015.7168898`.

[Mi, D. Li, Yang, X. Wang, and Chen 2019] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen (Mar. 2019). "SkyBridge: Fast and Secure Inter-Process Communication for Microkernels". In: pp. 1–15. ISBN: 9781450362818. DOI: `10.1145/3302424.3303946`.

[Microsoft Corporation 2017] Microsoft Corporation (Jan. 2017). *Processor Power Management in Windows 7 and Windows Server 2008 R2.* White paper. Originally published 15 January 2010. Microsoft. URL: `https://learn.microsoft.com/en-us/previous-versions/windows/hardware/design/dn613983(v=vs.85)`.

[Microsoft Corporation 2024] Microsoft Corporation (2024). *Processor power management options overview.* Describes Windows 10+ processor power management algorithms and configuration, including core parking, power profiles, and QoS. URL: `https://learn.microsoft.com/en-us/windows-hardware/customize/power-settings/configure-processor-power-management-options`.

[Mwaikambo, Raj, Russell, Schopp, and Vaddagiri 2004] Zwane Mwaikambo, Ashok Raj, Rusty Russell, Joel Schopp, and Srivatsa Vaddagiri (July 2004). "Linux Kernel Hotplug CPU Support". In: *Proceedings of the Ottawa Linux Symposium.* Vol. 2. Ottawa, Canada, pp. 466–480. URL: `https://kernel.org/doc/ols/2004/ols2004v2-pages-181-194.pdf` (visited on 04/23/2025).

[Pallipadi, S. Li, and Belay 2007] Venkatesh Pallipadi, Shaohua Li, and Adam Belay (June 2007). "cpuidle – Do nothing, efficiently..." In: *Proceedings of the Linux Symposium.* Vol. 2. Intel Open Source Technology Center. Ottawa, Ontario, Canada, pp. 119–126. URL: `https://landley.net/kdocs/ols/2007/ols2007v2-pages-119-126.pdf`.

[Palmur, Z. Li, and Zadok 2013] Madhu Palmur, Zhichao Li, and Erez Zadok (Jan. 2013). *CPUidle from User Space.* Tech. rep. FSL-13-01. Stony Brook, NY, US: File Systems and Storage Lab, Department of Computer Science, Stony Brook University. URL: `https://www.fsl.cs.sunysb.edu/docs/cpuidle/cpuidle-from-userspace.pdf`.

[Park, Rho, Kim, and Nam 2019] Geunchul Park, Seungwoo Rho, Jik-Soo Kim, and Dukyun Nam (Mar. 2019). "Towards optimal scheduling policy for heterogeneous memory architecture in many-core system". In: *Cluster Computing* 22. DOI: `10.1007/s10586-018-2825-4`.

[RISC-V Platform Specification Task Group 2022] RISC-V Platform Specification Task Group (Mar. 2022). *RISC-V Supervisor Binary Interface Specification.* Version 1.0.0.

RISC-V Platform Specification Task Group. Stanford, CA, US. URL: https://www.scs.stanford.edu/~zyedidia/docs/riscv/riscv-sbi.pdf.

[Roy, Mukhopadhyay, and Mahmoodi-Meimand 2003] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand (2003). "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits". In: *Proceedings of the IEEE* 91.2, pp. 305–327. DOI: 10.1109/JPROC.2002.808156.

[seL4 Foundation 2025] seL4 Foundation (2025). *MCS. seL4 docs.* URL: https://docs.sel4.systems/Tutorials/mcs.html#scheduler (visited on 02/15/2025).

[Shah and W. Li 2023] Maulik Shah and Wing Li (2023). *PSCI OS Initiated Mode: Design Guide.* Version 2.9.0. Trusted Firmware. URL: https://trustedfirmware-a.readthedocs.io/en/latest/design_documents/psci_osi_mode.html.

[Sharkey, Buyuktosunoglu, and Bose 2007] Joseph Sharkey, Alper Buyuktosunoglu, and Pradip Bose (2007). "Evaluating design tradeoffs in on-chip power management for CMPs". In: *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, pp. 44–49. DOI: 10.1145/1283780.1283791.

[Singh 2018] Deepak Singh (Feb. 2018). "DynamIQ vs big.LITTLE Architecture: What has Changed". In: *Smartprix Bytes.* Online technology blog. URL: https://www.smartprix.com/bytes/dynamiq-vs-big-little-architecture-changed.

[Snowdon, Le Sueur, Petters, and Heiser 2009] Dave Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser (Apr. 2009). "Koala: A Platform for OS-Level Power Management". In: ed. by John Wilkes. NICTA
UNSW
Open Kernel Labs. Nuremberg, DE, pp. 289–302. URL: https://trustworthy.systems/publications/nicta_full_text/1528.pdf.

[Sudheendra and Prabhakar 2020] Raghuvarya Sudheendra and Deepika Prabhakar (June 2020). "A Review on Power State Coordination in Application Processors using ARM Specified PSCI". In: *International Research Journal of Engineering and Technology (IRJET)* 7.6. RV College of Engineering, Bengaluru, pp. 2537–2540. URL: https://www.irjet.net/archives/V7/i6/IRJET-V7I6478.pdf.

[Takouna, Dawoud, and Meinel 2011] Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel (2011). "Accurate Mutlicore Processor Power Models for Power-Aware Resource Management". In: *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pp. 419–426. DOI: 10.1109/DASC.2011.85.

[D. Tang 2021] David Tang (Apr. 5, 2021). *Tuning Servers for Energy Savings.* Microsoft Sustainable Software blog post describing processor power management, idle states, and core parking in Windows Server. Microsoft. URL: https://devblogs.microsoft.com/sustainable-software/tuning-servers-for-energy-savings/.

[Texas Instruments 2024] Texas Instruments (2024). *INA226 36V, 16-Bit, Ultra-Precise I2C Output Current, Voltage, and Power Monitor With Alert.* Datasheet. Texas Instruments. URL: https://www.ti.com/lit/ds/symlink/ina226.pdf.

[The Business Research Company 2025] The Business Research Company (Jan. 2025). *Hybrid Memory Cube (HMC) Global Market Report 2025.* Tech. rep. Market analysis report. London, UK: The Business Research Company. URL: https://www.thebusinessresearchcompany.com/report/hybrid-memory-cube-hmc-global-market-report.

[The kernel development community 2025] The kernel development community (2025). *Energy Aware Scheduling.* Linux kernel documentation. URL: https://docs.kernel.org/scheduler/sched-energy.html (visited on 11/22/2025).

[The Linux Foundation 2024] The Linux Foundation (2024). *CPU and Device Power Management. Kernel Driver API Documentation.* The Linux Foundation. San Francisco, CA, US. URL: https://www.kernel.org/doc/html/latest/driver-api/pm.

[The Linux Kernel Developers 2010] The Linux Kernel Developers (2010). *Supporting multiple CPU idle levels in kernel `cpuidle` governors.* Overview of the `cpuidle` governor callbacks and policy role. The Linux Kernel. URL: https://www.kernel.org/doc/Documentation/cpuidle/governor.txt (visited on 11/29/2025).

[UEFI Forum 2024] UEFI Forum (Nov. 2024). *Advanced Configuration and Power Interface (ACPI) Specification.* Version 6.5A. UEFI Forum, Inc. Beaverton, OR, US. URL: https://uefi.org/sites/default/files/resources/ACPI_Spec_6.5a_Final.pdf.

[Velickovic 2025] Ivan Velickovic (Apr. 2025). *Microkit User Manual.* Version 0.1.0. Part of the seL4 ecosystem. Trustworthy Systems Group, University of New South Wales. Sydney, Australia. URL: https://github.com/seL4/microkit/blob/main/docs/manual.md.

[J. A. Winter, Albonesi, and Shoemaker 2010] Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker (2010). "Scalable thread scheduling and global power management for heterogeneous many-core architectures". In: *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 29–39.

[Wysocki 2018] Rafael J. Wysocki (2018). *CPU Idle Time Management.* The Linux Kernel Documentation. Linux kernel admin guide for the CPUIdle subsystem. URL: https://docs.kernel.org/admin-guide/pm/cpuidle.html (visited on 11/29/2025).

[Wysocki 2020] Rafael J. Wysocki (Nov. 2020). *CPU Idle Time Management Driver.* Linux Kernel Documentation. Part of the Linux kernel power management documentation. Santa Clara, CA, US: Intel Corporation. URL: https://docs.kernel.org/admin-guide/pm/intel_idle.html.