Karlsruhe Institute of Technology

System Suspend with Asynchronous Resume using CXL-Based Persistent Memory

Bachelor’s Thesis
submitted by

Felix Zimmer

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Yussuf Khalil, M.Sc.

June 5, 2024 – October 7, 2024

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

Karlsruhe, October 7, 2024

Abstract

Optimizing power consumption of computer systems has been a key objective in operating systems-related research for many years. Toward this goal, suspending a system to a sleep state while it is not in use has proven invaluable. Regarding sleep states, a trend away from ACPI-defined modes, such as S3, and towards operating system-directed sleep can be observed. While this development improves resume latency, it sacrifices on energy conservation, as the system cannot be powered off entirely when employing these modes. In this thesis, we present a novel approach to system suspend, Suspend-to-CXL, using CXL-based persistent memory to store the context of the system, allowing the machine to be powered off completely, thus consuming little energy during sleep. In order to ensure rapid recovery of the system after sleep, we propose an asynchronous resume strategy, which initially only restores context necessary for the operating system to resume and returns state of user applications to system memory only when the system is responsive again. We implement our design on physical hardware, employing a CXL-capable FPGA to mimic persistent CXL storage. Our evaluation reveals that Suspend-to-CXL exhibits constant resume times of approximately 11 s, regardless of system memory usage. Moreover, our implementation is able to undercut hibernation on Linux in terms of energy consumption by as much as 5.5 times on suspend and 6 times on resume.

Contents

Abstract **v**

Contents **1**

1 Introduction **3**

2 Background **7**

2.1 Firmware 7

2.1.1 UEFI, ACPI, and SMBIOS 8

2.1.2 SMM 9

2.2 coreboot 10

2.2.1 coreboot Architecture 11

2.2.2 Hardware Specialization 12

2.2.3 CBMEM 13

2.3 System Suspend 13

2.3.1 Suspend-to-Idle 14

2.3.2 ACPI Sleep States 14

2.3.3 Linux Hibernation 16

2.4 Compute Express Link 17

2.4.1 CXL-based Memory Expander 18

2.4.2 NUMA 19

2.5 Linux Memory Management 20

2.6 Related Work 21

2.6.1 Suspend-to-PMem 23

3 Design **25**

3.1 Requirements 26

3.2 Suspend 27

3.2.1 Linux 27

3.2.2 coreboot 29

3.3 Resume 31

3.3.1 coreboot 31
3.3.2 Linux 33

4 Implementation 37
4.1 Using CXL memory 37
4.2 Cross-component Communication 38
4.3 Linux Kernel 39
4.3.1 Memory Management 40
4.3.2 Suspend 41
4.3.3 Asynchronous Resume 41
4.4 coreboot 42
4.4.1 Memory Management 43
4.4.2 Copying Memory 44
4.4.3 System Reset 45
4.5 Development and Debugging 46
4.6 Deploying coreboot 46
4.6.1 64-bit coreboot 47
4.6.2 Debugging 48

5 Evaluation 49
5.1 Methods 49
5.1.1 Hibernation 51
5.1.2 Benchmarking 52
5.1.3 Measuring Power Consumption 53
5.1.4 Measuring Time 53
5.2 Resume Latency 54
5.2.1 Suspend-to-CXL 55
5.2.2 Hibernation 58
5.3 Suspend Latency 62
5.4 Reliability 64
5.5 Energy Consumption 66
5.5.1 Asynchronous Resume 68
5.6 Discussion 69

6 Future Work 73

7 Conclusion 77

Bibliography 79

Chapter 1

Introduction

Optimizing energy consumption is an important topic in computer science, for consumers and large businesses alike. For example, consumers may prioritize the battery life of their laptops, while large businesses focus on reducing energy costs in data centers. A crucial component for optimizing energy consumption is saving on energy while the computer is not in use. For achieving this goal, computers can temporarily transition to different low-power states, also called sleep states [64, Chapter 16], in which power draw is greatly reduced.

These sleep modes vary in the way they save the current state of the system as well as the low-power states connected devices are able to reach. Simultaneously, it is equally important for devices to be able to wake from sleep in a timely manner, so they can be used when they are needed. For this reason, a balance must be found between the energy saved while sleeping and the wake latency until the machine is able to resume normal operation. Finding this balance has proven to be difficult, as reducing the energy consumption during sleep often penalizes wake latency. Ultimately, the preferable power state depends heavily on the use case of the system. For instance, in laptops, enduring a brief delay upon waking up in the morning might be acceptable if it ensures sufficient battery life for the commute to work. Sleep states are formally specified by the Advanced Configuration and Power Interface (ACPI) [64, Chapter 16]. For this thesis, three sleep states are relevant: S3, S4, and Suspend-to-Idle.

In S3, also called "Suspend-to-RAM," all devices not actively maintaining memory are turned off, the processor halts execution, and system memory is left in a low-power, self-refreshing state. Thus, S3 ensures fast wake-ups, as all system state is still present in memory, and setting the system back up can be done quickly, only requiring minimal initialization. Also, significant amounts of energy are conserved, as only system Dynamic Random Access Memory (DRAM) is actively powered. Regardless, self-refreshing RAM continuously consumes power during sleep, resulting in non-negligible energy consumption.

More energy can be saved in the S4 power state, also called “hibernate,” “hibernation,” or “Suspend-to-Disk.” When entering S4, the entire system state is persistently saved to disk. S4 thus permits a complete power-down of the machine including RAM, resulting in essentially no power consumption during sleep. The drawback to this approach is a much larger latency on wake-up compared to S3. For resuming from S4, the system has to effectively complete a full boot sequence, including platform initialization and discovery and setup of devices in firmware. Firmware is low-level software that is tasked with setting up the machine, including system memory and connected devices, to a point where the operating system is able to take over. At the end of this sequence, memory context is restored by reading the data from disk and restoring it to memory.

Suspend-to-Idle, called S0Ix by Intel and S2Idle [67] by the Linux kernel, is a pure software implementation of sleep which is fully implemented in the operating system, requiring no firmware support. Nonetheless, depending on the system, S0Ix can provide substantial energy savings, in some cases even close to those of proper ACPI sleep states, such as S3, while simultaneously ensuring near-instant wake latencies [26]. Moreover, since Suspend-to-Idle operates without the need for additional firmware assistance, it is compatible with a wide range of devices.

In this thesis, we present Suspend-to-CXL (S2CXL), an implementation of system suspend to persistent memory. We use byte-addressable Compute Express Link (CXL) memory to store operating system and user data during sleep, subsequently restoring it on resume. In theory, this allows the machine to be powered off completely during sleep, consuming energy comparable to S4. Additionally, we promise low resume latencies by only copying necessary kernel data back to DRAM on resume and initially mapping user applications to memory on the CXL device. We then perform a priority analysis on user applications and asynchronously transfer their data back from CXL memory to system memory based on these priorities. Our design contains components in firmware and the operating system. For our implementation, we employ the open-source firmware coreboot [45] and the open-source operating system family Linux [61], which both allow the modification of their source code, thus enabling us to adapt functionality as needed. Using firmware for the initial restore process enables us to leverage existing mechanisms for jumping back to the operating system from suspend, avoiding the need for complicated, low-level bootstrap code for Linux to restore itself. Furthermore, using firmware for restoring the kernel to memory allows us to skip large parts of the boot process, thus ensuring short resume durations. On the other hand, implementing the asynchronous resume component in Linux provides the benefit of being able to utilize existing functionality in Linux to identify memory ranges used by the kernel and user processes, which need to be saved.

In the following chapters, we describe Suspend-to-CXL in great detail. We begin by providing necessary background information and touch on related work in Chapter 2. We continue by presenting the theoretical design and practical implementation of Suspend-to-CXL in Chapter 3 and Chapter 4, respectively. Next, we evaluate our proposed sleep mechanism on real-world, commodity hardware and discuss our findings in Chapter 5. We compare Suspend-to-CXL to hibernation on Linux in terms of resume latency, suspend latency, power consumption and reliability. Our analysis demonstrates that Suspend-to-CXL is able to reduce resume and suspend durations by at least $78\,\%$ and $58\,\%$, respectively, compared to hibernate. Moreover, the implementation of our proposed sleep mode is able to successfully resume in at least $97\,\%$ of cases, thus proving reliable. Lastly, we propose further improvements and opportunities for future work in Chapter 6 and finally conclude this thesis in Chapter 7.

Chapter 2

Background

In this chapter, we cover important concepts for the rest of this thesis. We start by giving a short introduction to system firmware, focusing on the open-source firmware coreboot, which was used to implement the sleep mode proposed in this thesis. Additionally, we present details on the Advanced Configuration and Power Interface and the therein defined sleep states, their properties, and performance characteristics. We continue by providing some information on Compute Express Link, a high-speed interconnect between CPU and peripheral devices, which we use to implement our suspend functionality. Lastly, we explain important aspects of memory allocation on Linux, before ending this chapter by touching on some related work concerned with suspending to persistent memory.

2.1 Firmware

System firmware is software that has fine-grained control over the hardware of the machine. In contrast to the operating system (OS) on most computers, firmware is typically stored in some form of read-only memory, e.g. *electrically erasable programmable read-only memory* (EEPROM), requiring external tools and software like `flashrom` [8] to modify its contents. Firmware is responsible for setting up the machine to enable the OS to use abstractions from the hardware, similarly to how user applications leverage abstractions from hardware provided by the operating system. Thus, the primary task of firmware is to initialize the machine and all its devices to a predefined state and provide the operating system with vital information on the hardware of the system. Additionally, firmware can provide runtime services for power management, device configuration, and general system information [41, Chapter 1, Page 7]. For example, an essential piece of information that an operating system needs for correct setup and which we use for our implementation, is a map of physical RAM (random access memory) in the system. This information cannot be obtained by the OS directly, as the detection mechanism is highly platform-dependent and must be performed before system RAM is usable [29]. Since the firmware initially runs on read-only memory and does not require system memory to be available, in contrast to conventional operating systems, firmware is tasked with detecting memory. Moreover, the firmware might reserve parts of physical RAM for its own data structures or runtime functionality such as System Management Mode as described in Section 2.1.2, which the OS must respect. The standard way for the operating system to detect the memory map on x86 is by leveraging a Basic Input/Output System (BIOS) function termed the "E820 call" [64, Chapter 15]. This call to firmware functionality is performed by storing the value `0xE820` in the EAX register and issuing a `0x15` software interrupt. The BIOS then fills a given buffer with entries of the memory map on subsequent calls, containing a base address, the length of the region, and its type, describing how each memory region is used and whether the operating system may access it.

```
BIOS-e820: [mem 0x0000000000000000-0x000000000009efff] usable
BIOS-e820: [mem 0x000000000009f000-0x00000000000bffff] reserved
BIOS-e820: [mem 0x0000000000100000-0x0000000009afffff] usable
BIOS-e820: [mem 0x0000000009b00000-0x0000000009dfffff] reserved
BIOS-e820: [mem 0x0000000009e00000-0x0000000009efffff] usable
BIOS-e820: [mem 0x0000000009f00000-0x0000000009f3bfff] ACPI NVS
BIOS-e820: [mem 0x0000000009f3c000-0x0000000049b5ffff] usable
BIOS-e820: [mem 0x0000000049b60000-0x000000005a77efff] reserved
BIOS-e820: [mem 0x000000005a77f000-0x000000005af7efff] ACPI NVS
BIOS-e820: [mem 0x000000005af7f000-0x000000005affefff] ACPI data
BIOS-e820: [mem 0x000000005afff000-0x000000005affffff] usable
BIOS-e820: [mem 0x000000005b000000-0x00000000ffffffff] reserved
BIOS-e820: [mem 0x0000000100000000-0x000000047fffffff] usable
```

Listing 2.1: Example results of an E820 call, logged to the Linux kernel ring buffer and obtained via `dmesg`. Each line describes a single contiguous memory region with its associated type.

2.1.1 UEFI, ACPI, and SMBIOS

Modern computers use the *Unified Extensible Firmware Interface* (UEFI) [65] and the *Advanced Configuration and Power Interface* (ACPI) [64] as both a specification for firmware functionality and an interface between firmware and the operating system. To share data between firmware and operating system, the ACPI specification defines nested data structures called ACPI Tables [64, Chapter 21]. These tables contain entries describing "devices on the system board or devices that cannot be detected or power managed using some other hardware standard, plus their capabilities." [64, Chapter 5] Additionally, they also list other system capabilities such as "the sleeping power states supported, a description of the power planes and clock sources available in the system, batteries, system indicator lights, and so on." [64, Chapter 5] This allows the operating system to utilize these capabilities and functions without requiring detailed knowledge of the specific hardware control mechanisms. When initializing the system, the firmware populates these tables with *ACPI Machine Language* (AML) code, which the operating system then parses to create a database of all devices and their supported functions. The OS then can interpret the AML code to execute the provided functions [64, Chapter 20]. For example, if the operating system wishes to power off the machine, the corresponding AML code would be called, containing implementation details on how to power off the specific hardware found in the system. Furthermore, most systems share hardware information like serial numbers and memory slot population with the operating system via *System Management BIOS* [52] (SMBIOS). Similarly to ACPI, SMBIOS defines a table format containing this information, which can be read by the OS.

2.1.2 SMM

An x86 CPU can operate in different modes, which determine the availability of specific parts of the instruction set. One of these modes, *System Management Mode* (SMM) [20], is discussed in detail in the following section. SMM is entered by creating an interrupt called *System Management Interrupt* (SMI). Creating an SMI can either be performed by issuing a hardware interrupt on the designated *SMI#* pin of the processor or by triggering a software interrupt by writing to a specific I/O port. After receiving an SMI, the processor saves context and performs a jump to firmware SMM code. In addition to the mechanisms described in Section 2.1.1, the x86 architecture thus provides SMM as an additional means for firmware code to run after control of the system has been handed over to the OS. The operations then performed in SMM are completely transparent to the operating system.

As a result of firmware having complete control over the system, SMM is a popular target for attackers looking for a way to execute code without knowledge of the OS [7]. For this reason, SMM code is executed in a separate address space called *system management RAM* (SMRAM), which is not accessible to the operating system or user programs. When it was introduced, SMM was used to provide an operating system "with a transparent mechanism for implementing platform-specific functions such as power management and system security." [20, Volume 1, Chapter 3.1] If the OS signals that it is ACPI compliant, the firmware will refrain from performing these low-level functions and instead only update ACPI table structures if needed and return from SMM after an SMI occurred. However, as essentially every x86 machine built in recent years is ACPI compliant, responsibility for low-level control of the hardware has increasingly shifted to the operating system, making SMM less important. Despite its decreasing importance, SMM continues to be supported on all x86 and x86_64 processors and still contains code for device management on most platforms, as the firmware cannot assume that the operating system it will load is ACPI compliant and capable of completing generic low-level functions.

2.2 coreboot

Traditionally, firmware implementations for desktop and server platforms are proprietary, meaning their code is private, and therefore, their behavior cannot be easily adapted. Since firmware tasks are highly dependent system hardware, most original equipment manufacturers (OEMs) often only implement needed functionality for the specific platform and then rarely update the firmware. This development style results in fragmented source trees and a "throw-away" mentality, where older platforms rarely receive updates unless addressing critical security issues [2, Chapter 1, Page 3]. In contrast, *coreboot* [45] is an open-source firmware implementation focusing on speed and simplicity. The project attempts to "do the bare minimum necessary to ensure that hardware is usable and then pass control to a different program called the payload." [47] Therefore, coreboot provides no user interface at runtime and is solely concerned with platform and device initialization, handing over control to the payload binary as early as possible to implement further functionality. As is common for firmware, the majority of the source code is written in C or architecture-specific assembly, with some exceptions like x86 graphics code written in ADA SPARK. coreboot's boot flow is divided into different stages, each responsible for a different part of hardware initialization, and is described further in Section 2.2.1.

This design choice ensures a flexible boot sequence, enabling the use of an adequate payload for varying situation. Popular payloads include Intel's reference implementation of UEFI called *TianoCore EFI Development Kit II* (EDK2) [18] providing UEFI runtime services, an EFI shell and user interface, *SeaBIOS* [39] for legacy BIOS systems, and *LinuxBoot* [62], which attempts to use the Linux kernel to complete common UEFI functionality. coreboot is primarily compatible with x86 and x86_64 CPUs from AMD and Intel but also supports ARM and RISC-V CPUs. The project uses a single source tree for all platforms and architectures, guaranteeing that older platforms benefit from newer code changes. Although the coreboot code is entirely open source, certain architectures, particularly x86 and x86_64, require additional proprietary BLOBs (binary large objects) to fully initialize hardware such as the CPU and RAM. For this reason, coreboot allows for the integration of these binaries and invokes their functions when required. Intel provides a *Firmware Support Package* (FSP) [14], which fills this role. coreboot recently introduced stable 64-bit support [46], which is essential for our implementation, as we have to be able to address memory over 4 GiB, which is not possible in 32-bit mode.

2.2.1 coreboot Architecture

coreboot's boot process is divided into multiple, separately compiled stages [49]. These stages have different responsibilities and after one stage has finished, it loads and runs the next stage. As a result, each stage has access to a different set of library functions and data structures. An overview of coreboot's stages is given in Figure 2.1.

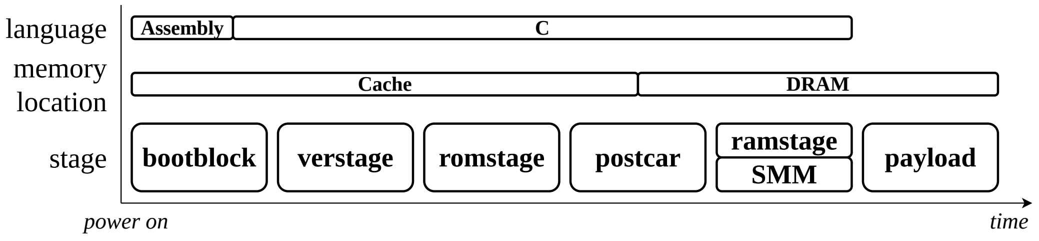

Figure 2.1: Simplified overview of coreboot's boot stages on x86, including the source code languages and the memory locations used in each stage.

The first stage to run is the *bootblock*, which is written mostly in assembly and is tasked with setting up an environment for the C code of following stages to run. C relies on byte-addressable memory for its stack and heap; however, at the initial stages of system startup, system memory is not yet initialized. For this reason, coreboot uses CAR (cache-as-RAM) in its early stages, allowing the CPU cache to be used as system memory, thus enabling C code to run. On x86 systems, the bootblock is additionally responsible for updating microcode, setting up hardware timers and switching from 16-bit real-mode to 32-bit protected mode.

After the bootblock finishes, it may load the optional *verstage*. This stage is used to establish a "root-of-trust" [49] for verified boot, a security mechanism to ensure firmware integrity. Next, the *romstage* is run. The romstage primarily sets up the system's DRAM for the following stages to use. On x86, the very short *postcar* stage then disables CAR and loads the *ramstage* to run from regular system memory.

The ramstage accomplishes the majority of actual device initialization. It enumerates and initializes *Peripheral Component Interconnect* (PCI), *Peripheral Component Interconnect Express* (PCIe), and on-chip devices and sets up graphics output if supported. The ramstage also configures SMM, as discussed in Section 2.1.2, to handle SMIs, by establishing memory regions, such as CPU stacks, necessary for executing SMM code. Lastly, information on the system is written to the ACPI and SMBIOS tables for the OS to read and process. Additionally, coreboot maintains its own tables called coreboot tables, in which it saves information like the system memory map, log messages, and timestamps. After the ramstage has written its data to these tables, the payload binary is loaded and executed, at which point coreboot has finished initializing the system.

2.2.2 Hardware Specialization

As mentioned in Section 2.2, every platform uses the same coreboot source tree. Necessary modifications for varying hardware are performed in the subdirectories of the `src` directory of the coreboot repository [51]. For example, modifications to the mainboard, e.g., memory module slot population restrictions or available *general-purpose input/output* (GPIO) pins, are applied in the mainboard-specific subdirectories of `src/mainboard/`. Similarly, CPU-architecture specific code is placed in `src/arch/`, including assembly implementations for low-level functionality, which inherently differs between architectures. Similar to the Linux kernel, coreboot makes use of Kconfig [59], a mechanism based on configuration files for selecting specific high-level options such as the target mainbard, PCI support, 64-bit execution, or serial port logging, for example. Based on the selected values, a `.config` file in the project root is generated, containing the project-wide configuration state, based on which the firmware binary is built. To then create the binary, a collection of GNU make [42] files is employed to decide which source files are compiled, based on the variables set in the `.config` file. To enable modules belonging to specific hardware to modify the behavior of the firmware, coreboot defines abstract functions in header files and provides default implementations with weak linkage, which can be overridden. For example, we require some dynamic memory management for our suspend and resume logic (see Section 4.4.1 for more details). For this reason, we override the empty, weakly-linked function `bootmem_platform_add_ranges` [51] defined in `src/include/bootmem.h` to reserve a memory region for us to use, which the operating system recognizes as firmware-reserved and refrains from using.

Given that system firmware is highly hardware-dependent, coreboot requires critical hardware information about each mainboard, such as present devices, slots, and buses for devices that cannot be easily probed at runtime. To address this issue, coreboot interprets a mainboard-specific `devicetree.cb` file containing information on connected devices, including daughter boards, serial ports, and baseboard management controllers.

2.2.3 CBMEM

While coreboot adopts a minimalistic approach to firmware runtime services and strives to remove itself entirely from memory before transferring control to the payload, certain memory regions, such as those designated for setting up SMM, must remain in memory even after coreboot completes its initialization. Furthermore, the tables outlined in Section 2.1.1 must be retained as well for the OS to use. These tables are placed in a memory area called *CBMEM* [48]. Additional data, such as boot logs and timestamps may also be saved to CBMEM. After the system has booted, the data in this memory region can be read, but not written, from Linux using coreboot's `cbmem` program [51, `util/cbmem/`]. We use both the boot logs and timestamps to collect information on latencies and memory amounts for the evaluation of our implementation, as we describe in further detail in Section 4.5. On top of saving messages to CBMEM, coreboot also supports logging these messages in real time to on-board serial ports. We make use of this fact for debugging purposes as it does not require the system to successfully boot to Linux.

2.3 System Suspend

Energy consumption makes up a large chunk of the total cost of ownership of a computer. For optimizing energy consumption, it is of upmost importance to reduce the power consumption of a system while it is not in use. While placing a computer in a low-power state when it is not used is no novel idea, the approaches toward achieving this goal have changed over time.

Before the release of ACPI, as introduced in Section 2.1.1, power management relied on complicated, platform-dependent firmware code, which provided less flexibility in terms of usable policies than the *OS-directed configuration and power management* (OSPM) [64, Chapter 1] provided by ACPI. For managing the system and connected devices, ACPI defines power states and transitions between them, which the operating system can invoke, for example when the user requests the system to shut down. To enter a selected sleep state, OSPM has to follow specific steps defined by the ACPI specification [64]. This specification additionally defines different *sleep states* [64, Chapter 16.1], called S0 to S5, which all exhibit different power characteristics.

While powered on and executing instructions normally, an ACPI compliant computer is in sleep state S0, also called the *Working* state. On the other hand, a system in S5, also called *Soft Off*, is completely powered down and waiting for the power button to be pressed to begin a full boot sequence, consuming only minimal energy. S1 to S4 progressively power down more hardware and are described in greater detail in the following sections. In recent years, a trend toward putting the OS in charge of power management by using pure software solutions rather than involving the firmware with ACPI sleep states can be seen. Despite this fact, ACPI sleep states still remain relevant today.

2.3.1 Suspend-to-Idle

Suspend-to-Idle [67], also called S2Idle or S0Ix, is not a proper sleep state as defined by ACPI [64, Chapter 16]. As far as ACPI is concerned, a system in S0Ix is running normally, meaning it is in the S0 *Working* state and is able to execute instructions. Instead, Suspend-to-Idle is a pure software implementation of suspend. When in this state, the entire user space is frozen and all I/O devices are put into low-power states. As a result, the CPU stays in its lowest power idle state, thus consuming little energy. Since S0Ix is no proper, ACPI-defined sleep state, code execution can still occur, resulting in both more devices being able to wake the system and important kernel tasks theoretically being able to continue execution. A system in S0Ix can be woken up by interrupts from I/O devices like a keyboard and even from network events [67].

2.3.2 ACPI Sleep States

Entering an ACPI sleep state requires writing values to different hardware and software structures defined by ACPI [64, Chapter 16.1]. To control power management, ACPI defines the PM1 control register, `PM1_CNT` [64, Chapter 4.8.3.2], which is located in I/O space on x86 or mapped in physical memory space. The exact location of this register can be found by walking the ACPI tables, more specifically it is given in the *Fixed ACPI Description Table* (FADT). To enter an ACPI sleep state, OSPM first sets the 3-bit field `SLP_TYPx` in the `PM1_CNT` register to the target sleep state. Afterward, the `SLP_EN` bit in the same register is set, initiating the transition to the specified state.

Moderate energy savings can be achieved by employing the sleep state S1 [64, Chapter 16.1.1]. Before entering S1, user space is frozen and I/O-devices are placed in low-power states, similar to S2Idle. Additionally, CPU caches are flushed before sleep [64, Chapter 16]. Aside from these caches though, the system context, system memory in particular, is further maintained during sleep. An example implementation of S1 is "standby" on Linux [67] which leaves the CPU powered and disables all non-boot CPUs, allowing further energy savings.

S2 [64, Chapter 16.1.2] builds upon S1 but allows the entire CPU to be powered off. It only requires system memory to remain powered, while other devices can be put into low power states or turned off completely. When going to sleep, important context is saved to memory, including the wake vector, a memory address the OS writes before suspend where it has prepared code to resume itself. Analogous to S1, CPU caches are flushed before suspend, however the CPU and cache can then be completely powered off, which is not possible in S1. On waking up, the firmware must identify that the system was sleeping, restore context like the CPU cache, enable the previously disabled devices, and jump to the wake vector rather than running the normal OS entry code. S2 thus focuses on low wake-up latencies, while sacrificing on energy savings compared to S3 and S4.

Similarly to S2, S3 [64, Chapter 16.1.3] powers off external devices as well as the CPU while keeping RAM in a self-refreshing state. A system in S3 experiences a further reduction in energy consumption by permitting certain devices, such as the system power supply, to be placed in a lower power state, thus consuming even less energy. Until recently, S3 was a very popular sleep state to implement, though operating systems have started to encourage the usage of Suspend-to-Idle, as S0Ix allows the OS to control the entire sleep process [26]. In theory, S3 provides substantial power savings and quick resume times, but requires low-level platform support. Especially on x86, this platform code is often proprietary in nature, preventing third parties from providing S3 capabilities. This, paired with additional hardware requirements, hinders S3 from being available on every platform.

S4 [64, Chapter 16.1.4], also called Suspend-to-Disk or hibernation by the Linux kernel, is the lowest power sleep state that still maintains system context. When entering S4, system context is written persistently to disk. Subsequently, all devices can be powered down, resulting in a situation similar to when the machine is powered off completely, thus ensuing minimal power consumption during sleep. A core advantage of S4 is that it can be implemented without support from the underlying platform and needs no support from the firmware at all to wake correctly. On the other hand, resuming from S4 essentially requires an entire boot sequence after which the OS restores system context from the disk, resulting in the slowest resume of all sleep states. The transition to S4 can either be issued from the OS or from platform firmware, in which case the transition is called S4BIOS. To enter S4 via the S4BIOS mechanism, the OS generates an SMI with the value of `S4BIOS_REQ`, which can also be found in the *FADT*, stored in the `SMI_CMD` register [64, Chapter 5.2.9]. As S4BIOS is both very rarely used today and requires additional platform support, we focus on the entry to S4 from the operating system.

2.3.3 Linux Hibernation

To understand the performance comparison between our implementation and hibernation on Linux, we broadly sketch out the sequence of events after a hibernation call [67] in the following. To initiate the transition to S4, the low-level kernel `sysfs` [27] interface can be used directly by writing the string "disk" to `/sys/power/state`. Alternatively, the high-level interface via systemd [44] can be used, which performs various sanity checks and supports the execution of user-defined scripts before hibernation, after which the kernel interface is also used to initiate the transition. Afterward, all user processes are frozen to prevent applications from ending up in unexpected states on resume. Next, so called "nosave pages" are identified, which are pages that do not require saving in the hibernation image as they are either reserved by the platform or not mapped to DRAM at all, such as memory-mapped I/O (MMIO) regions of PCI devices [20, Volume 1, Chapter 19.3.1]. Linux allows configuring the target size of the hibernation image through the `sysfs` interface at `/sys/power/image_size` [67]. Values written to this file "will be used as a best-effort upper limit of the image size." [67] By default, the `image_size` is set to $2/5$ of the size of system memory. Tweaking this image size is important for our evaluation, which is elaborated further upon in Section 5.1. For OS-initiated S4 on Linux, the entire system context is compressed with a compression algorithm, for example LZ4 [4] or LZO [28], to create the hibernation snapshot and then copied persistently to a swap partition or swap file on disk. As the time needed for copying and compressing increases with the amount of data being processed, both suspend and resume latencies are significantly dependent on the size of the hibernation image. When writing is completed, the system may enter the low-power S4 state to permit waking up by means of keyboard or other device interrupts. By default, Linux deviates from the ACPI specification and simply shuts down the machine by entering S5, enabling further energy savings. Although putting the system in S5 saves more energy, it also prevents the machine from being able to wake up from interrupts generated by devices such as keyboards and laptop lids.

When the system is powered on again, firmware initializes the system as usual and starts a fresh instance of the Linux kernel. This kernel then searches the swap file for a hibernation image and loads it if present. Before loading it to memory, the kernel performs some sanity checks in `arch_hibernation_header_restore` [61, `arch/x86/power/hibernate.c`], one of which is ensuring that system firmware has initialized the memory map in the same way as when the hibernation call was made [61]. If these checks pass, the fresh kernel then overwrites itself with the loaded image and restarts user space, thus completing the suspend/resume cycle.

2.4 Compute Express Link

The rising need for storing and processing ever-growing volumes of information in data centers and cloud applications demands new memory interfaces without the limitations in bandwidth and scalability of traditional *Double Data Rate* (DDR) memory. Normally, computer systems fall back to block devices, most commonly SSDs, for storage when running out of DDR memory. Despite providing large storage capacities, SSDs introduce latencies multiple orders of magnitude greater than DRAM-based memory, severely reducing application performance. Furthermore, the growing interest in machine learning, data analytics, and cloud computing has increasingly transformed the computing landscape toward heterogeneous systems, employing various accelerators, GPUs, memory expanders, and FPGAs in addition to conventional CPUs for their calculations.

Compute Express Link (CXL) [5] attempts to fill this gap between DRAM-based DDR memory and SSDs by providing a standardized interface for accessing accelerator memory. CXL is an open standard interconnect for high-bandwidth, low-latency, and cache-coherent data transfer between the processor and aforementioned devices. After its initial release in 2019, the standard continues to be actively developed by a group composed of researchers, hardware vendors, and cloud providers. For its implementation, CXL builds upon PCIe point-to-point infrastructure for communication between devices.

CXL defines three protocols for communication between components [5, Chapter 3]:

- **CXL.io** supplies the fundamental operations of CXL on top of PCIe, such as device discovery, link initialization, interrupt controlling, and register I/O.
- **CXL.mem** enables the CPU to coherently access the memory attached to CXL devices with conventional load/store instructions.
- **CXL.cache** provides a way for devices to cache data from host memory while ensuring consistency.

While these protocols are employed by devices and the host, they are transparent to the applications using CXL. The CXL standard categorizes CXL-capable devices into three distinct categories [5, Chapter 2]:

- **Type 1** devices expose no memory to the host but might maintain a private cache for their operation. Such devices might employ complex atomic operations, which standard PCIe does not provide. The cache-coherent transactions provided by CXL.cache enable both flexibility in the choice of atomic operations and ordering used by the device when accessing host DRAM.

- **Type 2** devices, such as FPGAs and GPUs, contain own memory. They use all three CXL protocols to expose their memory to the host and access the host's memory transparently during their computations.
- **Type 3** devices such as passive memory expanders rarely carry out computing work themselves and rather expose memory for the CPU or other CXL devices to use, increasing the capacity and bandwidth of host memory. These CXL devices may use any memory type internally, enabling the host to access both persistent and non-persistent, byte-addressable memory with conventional load/store operations.

For this thesis, we require a coherent memory space to be maintained between the CPU and the device, enabling byte-granular memory access from both components. This memory space is detected in firmware and passed to the OS as a separate ACPI proximity domain. Once configured, there is no discernible difference from the user's perspective between accessing conventional system DDR memory and utilizing the memory of the CXL device, which is crucial to our implementation and discussed in greater detail in Chapter 3. Since CXL only defines the interface for accessing memory, the type of backing storage can be chosen freely, as long as it is able to implement CXL semantics. Although we desire persistent storage to be able to retain the system state even when the machine is not powered, these solutions are not yet commercially available, as described further in Section 2.4.1.

2.4.1 CXL-based Memory Expander

While CXL has only begun to be supported on newer computing platforms [19], hardware manufacturers have announced first products using CXL as an interface, such as the *CXL Memory Module - Hybrid* (CMM-H) by Samsung, which supplies persistent, byte-addressable storage [40] to the host system. In theory, this type of device is perfectly fit for our use case. Due to its byte-addressable nature, CMM-H enables storing data effortlessly via conventional load/store instructions while simultaneously saving said data persistently without requiring additional measures. Therefore, devices such as CMM-H, could enable a system to save its context and fully power off to save energy. Then, when powering back on at a later time, retrieve the data to restore the state of the system, thus implementing a type of system suspend as described in Section 2.3.

Unfortunately, none of these products are commercially available at the time of writing this thesis. As a result, we cannot evaluate our implementation on persistent and CXL-capable hardware. Instead, we use an FPGA-based memory expander that is capable of communicating via CXL. Internally, the FPGA is equipped with a single DDR4-3200 memory module to emulate persistent storage.

Although the performance characteristics of our FPGA will certainly differ from actual CXL hardware, our evaluation in Chapter 5 is nonetheless able to provide insights into performance and the characteristics of using CXL for system suspend. Furthermore, the perspective of the operating system and firmware on CXL memory, as described in the following section, remains unchanged when transitioning to commercial hardware such as CMM-H. This consistency ensures that switching to these devices will be trivial once they become available.

2.4.2 NUMA

The conventional paradigm of *Symmetric Multiprocessing* (SMP) [30, Page 509] describes a system topology containing multiple CPUs with identical access times to resources like memory, block storage, or networking. Although this view is easy to understand and sufficient for many situations, it is not accurate on modern systems. For example, a system might contain multiple CPU sockets with their own physical CPUs and memory controllers. In a system like this, a CPU experiences greater latencies when accessing memory belonging to the other memory controller than when accessing memory connected directly to its own memory controller. As a result, contemporary operating systems such as Linux consider such a machine to possess a *Non-Uniform Memory Access* (NUMA) architecture [37]. In this view, resources are assigned to NUMA nodes, which may or may not have one or more CPUs and some amount of memory and I/O buses. Each node associates the other nodes with a distance based on how large the bandwidth and latency is when accessing resources belonging to the other node. According to this model, conventional SMP systems can simply be treated as a system with a single NUMA node containing all CPUs and the entire system memory. NUMA promises to improve both performance and scalability of systems by optimizing resource usage. For example, a NUMA-aware operating system scheduler might decide to only schedule a compute-intensive task on the NUMA node with memory containing the task's data structures. By default, the CXL memory devices described in Section 2.4 appear to the operating system as a separate NUMA node with the memory capacity of the device and no CPUs. As such, their memory is mapped to the physical address space and can be used to satisfy allocations for both the OS and user applications.

To ensure availability of CXL memory at the time of suspend, some changes to the Linux kernel are necessary, which is elaborated on in further detail in Section 4.1. For this reason, a basic understanding of how the Linux kernel handles NUMA policies and allocations is essential. Internally, the kernel uses "memory policies" to decide which node to use for an allocation [12]. To allow for more flexibility in allocations, memory policies are assigned to a scope, enabling different treatment of tasks and use cases. These scopes include the system default scope, a per-process scope, a scope for a specific part of the virtual address space of a process, and a scope shared between multiple processes. These memory policies contain a "mode" describing the allocation strategy, the most important of which are described in the following.

- **MPOL_DEFAULT** is only used internally and by default for new processes and simply falls back to the next specific scope.
- **MPOL_BIND** exclusively allows allocations from the set of nodes specified in the policy.
- **MPOL_PREFERRED** and **MPOL_PREFERRED_MANY** specify a single or multiple preferred nodes respectively, from which allocations are attempted initially. If obtaining memory from the preferred node(s) fails, the other nodes are checked in increasing order of distance. This policy is used in the system default scope with the preferred node set as the node of the CPU the process is running on.
- **MPOL_INTERLEAVE** ensures that allocations are interleaved across nodes on a page granularity. To distribute memory usage evenly during booting, Linux uses this policy in the early boot phase before switching the mode of the system default scope to `MPOL_PREFERRED`.

User applications can request specific NUMA policies via system calls such as `set_mempolicy` [12]. Furthermore, kernel device drivers can also use provided functions to allocate from and even migrate already running processes to different NUMA nodes. Lastly, the command-line tool `numactl` [22] enables users to launch programs with specific NUMA policies, inspect memory capacity and usage of nodes present in the system and display the memory policy for the current process. Further details on the memory allocation interface on Linux are given in the following section.

2.5 Linux Memory Management

For our implementation, we make use of the memory management implementation of the Linux kernel to find the memory regions that must be persisted before powering off the system. Linux provides several APIs [57] for memory allocation, each designed for different use cases [60]. At the physical page-level, Linux uses a buddy allocator which maintains free-lists for multiple orders of pages. Memory can be allocated from the buddy allocator directly via `alloc_pages` [57]. Additionally, for smaller allocations, calls to `kmalloc` [57] can be made, which internally allocates pages from the buddy allocator if needed. For allocations larger than a single page that do not require pages to be physically contiguous, `vmalloc` [57] can be employed. Lastly, for use cases requiring repeated allocation and freeing of custom, similar-sized objects, the Linux kernel provides a slab cache allocator via `kmem_cache_create` and `kmem_cache_alloc`, which caches currently free blocks for fast reallocation [57]. Additionally, physical pages can also be mapped directly to the virtual address space of the kernel via `vmap` [57].

Since the buddy allocator requires some setup to complete allocations, it cannot be utilized immediately during the early boot stages of Linux. For this reason, Linux employs a different allocator during boot time called *memblock* [53]. Simply put, from memblock's point of view, system memory is a collection of contiguous regions which are either free or reserved. Allocations can be made by calling `memblock_alloc` or `memblock_phys_alloc`, which return a virtual or physical address, respectively. During the allocation of a region, its type is changed to reserved. At some point in the boot process, the memory allocated via memblock is freed to the buddy allocator [53]. As all allocations during normal operation are fundamentally provided by the buddy allocator, it is sufficient to traverse its data structures to find the parts of physical memory that are in use. Section 3.2.1 further describes how we capitalize on this fact to identify used memory regions that have to be saved on suspend and restored on resume for the implementation of Suspend-to-CXL.

2.6 Related Work

Previous work on optimizing system suspend in terms of latency and energy consumption has been conducted with varying approaches. We begin by giving an overview of relevant publications and conclude with a discussion of a closely related approach to our proposed sleep mode in Section 2.6.1.

Lo et al. [23] present swap-before-hibernate, a modification to conventional hibernate, employing the properties of flash storage devices such as SSDs. When a hibernation request is made, all swappable pages are moved to swap space on the flash device while static data, like code and file content, is simply discarded, reducing the amount of memory that has to be saved. As flash storage exhibits low seek times as well as fast reads, this introduces acceptable latencies. Swap-before-hibernate then bundles together non-swappable pages, such as kernel pages, and writes them to a hibernation file. On resume, Lo et al. copy back data from one of three sources: non-swappable memory is transferred back from the hibernation file, static data is read back from disk via demand paging, and swappable user data is read back from swap space via demand paging. Subsequent hibernation requests then complete even faster, as pages that have not changed since their last write to swap do not need to be copied again. Lo et al. manage to achieve a $5\times$ speedup in resume times compared to conventional hibernate on Linux.

To combat larger hibernation latencies due to increasing application memory footprints, Ho et al. [13] propose a classification scheme for prefetching memory pages for user applications. Pages are classified based on their access pattern into priorities and written to the hibernation image in this order. On resume, pages belonging to the Linux kernel are loaded first, allowing interactions with the system, like logging in and starting applications. Afterward, user pages are prefetched to memory in decreasing priority from the image, allowing urgent applications to access their pages quicker. In this sense, Ho et al. also implement the concept of an "asynchronous" resume. However, their approach does not profit from the flexibility offered by byte-addressable persistent storage, such as the capability to map regular memory pages directly, similar to system memory. Nonetheless, they show an improvement of up to $30.8\,\%$ in suspend and resume latencies compared to conventional hibernate. The sleep mechanism presented in this thesis implements a comparable strategy to the one described by the authors by prioritizing applications on resume, which is further described in Section 3.3.2.

Zi et al. [68] present a comparable approach to system suspend by saving the contents of system memory to Phase-change memory (PCM), a type of non-volatile memory capable of retaining data without continuous power supply. The authors implement their proposal, Suspend-to-PCM, entirely in software, utilizing the capabilities of the QEMU emulation platform. In this context, they emulate a PCM device, though they give no details on how this is accomplished. On a suspend request, the operating system completes the usual task of freezing user processes and subsequently transfers control to firmware. QEMU then saves the entire contents of DRAM to an emulated PCM device and powers the system off. On resume, the data saved in PCM is loaded back to DRAM, from where the operating system can resume. The authors evaluate their Suspend-to-PCM implementation by comparing it against Suspend-to-Disk and Suspend-to-RAM, conducting tests on both physical hardware and in the same virtualized environment used for Suspend-to-PCM. They find that both suspend and resume latencies are heavily dependent on system memory utilization. For instance, resume times are reported to be $36.7\,\mathrm{s}$ for $128\,\mathrm{MiB}$ of memory usage and $66\,\mathrm{s}$ for $1\,\mathrm{GiB}$. However, the authors do not explain the consistent observation that resume latency exceeds suspend latency, despite the fact that reading from most types of storage is typically faster than writing. Zi et al. compare the latency of Suspend-to-PCM to Suspend-to-Disk and Suspend-to-RAM as described earlier, claiming that resume latencies are comparable to those of Suspend-to-Disk on bare-metal, though they do not elaborate further at which memory usage level this comparison holds. Since no physical hardware was available for direct measurement, the authors calculate the theoretical energy consumption of their implementation by assuming values for the power draw of the system in idle, load and suspended states. However, they do not clarify how they derive these numbers. Notably, the power draw of 2.5 W Zi et al. assume the system exhibits when sleeping seems curious, as a system suspended to persistent memory should, theoretically, only consume negligible energy. How the results the authors present relate to an application on real hardware is not clear, as the authors do not reveal how they emulate the PCM device. Although this work leaves room for further investigation, we share the author's vision of suspending the system to non-volatile storage, nonetheless.

2.6.1 Suspend-to-PMem

In his thesis, Meyer [24] presents Suspend-to-PMem, a mechanism for suspending a system to Intel Optane [16] memory, a form of persistent memory. They implement this functionality entirely in firmware by copying the complete system memory to Optane on suspend via an FPGA connected to the system over PCIe. On resume, the firmware detects that the system was suspended via Suspend-to-PMem, reads memory back from Optane to host memory and subsequently jumps to the ACPI wake vector to resume. While the wake latency of Suspend-to-PMem is higher than that of normal S3, thus requiring more energy to resume due to necessary initialization of the FPGA, Meyer shows a larger reduction in energy consumption during sleep, due to the machine being able to completely power off. In their test setup, this results in a break-even point in energy consumption with S3 at a suspend duration of approximately one hour, past which their implementation provides substantial energy savings over S3.

Similar to our implementation, Suspend-to-PMem is implemented using coreboot [45] for copying memory in firmware. We build upon Meyer's idea of transferring memory in coreboot and expand on it by using CXL as our storage interface. Moreover, since Suspend-to-PMem is implemented entirely in firmware, the entire system memory is copied to Optane, which may contain memory ranges that are not used and thus copied in vain. We avoid this issue by incorporating a component into the Linux kernel that identifies utilized memory ranges and hands over this information to coreboot, as detailed in Section 3.2.1.

Chapter 3

Design

Building upon the fundamentals discussed in the last chapter, we present the design of Suspend-to-CXL (S2CXL), our proposal for a persistent system suspend. Our suspend mechanism borrows characteristics from S3, namely the straightforward access of byte-addressable memory and fast resume times, as well as from S4 in the form of persisting memory and powering down the system entirely during sleep and having to copy memory back on resume. We construct our design with components within the open-source firmware coreboot the Linux kernel. The high-level approach of our proposal is illustrated in Figure 3.1. Other sleep modes to persistent memory, such as hibernation, as described in Section 2.3.3, do not require firmware support. Instead, the conventional boot procedure is followed until the operating system registers that it is resuming from hibernation, at which point the previously saved system context is restored. In order to avoid the additional latency of completing an entire boot sequence, we opt to copy the data of the kernel back to system memory in firmware on resume. Furthermore, we use the ACPI wake vector as a means to jump back to the OS, similar to S3. We also employ firmware to save kernel memory on suspend, as opposed to copying memory directly from Linux, ensuring that no kernel data structures are changed at a later stage in the suspend process. For the above described reasons, we require components in Linux as well as in coreboot.

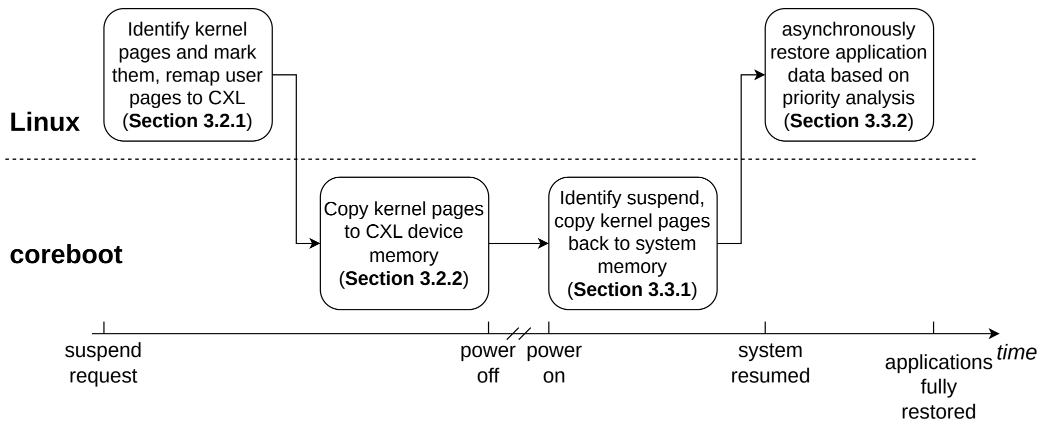
Linux
Identify kernel pages and mark them, remap user pages to CXL (Section 3.2.1)
asynchronously restore application data based on priority analysis (Section 3.3.2)
coreboot
Copy kernel pages to CXL device memory (Section 3.2.2)
Identify suspend, copy kernel pages back to system memory (Section 3.3.1)
suspend request
power off
power on
system resumed
applications fully restored
time

Figure 3.1: High level approach of Suspend-to-CXL. Applications are restarted after resume, initially accessing CXL memory until their data has asynchronously been transferred back to system memory.

3.1 Requirements

Our implementation is designed to be able to compete with existing mechanisms such as S3 and hibernate, which are characterized in further detail in Section 2.3.2 and Section 2.3.3, respectively. Toward this goal, we establish four key requirements, based on which we create and evaluate our implementation.

Resume latency

For users of a system, resume latency is the most noticeable metric, as it essentially measures the time in which a user wants to use the system, but is barred from doing so. As such, we focus on keeping resume latencies as close to Suspend-to-Idle and S3 as possible.

Suspend latency

Although arguably less important than resume latency, suspend latency nonetheless poses a metric worth optimizing for. Especially in settings where sleep times are small, a lower suspend latency can mean the difference between saving energy or issuing a suspend request, simply to power on the machine again immediately, thus wasting more energy than if the system had been left in an idle state.

Reliability

The fastest suspend mechanism is unusable, if it is not reliable. In fact, an unreliable suspend is worse than no suspend at all. For example, if a user expects their work to be present when they open their laptop lid in the morning, only to find all their progress lost due to a faulty suspend, this is a worse user experience than forcing the user to save their work, power off their computer and start it again when it is needed. We measure reliability as the ability of a system to achieve the target low power state and successfully recover to normal operation again.

Energy consumption

As the main objective of suspend is the system using less power than when remaining in idle, the energy savings that can be achieved with our implementation are of key interest. Specifically, we attempt to minimize energy consumption during both the suspend and resume processes by reducing the duration of these phases and ensuring that the system enters a low-power state to maintain minimal power draw while sleeping.

3.2 Suspend

Our suspend implementation is made up of a Linux component and a coreboot component. While we chose Linux to be able to directly modify kernel code, there is no reason why other operating systems should not also be able to suspend in the proposed way as long as they use the same interface for communicating necessary information with coreboot. In the following, both of these components are described in further detail. We start by offering details on the design of our Linux component, followed by a description of the functionality implemented in coreboot required for Suspend-to-CXL.

3.2.1 Linux

The first stage of our implementation is executed in the Linux kernel when suspend is requested. We require the following steps to be completed:

1. Move user space pages to CXL device
2. Identify physical pages used by the Linux kernel in DRAM
3. Identify physical pages used by the user space applications in CXL memory
4. Save metadata information on kernel memory usage for coreboot to process

In the following, the aforementioned steps are described individually in greater detail.

Moving user space

To move pages from Linux between host memory and CXL memory we capitalize on the fact that the CXL device memory and host DRAM are located on two separate NUMA nodes. As mentioned in Section 2.4.2, Linux provides a “page migration” API, which is capable of migrating memory between NUMA nodes. We employ this mechanism to move all user space processes to the CXL device before handing over control to firmware on suspend. Choosing this existing mechanism conveniently relieves us from having to manually track used memory on the CXL device and update the kernel’s memory data structures. We explore possible enhancements to performance of this procedure in Chapter 6.

Identifying physical host memory usage

Since our goal is to save and then restore only data required for the Linux kernel to resume, we need to identify which pages in physical memory this data resides in. To achieve this, we walk the Linux buddy allocator’s data structures, which provide information on free page ranges of different sizes. We then simply “invert” this information on free pages to receive the memory ranges that are in use. While this approach provides us with all pages used by the Linux kernel, it is likely not fine-grained enough to supply the minimal set of pages required for Linux to resume, allowing for further optimizations, as examined in Chapter 6.

Identifying physical CXL memory usage

As we copy the Linux kernel’s memory to CXL memory from coreboot, it is essential to explicitly inform coreboot of the memory ranges on the CXL device that are already occupied by user processes to prevent overwriting any data. Specifically, we lack direct control over the placement of user data on the NUMA node corresponding to CXL memory, making it impossible to simply relocate everything to a contiguous block. It is therefore necessary to track the entire CXL memory space instead of a simple upper boundary. Luckily, because we use the existing page migration mechanism of the kernel, the pages in CXL memory used by user space processes are also allocated by the page allocator. As a result, we can traverse the allocator’s data structures for the NUMA node corresponding to CXL memory in the same manner as for host memory to find the used pages on the CXL device.

Saving metadata for coreboot

Lastly, the gathered information on used memory has to be saved for coreboot to use. To convey this information, we employ two bit vectors as a memory map, one for host and one for CXL memory. Semantically, a set bit at index `i` represents the physical page at page frame number `i` being in use in the respective memory space. To convey this information to coreboot, we reserve some memory at the beginning of the CXL memory space as a metadata header tracking important information for performing Suspend-to-CXL. The implementation details and members of this header are further explained in Section 4.2. Placing metadata in CXL memory yields the advantage of both coreboot and Linux being able to easily read and write to the contained fields while simultaneously being persistent across reboots to ensure the system is resumed correctly.

3.2.2 coreboot

Next, coreboot is responsible for persisting kernel memory and powering the machine off. To do this, the following steps are performed:

1. Identify that the system is about to suspend and call Suspend-to-CXL logic
2. Copy marked kernel memory to CXL device memory
3. Save the ACPI wake vector
4. Mark the system as being suspended
5. Power off the system

Jumping to S2CXL logic

After Linux finishes its suspend procedure, it generates an SMI as described in Section 2.3.2 to enter S3. We adapt the SMM trap handler in coreboot to call into our suspend function. Building the coreboot suspend component into SMM allows us to program in C and use the elevated privilege level of SMM to access host DRAM as well as CXL memory. Unfortunately, the majority of coreboot's data structures generated during initialization, including the E820 tables and proximity domains, become inaccessible in SMM. Hence, it is necessary to save this information to the metadata header during the boot process, specifically within the ramstage which is described in further detail in Section 4.4.

Copying marked kernel memory to CXL memory

To copy kernel memory to CXL memory, we traverse the bit vector mapping all physical pages of host DRAM provided by Linux as described in Section 3.2.1. This allows us to identify contiguous ranges, which can be copied at once, sidestepping the inefficient burden of copying memory page-by-page. Since only memory belonging to the operating system is to be copied, we filter out memory ranges belonging to firmware or marked as hardware-reserved in the E820 table. We successively copy each range to CXL memory, resulting in a contiguous range of CXL memory being occupied. As we already copied user space memory to CXL from Linux using the page migration mechanism, we must be cautious not to overwrite this data. For this reason, we skip over the pages marked as being used in the CXL bit vector and split up memory ranges when needed to ensure that the entire memory space supplied by the CXL device is usable for persisting data.

Saving the ACPI wake vector

As described in Section 2.3.2, ACPI tables contain an address called the "ACPI wake vector." This value is written into the table by the OS on suspend. When the system is suspended to conventional S3, RAM remains powered, so that the wake vector can be read on resume. However, as our implementation is designed to power off the system during suspend, and we only save kernel and user application data, but not the ACPI tables located in the firmware-reserved regions of RAM, this information is lost. Therefore, we obtain the wake vector from the ACPI tables in SMM before powering off the machine and save it to our metadata header located at the beginning of CXL-memory.

Marking system as being suspended

When the system is powered on again, the firmware begins initializing the system. At some point, coreboot must detect that the system was suspended, or else it will simply perform a normal boot sequence. To keep the implementation simple, this is done by adding the boolean `suspended` value to our header in CXL memory, which is set to true right before the system powers off. Another possibility would have been to create an EFI variable [65, Chapter 8.2], i.e., a UEFI-defined key-value mapping, which can be accessed from firmware as well as from the operating system. EFI variable storage does not require system power, as data is typically saved to Non-volatile Memory (NVRAM) chips, e.g. flash, located directly on the mainboard. Despite these advantages, using EFI variables requires additional logic and can be less reliable, which is why we opted for the simpler approach of writing the variable to the metadata header discussed in detail in Section 4.2.

Powering off the system

As a last step to suspending the machine, the system is placed in ACPI S5 as described in Section 2.3.2, waiting to be powered on again for the resume mechanism to begin. Placing the system in S5 allows for minimal energy consumption during sleep, thus contributing to the goals described in Section 3.1.

3.3 Resume

In the following section we describe the steps necessary for the system to resume successfully from CXL memory. Similar to suspend, our resume algorithm is also divided into a coreboot and a Linux component. While coreboot performs similar tasks as on suspend, i.e., restoring kernel memory, the Linux component refrains from copying any memory directly. Instead, it queues page migrations for asynchronous execution. In a manner similar to the previous section, the following sections details the roles played by coreboot and subsequently by Linux during the process of resuming from Suspend-to-CXL.

3.3.1 coreboot

On suspend, our code is called from the SMI handler in case of an S3 request. When the system is powered on again, initially no SMM is set up and no way to enter it is available, necessitating direct modification of existing functions to ensure our resume code is called correctly. Since the system was previously fully powered off, coreboot must reinitialize the machine again by completing the stages outlined in Section 2.2.1, including most of the ramstage. However, before the ramstage completes, our code must be invoked to prevent coreboot from loading a new instance of the payload, and consequently, a new instance of the operating system. Unfortunately, we are unable to utilize the code branch of conventional S3 as detailed in Section 2.3.2. This stems from the fact that S3 preserves the entire system memory, eliminating the need for coreboot to write the ACPI and coreboot tables to memory during resume, as these tables persist from the previous boot sequence. In contrast, our approach fully powers off the system, and only saves the memory ranges usable by the operating system, requiring the tables to be rewritten upon resume. Instead, we must ensure that the function responsible for executing our resume algorithm and subsequently jumping to the ACPI wake vector is invoked during the ramstage, prior to booting the payload. To accomplish this, we branch to our resume code right before coreboot would, under normal circumstances, load the payload. This shortcut comes with the added benefit of eliminating the need to decompress the payload, thereby conserving valuable resume time, as covered in Section 5.2. Calling into our Suspend-to-CXL code after coreboot has finished initializing the system ensures that the machine is set up properly and able to jump to the operating system. Analogous to suspend, our coreboot resume component must complete these tasks:

1. Identify that the system was suspended
2. Copy back marked kernel memory to host DRAM
3. Cleanup to ensure that subsequent suspend calls succeed
4. Jump to the ACPI wake vector

Identifying suspend

We employ our saved metadata to identify whether the system was suspended just before coreboot would normally load the payload at the end of ramstage. If suspend is detected, we transition to our resume logic, where we handle resuming the system, thus skipping loading and execution the payload entirely.

Copying marked kernel memory to host memory

Similarly to suspending, resuming involves traversing the reserved memory bit vector again, this time using the page frame numbers as destination for the copy operation. We fill these ranges sequentially with memory from the CXL device. To ensure we copy the correct memory back we use the CXL bit vector once more, skipping the memory ranges marked as "used," as these contain user data we do not want to copy back. Again, we ignore firmware-reserved memory regions. Skipping these regions is possible and even necessary to ensure correct operation, because the firmware-reserved parts of memory have freshly been set up by coreboot, likely with different contents entirely. Copying the old contents back to these regions would thus result in an inconsistent state and potentially even crash the system.

Cleanup

We want to ensure that subsequent suspend-resume cycles succeed as well. For this reason it is necessary to clean up the metadata header structure. First, the `suspended` boolean is reset to `false`, ensuring that the next normal boot correctly loads the payload instead of erroneously executing the resume code and jumping to the wake vector. Additionally, we zero the memory of both of the bit vectors to avoid unnecessarily copying pages in case the system is to be put to sleep again.

Jumping to wake vector

As previously stated, we explicitly skip over regions marked as firmware-reserved. Consequently, the wake vector contained in the ACPI table is lost when RAM loses power upon entering S5 during suspend. To resume to the OS we instead jump to the wake vector we saved in our header data structure in CXL memory. Since Linux saves the wake vector to the ACPI on every suspend request, writing it back to the table from coreboot is not necessary.

3.3.2 Linux

After copying the required data back to DRAM and jumping to the wake vector, Linux resumes execution, enabling secondary CPUs and restarting user space. Our implementation then requires some last steps to complete the suspend/resume cycle:

- Perform priority analysis of processes
- Migrate user pages back to host DRAM in order of priority
- Run the above tasks asynchronously

We modify the kernel to call our resume code right before exiting the S3 sleep code path. As page migrations occur asynchronously, all calls made in our resume code are non-blocking. This behavior results in our resume function returning almost instantly, ensuring that resuming the system completes quickly, which is further analyzed in Section 5.2. At this point, we capitalize on the capability of CXL to provide byte-addressable memory mapped to the physical address space by allowing user processes to continue their execution. Instead of accessing their data in system RAM, user applications then transparently use the memory on the CXL device, as their pages were migrated to CXL memory on suspend. In the following, we supply comprehensive details on the above-mentioned steps.

Priority analysis

For achieving a satisfactory user experience in general, processes the user interacts with should feel responsive, especially directly after resume. As previous work by Sun et al. [43] shows, running applications on CXL memory induces a significant impact on performance. For example, the authors identify an approximately $2.2\times$ lower memory access latency and $7.7\times$ lower maximum bandwidth when using the same non-temporal store instructions we use in this thesis for CXL memory compared to the DDR5 memory of their test system. We strive to ensure that the processes for which the user could notice these performance impairments are migrated back to host memory first. Toward simplifying our approach, we decide to complete migration of pages back to system memory on a per-process granularity, as opposed to a per-page granularity. Our reasoning for this design decision is that determining priorities per processes is both algorithmically easier and more time efficient. Further optimizations to this priority analysis are explored in Chapter 6. Additionally, desktop computers run certain applications, such as desktop environments that are more latency-critical than background processes, for example. For instance, the user might not care about a memory-hungry background process requiring a few more seconds to finish. Moreover, migrating a process with significant memory usage back to host memory can be time-consuming, adversely impacting the performance of latency-critical processes still running on CXL memory. If this background task were to be relocated to host DRAM before an interactive process, such as the aforementioned desktop environment, the overall user experience would deteriorate. Therefore, we choose to use a process's interactivity as a criterion for prioritizing its restoration to system memory. To achieve this, we leverage Linux's extensive process-specific tracking of scheduling data, specifically using the time a process last started running on a CPU, known as the arrival time, as a measure of priority. We associate a recent arrival time with high a priority. The rationale behind this is that a process that frequently runs and quickly releases the CPU likely corresponds to the type of interactive application we aim to prioritize, as opposed to a process that runs less frequently but for longer durations.

Migrating user pages back to system memory

After retrieving a list of current processes in the system and subsequently sorting them based on the priorities described earlier, we are ready to migrate back user pages to host memory. To do this, we employ the same page migration API as on suspend, detailed in Section 3.2.1, this time migrating from the CXL node to the node containing system memory. In theory, the kernel API supports different migration modes defined in the enum `migrate_mode` [61, `include/linux/migrate_mode.h`]. These modes include settings for both synchronous and asynchronous migration, with a particularly noteworthy mode for non-CPU copying, which could benefit from accelerators such as Intel's Data Streaming Accelerator [15] explored further in Chapter 6. However, the high-level function utilized in this work exclusively employs synchronous migration [61, `mm/mempolicy.c`]. Therefore, an alternative mechanism is required to achieve our objective of asynchronously migrating pages. In the next section we present further details on how we implement the asynchronous migration of user pages after resuming.

Asynchronous execution

Toward keeping resume durations as short as possible, we avoid migrating pages synchronously like we do on suspend. Instead, we tolerate reductions in application performance in favor of reducing resume durations by first resuming tasks and then migrating pages back to host DRAM asynchronously. To accomplish this asynchronous migration, we employ the Linux kernels' workqueue API [11]. The workqueue (wq) API provides an interface for submitting tasks to a queue, which will then be executed asynchronously by a kernel-owned thread called "worker." The kernel maintains a pool of these workers, supporting concurrent processing of tasks. The extensive API allows for the use of highly customized queuing tailored to the current use case. For example, tasks can be submitted to two distinct worker pools, one for normal jobs and one for high priority work. Additionally, the API exposes various configuration options, such as cache affinity, i.e., whether the job is only allowed to run on specific CPUs. We submit our asynchronous restore function to the global workqueue as we do not have any special requirements for our task, which could be satisfied by the more specialized functions the API provides.

Chapter 4

Implementation

In the following chapter, we present our implementation of Suspend-to-CXL, following the theoretical design we described in the previous chapter. When building our sleep implementation on commodity hardware, the selection of mainboards was severely limited due to our extensive requirements. Specifically, our implementation requires a board that supports CXL and is compatible with coreboot as its firmware. Although coreboot supports CXL and initialization thereof is the responsibility of the FSP, as characterized in Section 2.1, none of the commercially available mainboards currently maintained by upstream coreboot are capable of using CXL. For this reason, we utilize a coreboot port to the ASRock SPC741D8-2L2T/BCM [1] mainboard, developed by Meyer and Khalil [25]. In the following, we simply refer to the SPC741D8-2L2T/BCM as the "ASRock board." Most importantly, the board's PCIe slots support the CXL protocols outlined in Section 2.4. This allows the integration of the persistent storage modules needed for Suspend-to-CXL, making the ASRock board well-suited to our use case. In the following, we discuss necessary modifications, crucial additions, and other implementation details of implementing Suspend-to-CXL in coreboot and Linux.

4.1 Using CXL memory

Although CXL memory is set up by the FSP (Section 2.2) and can from then on be addressed identically to normal system memory, making it very comfortable to access from both firmware and the operating system, this behavior is not entirely unproblematic for our use-case. Specifically, coreboot marks the CXL memory region as normal system RAM in the E820 tables described in Section 2.1. From Linux's point of view, the region contains perfectly usable memory, simply located on a different NUMA node. For our implementation, we wish to retain as much of this CXL memory as possible for our suspend purposes even when the system experiences high memory loads, which is why we attempt to prevent Linux from using this region for normal allocations. In theory, this problem can be approached from two angles: From firmware and from within Linux.

A firmware-based solution would include marking the entire CXL memory range as firmware-reserved in the E820 table, prohibiting Linux from using it entirely. Despite the simplicity of this implementation, it is not suitable for our needs. Most importantly, Linux would not create the data structures required for using CXL memory as a target for the page migration mechanism mentioned in Section 2.4.2. Having to manually remap pages would drastically increase the complexity of our Linux components, especially for pages also shared with the kernel, which is why we avoid this approach.

Instead, we opt for an OS-centric path to achieve our goal. Luckily, only minor changes in a handful of files, which are described in Section 4.3.1, are needed in order to prevent Linux from allocating user memory from the CXL device. On the one hand, it is necessary to ensure that memblock, i.e., the allocator described in Section 2.5 used during early boot phases, does not allocate CXL memory while setting up the operating system. On the other hand, we must make changes to the system default memory policy. Section 2.4.2 highlights the allocation rules enacted by the system default policy. Essentially, allocations are attempted from the memory belonging to the node the process is running on. In this situation, no memory is ever allocated from the CXL device, since the CXL node possesses no CPUs. The problem only arises if physical RAM is exhausted, as allocating memory from other nodes is permitted in these cases. To deal with this issue, the hard-coded system default policy is adjusted to use the `MPOL_BIND` mode, which only allows for allocations to be satisfied from the specified nodes.

4.2 Cross-component Communication

Chapter 3 explains how we leverage Linux as well as coreboot for our implementation. This split software stack necessitates a method for passing data between these distinct components. Fortunately, since both Linux and coreboot are primarily written in the C programming language, the same data structure definitions can be shared between the two. To facilitate this, we place a structure at the beginning of CXL memory, containing the fields necessary for performing Suspend-to-CXL. This structure is mapped to virtual memory in both coreboot and Linux, enabling direct access to its members due to CXL memory's byte-addressable nature. Furthermore, this structure is expected to persist across resets, as Suspend-to-CXL is designed to utilize persistent CXL memory for storage. An excerpt of the fields contained in our metadata header struct is given in Table 4.1.

Name	Type	Description
`suspended`	boolean	True, if the system is currently suspended to CXL, false otherwise.
`wakeup_vector`	pointer	Pointer to the ACPI wake vector, must be saved before power off
`bootmem_ram`	array of E820 entries	Entries of the E820 table representing system memory usable by the operating system.
`ram_pfns`	bit vector	Bit vector representing a memory map of system memory. For each page frame number, a set bit indicates that the page is in use.
`cxl_pfns`	bit vector	Bit vector representing a memory map of CXL memory. For each page frame number, a set bit indicates that the page is in use.

Table 4.1: Example definition of the metadata header we employ to communicate and transfer data between coreboot and Linux. We place this header at the beginning of CXL memory in order to ensure it is available after the system is powered off during sleep. The exact sizes of the bit vectors depend on the size of installed system and CXL memory.

In particular, the struct contains bit vectors mapping out system as well as CXL memory and the parts of system memory usable by the OS in form of E820 table entries. In the following, we describe how these fields are used to complete the tasks necessary in Linux and in coreboot to successfully perform Suspend-to-CXL.

4.3 Linux Kernel

We build our mechanism for the latest stable Linux kernel version at the time of writing this thesis, version 6.9.6 [61]. In order to maximize compatibility with different versions, we try to keep our modifications to the kernel source code as slim as possible. The implementation of our Linux components consists of changes to two files. First, we create a header file, `include/linux/s2cxl.h`, containing constants and structure definitions such as our header struct. Second, the suspend and resume algorithms are implemented in `kernel/power/suspend.c`. Additionally, minor changes to other files are necessary, for example to prevent

Linux from allocating from the CXL NUMA node when memory is scarce. In the following, we present implementation details and describe necessary changes to the Linux kernel source code for implementing Suspend-to-CXL on our ASRock board.

4.3.1 Memory Management

By default, Linux uses all available memory when confronted with memory pressure. Since we do not want the OS to treat CXL memory as normal memory that can be utilized for general purpose allocations and instead want to preserve this memory for suspending the system, as outlined in Chapter 3, we must prevent allocations explicitly. Toward hindering Linux from claiming memory for allocations on the CXL device, individual changes are required for both boot-time allocations and allocations when the system running.

As described in Section 2.5, the Linux kernel leverages the memblock allocator [53] in its early boot stages. Among other tasks, memblock must decide whether to begin allocating from the bottom, i.e., at low physical addresses or from the top of physical memory, returning high physical addresses. Toward the goal of maximizing available storage space on the CXL device for suspend, this decision is very important, as some structures allocated by memblock, such as page tables, are kept in memory for the entire lifetime of the operating system. We ensure that `memblock_set_bottom_up` [61, `arch/x86/mm/init.c`] is called with its parameter set to `true` to prevent memblock from allocating CXL memory, which is located at the top of the physical address space. Furthermore, we hard code the NUMA node ID in the memblock allocation function [61, `mm/memblock.c`] to the default node containing system DRAM, ensuring NUMA-aware allocations are only satisfied from DRAM. These measures ensure that memblock refrains from placing vital kernel data structures in CXL memory.

Additionally, we must prevent the page allocator from handing out pages from CXL memory. Fortunately, this only requires changing the static system default policy to use the `MPOL_BIND` mode. To implement this, we add code in `numa_policy_init` [61, `mm/mempolicy.c`] to configure the default policy to use `MPOL_BIND` and only allocate from the node containing system memory, which is node 0 in our case. Furthermore, we disable the boot-time interleaving of nodes as our system possesses sufficient memory on a single node to boot successfully.

Finally, an additional modification to Linux's memory management is required for the implementation of Suspend-to-CXL. By default, automatic NUMA balancing [35] ensures that application memory is kept near the CPU where the corresponding process is executing. However, since we relocate memory to the NUMA node associated with CXL memory, which lacks any CPUs, this memory is immediately transferred back to system memory upon resume. This behavior interferes with our ability to collect accurate timestamps and other evaluation data and thus must be prevented. To address this, we disable automatic NUMA balancing by appending the `numa_balancing=disable` parameter to the Linux kernel command line [55]. The above changes ensure that Linux's memory management system does not interfere with the Suspend-to-CXL.

4.3.2 Suspend

To integrate our suspend logic into Linux's suspend procedure, it is necessary to call our hooks on both suspend and resume. To guarantee execution of our code, we add a call to `s2cxl_suspend` in the `suspend_enter` function [61, `kernel/power/suspend.c`]. Our suspend component begins by mapping the CXL header, which is located directly at the beginning of CXL memory space, to virtual memory to be able to access its members directly. We request a new mapping of the physical pages to a virtually contiguous range via `vmap` as described in Section 2.5. To reduce the amount of memory copied in coreboot, we flush Linux caching data structures. Specifically, we drop the page cache to avoid having to needlessly copy pages that can be retrieved from persistent block storage anyway. Additionally, we free the cached blocks of the slab allocator. Before identifying used memory, we make use of the Linux kernel's page migration mechanism. We call `do_migrate_pages` [61, `include/linux/mempolicy.h`] for each user process with arguments ensuring that all the memory used by the process is migrated to the NUMA node containing CXL memory. After moving user memory to persistent storage, we begin identifying memory used by the kernel. For convenience, we begin by setting all bits of both bit vectors to 1, as we can then simply set the bits we receive from walking the free lists of the buddy allocator back to 0, resulting in the semantics described in Section 3.2.1. For each free page, we unset the corresponding bit in either the CXL bit vector or the system memory bit vector, depending on where the page is located in physical memory. After finishing walking the buddy allocator, we are left with two bit vectors mapping the used regions of both system and CXL memory. We complete the Linux component of suspend by copying both bit vectors to the header at the beginning of CXL memory.

4.3.3 Asynchronous Resume

After coreboot has copied kernel data back to system memory and jumped back to Linux, the Linux resume code executes almost identical to when resuming from ACPI S3. Necessary setup such as bringing other CPUs back online and restarting user applications is completed before unfreezing user space processes.

We add a call to our resume function, s2cxl_resume, at the very end of Linux's S3 resume procedure in enter_state [61, kernel/power/suspend.c], right before user space is restarted. This function then returns quickly, only performing a single call to schedule_work [61, include/linux/workqueue.h], scheduling the function s2cxl_async_migrate for execution by the kernel workqueue framework as covered in Section 3.3.2. The queued function is then executed asynchronously to the rest of the Linux resume procedure.

The asynchronous migration function begins by performing a priority analysis on all user processes to determine the order in which processes are migrated back to system memory. This is done by sorting processes by the last_arrival field of the sched_info struct [61, include/linux/sched.h] of each thread. For the task of sorting, we employ Linux's sort function [61, linux/sort.h], which internally utilizes heapsort. In comparison to quicksort, heapsort has an average-case and worst-case time complexity of $\mathcal{O}(n \log n)$, allowing for more consistent sorting durations by avoiding quicksort's $\mathcal{O}(n^2)$ worst-case time complexity. Afterward, the sorted list of processes is traversed, migrating the pages of each task back to host memory. Upon finishing this step, the asynchronous resume function exits and thus fully concludes the suspend/resume cycle.

4.4 coreboot

As mentioned in Chapter 3, we make use of coreboot's weakly linked symbols to execute our functionality. Unfortunately, this approach is not sufficient for fully implementing our mechanism, which is why it is necessary to directly modify the source code in a few locations described in the following.

mainboard_smi_sleep

As described in Section 3.2.2, we use the SMI generated by the operating system when suspending to execute our custom code. The SMI handler in coreboot identifies that the system is to be suspended and determines the requested sleep state. Before performing the ACPI state transition, it calls the weakly linked function mainboard_smi_sleep [51, src/include/cpu/x86/smm.h]. We override this function to test whether calling our suspend code is necessary. Specifically, we jump to our suspend entry point only when the requested sleep state corresponds to the constant representing S3. This ensures that coreboot does not execute our suspend logic during other transitions, such as system shutdown. Example code we employ for performing this procedure can be found in Listing 4.1.

Since S3 is fundamentally not supported on our platform, we do not require additional logic to maintain the capability of executing regular S3 and simply perform S2CXL on every S3 request. Nonetheless, other boards still supporting S3 might want to retain this option by including additional conditions in `mainboard_smi_sleep` before jumping to the Suspend-to-CXL entry point. Enabling conventional S3 only requires changes in the corresponding mainboard's directory, no alterations to the existing Suspend-to-CXL code are necessary.

Listing 4.1: Minimal implementation for intercepting the default S3 code path and calling Suspend-to-CXL logic.

```
void mainboard_smi_sleep(u8 slp_typ)
{
        if (slp_typ == ACPI_S3) {
                s2cxl_sleep();
        }
}
```

bs_payload_load

According to Section 3.3.2, we call our custom resume code at the end of coreboot's ramstage, immediately before the payload is loaded. Specifically, we modify `bs_payload_load` [51, `src/lib/hardwaremain.c`]. Here, we query the header to find out whether the system is suspended and call our resume function, skipping the rest of the ramstage, which mainly consists of loading and decompressing the payload. Bypassing payload decompression allows us to save considerable time on resume, which is explored further in Section 5.2. In case the machine was not suspended, i.e., is currently completing a normal boot sequence, we handle necessary tasks for subsequent suspend calls to complete successfully. In particular, we initialize the metadata header, as outlined in Section 4.2, which includes saving the E820 table. This table is essential for managing memory during both the suspend and resume processes as described in Section 3.2.2 and Section 3.3.1. Persisting the E820 table in our header is necessary, since coreboot does not permit reading it in SMM.

4.4.1 Memory Management

Although firmware can, in theory, simply write to any valid memory location it pleases, we want to avoid overwriting any operating system or application data, as our code runs after the operating system has booted. For this reason, we create entries in the system memory map and mark these additional regions as firmware-reserved by making use of coreboot's weakly linked symbols once more, preventing Linux from using our reserved ranges.

bootmem_platform_add_ranges

When setting up the machine, this hook [51, `src/include/bootmem.h`] is called to allow for platform components to reserve memory regions for themselves. These reserved ranges are then marked as firmware-reserved in the E820 table, signaling the operating system to refrain from using them. We implement this hook and reserve two memory ranges.

First, we reserve the memory occupied by our metadata header, which is characterized in Section 4.2. If memory is marked as normal RAM in the memory map, the operating system may do as it pleases with this region, for example allocate from it on requests from applications. To avoid our critical data structures being overwritten, we prevent Linux from using the region altogether. As touched on in Section 4.3.1, we maintain the ability to explicitly map the header to virtual memory and access it from Linux, nonetheless.

Second, we require some space for the dynamic allocation of data structures required for our implementation. As described in Section 3.2.2, we copy memory to the CXL device while skipping pages marked as being used in the CXL bit vector. Our resume algorithm relies on the assumption that both bit vectors remain unchanged during sleep, offering the same information on traversal as during suspend. This allows us to copy the kernel data back to its original location in host memory, where it was situated prior to suspending. To avoid having to walk the CXL bit vector for every copy operation, we walk it once at the beginning of our suspend and resume procedures, creating a list of structures representing a free, contiguous range of CXL memory. The amount of structures created this way is unknown at compile time and can grow very large depending on the degree of fragmentation. For this reason, we create a simple arena allocator that is tasked with answering allocation requests. The allocator exposes a straight-forward `malloc`-style interface for allocating new memory from the reserved area and only supports freeing the entire space at once. Despite its simplicity, this interface is sufficient for our use case, since we only request new memory when traversing the CXL bit vector once during suspend and once during resume.

4.4.2 Copying Memory

Owing to CXL memory being addressable in the same way as host memory, we do not require any setup or complicated logic for copying memory between system DRAM and CXL memory.

coreboot offers a traditional `memcpy` [51, `src/include/string.h`] interface for copying *n* bytes from a source to a destination address, which would satisfy our needs. However, `memcpy` in coreboot is rarely tasked with moving gibibytes of data at a time and thus not optimized for this use case. For this reason, we create our own `memcpy` implementation, `s2cxl_memcpy`. To improve transfer speeds, we capitalize on the "direct stores" instruction set extension [20, Volume 2B, Chapter 4] available on newer Intel processors such as the Sapphire Rapids CPU running in our test system. To construct our custom `memcpy` implementation, we create a new Kconfig variable, `CPU_HAS_MOVDIR64B`, and select the option for our ASRock board. If the variable is set, `s2cxl_memcpy` leverages the `MOVDIR64B` instruction and falls back to regular `memcpy` on systems that do not support the extension. Whether a processor supports the extension is also indicated by the presence of the CPUID feature flag for `MOVDIR64B` [20, Volume 2B, Chapter 4] at runtime. Since we require `memcpy` to be as performant as possible, we avoid checking the feature flag at runtime and instead rely on the config option being set at compile time.

Basing our `memcpy` version on the `MOVDIR64B` instruction grants two key benefits over other memory operations. First, the number of bytes moved with `MOVDIR64B` exceeds the logical memory data bus width, which is 8 bytes on x86_64. Instead, it "moves 64-bytes as direct-store with 64-byte write atomicity from source memory address to destination memory address" [20, Volume 2B, Chapter 4]. Moving 64 bytes at once compared to only 8 bytes for coreboot's memory copy operation results in fewer instructions that have to be executed. Second, when copying memory with `MOVDIR64B`, the processor completely bypasses the entire cache hierarchy, i.e., refrains from both writing data to a cache line and fetching the corresponding data from memory into cache. For conventional `memcpy` operations, moving the copied data into CPU cache is usually beneficial, as the copied data might be accessed afterward. However, our use case requires moving data in bulk without ever accessing it afterward, even resetting the entire CPU before reading it the next time. For this reason, `MOVDIR64` is perfectly fit for our situation, resulting in our `memcpy` implementation averting the latency penalty incurred by storing data in cache on every copy request.

4.4.3 System Reset

Unfortunately we did not have access to real persistent CXL storage devices for this thesis. As mentioned in Section 2.4.1, we avert this roadblock by using a CXL-capable FPGA as our storage backend. While the exposed functions of the FPGA are identical to that of real hardware, its performance characteristics and attributes are quite different. Most importantly, the FPGA is equipped with DDR4 memory to expose as CXL memory, which is not capable of persistently storing data and thus would essentially defeat the entire purpose of our proposal. Therefore, to retain the capability of implementing and evaluating our mechanism, we must abstain from fully powering off the system. Instead, we perform a "warm" reset, meaning the CPU is reset without ever power-cycling the board. Specifically, this enables the CXL FPGA to remain powered, thereby preserving the contents of its DRAM and preventing data loss. This method of resetting enables the CXL device to behave as though it can persistently retain data across reboots. Issuing a warm reset in place of properly powering off the system entails some side effects, which is discussed in further detail in Chapter 5. It is essential to highlight that this limitation arises from the currently available hardware and our test system, rather than from any inherent issues with our implementation. With an actual persistent CXL device, such as the ones mentioned in Section 2.4.1, the system is placed in ACPI S5 instead of being reset, thus enabling the use of our implementation in real-world scenarios.

4.5 Development and Debugging

The following section examines the challenges and opportunities associated with working with firmware. We give details on the steps required to configure, develop, and deploy coreboot to physical hardware. Furthermore, we discuss firmware-related problems linked to 64-bit execution of coreboot we encountered and how we managed to solve them. Lastly, we review the debugging capabilities that firmware provides.

4.6 Deploying coreboot

Modifying firmware behavior requires recompiling the coreboot project and writing the output binary to the mainboard's firmware flash chip. Depending on the hardware, this can require different hardware and software tools. Fortunately, the ASRock board we utilize features a socketed flash chip for its firmware image, offering two significant advantages. First, the flash chip can be easily removed from its socket, allowing the use of an external flash programmer to write the image before reinserting the chip. For this task, we used a Raspberry Pi Pico [33] running the pico-serprog [31] firmware together with the `flashrom` [8] command line utility, which supports providing a reference file containing the current flash contents. Supplying flashrom with the current flash contents eliminates the need for reading the entire data before flashing in order to determine regions that have to be replaced, which results in drastically shorter write durations. Second, we do not require the machine to be able to boot back to the OS to flash a different firmware image. This way, if booting fails we can simply remove the firmware chip, flash the patched binary and boot successfully after inserting the chip back into its socket.

As described in Section 2.2.2, coreboot employs a wide variety of Kconfig files containing variables signaling hardware capabilities and enabling or disabling functionality based on these capabilities at compile time. From these individual Kconfig files a single `.config` file in the root of the directory structure is generated, which contains the configuration used when compiling the project. Modifying these variables can be performed by directly editing the respective Kconfig file, which permits committing these changes to a git repository, allowing persisting of the configuration for others to utilize. Additionally, the Kconfig system provides a configuration terminal user interface, enabling navigating settings by categories and even supplying a search function, which proved very useful for finding the defining Kconfig file. Upon quitting, values can then be saved directly to the `.config` file. While variables can be changed quickly via the menu, these changes are not reflected in version control and overwritten when changing target mainboards for example. For this reason, we found using the menu to only be practical for quickly changing single variables and preferred changing the Kconfig files directly. Typically, most of the configuration options, such as supported CPUs, valid architecture, and hardware capabilities are set in the Kconfig file of the target mainboard and then transferred to the global `.config` by building the config of the target mainboard via `make defconfig`.

4.6.1 64-bit coreboot

Traditionally, coreboot is built for 32-bit execution, which is sufficient for most situations. However, as touched on in Section 2.2, we require 64-bit execution to be able to address the memory of our CXL device, which is placed well above the 4 GiB limit for 32-bit addressing. Although the coreboot port of Meyer and Khalil we utilize supports 64-bit execution of all stages, some minor changes to configuration, as well as to code, were necessary.

For example, coreboot uses a simple identity page mapping scheme [50], essentially creating a 1:1 mapping from virtual address to physical address. By default, coreboot builds this identity mapped page table only for the lower 4 GiB of memory, even if 64-bit execution is activated and thus the CPU could address more than 4 GiB of memory. To circumvent this limitation, coreboot instead supports the use of 1 GiB pages, allowing for the identity mapping of up to 512 GiB of physical memory. We activate this option via the `USE_1G_PAGES_TLB` Kconfig variable [51, `src/cpu/x86/Kconfig`].

While the above changes enable the construction of 1 GiB pages for the regular coreboot page table, this does not apply to SMM. As mentioned in Section 2.1.2, SMM lives in its own address space to prevent exploit capabilities. As a quick and dirty fix to allow coreboot to address memory over 4 GiB in SMM as well, we modify the SMM entry code to simply load the page table used by coreboot's other stages instead of the SMM page table. While this is sufficient for our implementation, it is not a viable long-term fix for the security reasons mentioned in Section 2.1.2.

4.6.2 Debugging

During development of firmware and low-level operating system functionality, debugging capabilities are severely limited compared to traditional user space applications. Debugging in coreboot is primarily possible through the use of the `printk` function [51, `src/include/console/console.h`]. These log messages can then be placed in a buffer in CBMEM as described in Section 2.2.3. While this does not demand any additional hardware capabilities, retrieving logs requires the operating system to boot successfully. Fortunately, our ASRock board is equipped with a serial port, which supports logging from both coreboot and Linux. For coreboot, this requires setting the `CONSOLE_SERIAL` Kconfig variable to `true`, configuring the baudrate, and specifying the correct I/O port address. When configured correctly, all output generated via `printk` is outputted over the serial port and can be followed in real time on a different system via a serial communication program such as `minicom` [66].

An additional advantage of debugging via serial is the ability to output individual bytes directly and without buffering from assembly code, as illustrated in Listing 4.2. This capability was particularly useful when debugging the transition from coreboot to Linux during the resume process, where the low-level resume code failed before the Linux console could be properly initialized.

```
mov $0x3f8, %dx
mov $'a', %al
out %al, (%dx)
```

Listing 4.2: x86 assembly code to output the single character "a" to the serial port located at I/O port `0x3f8`.

Outputting all messages logged to the kernel ring buffer to the serial console can be configured on Linux via the `console` kernel command line parameter [55], which enabled us to debug Linux's early initialization code before VGA output becomes available. The ability to read log messages produced by low-level code in real time proved invaluable during the development of our sleep implementation.

Chapter 5

Evaluation

In the following, we analyze the performance of our implementation of Suspend-to-CXL and compare the results to other available sleep modes. While a comparison of Suspend-to-CXL with more widely used sleep modes, such as Suspend-to-Idle and S3, would have been desirable, neither of these modes functioned correctly at the time of writing. Although Suspend-to-Idle could be successfully initiated and exited, power consumption remained unchanged compared to the system's idle state. In contrast, while the system successfully entered S3 sleep and achieved energy savings, it was unable to resume operation due to the lack of support from the FSP. For this reason, Suspend-to-CXL is matched against Linux's implementation of S4, hibernate, further detailed in Section 2.3.3. We perform our evaluation according to the requirements defined in Section 3.1: resume latency, suspend latency, reliability, and energy consumption. Before presenting our results for each requirement, we give a detailed description of our test setup and measurement methods.

5.1 Methods

As mentioned in Chapter 4, we implement and evaluate Suspend-to-CXL on the ASRock SPC741D8-2L2T/BCM server mainboard. We pair this board with a 4th-Gen Intel Xeon Silver 4410Y CPU [17] (12 cores, 24 threads), which is part of the Eagle Stream server platform. The SPC741D8-2L2T/BCM supports CXL 1.1 at PCIe 5.0 x16 speeds [1], which, although superseded by the CXL 3.0 specification, is sufficient for our implementation. The system contains a single module of Kingston 16 GiB DDR5-4800 ECC memory [21]. Power is provided by a Seasonic PRIME TX 650 W power supply [38]. The PRIME TX is 80 PLUS [3] certified, a widely recognized certification program for power supplies, and holds the highest certification level, Titanium. In the 80 PLUS certification scheme for

230 W power supplies sold in the EU, Titanium is the highest available rating and attests efficiencies of 90 %, 94 %, 96 %, and 91 % at loads of 10 %, 20 %, 50 %, and 100 %, respectively. Due to our Xeon CPU having no onboard GPU, we use a dedicated AMD Radeon RX 550 2 GiB GPU from PowerColor [63], mainly due to its low power consumption and compatibility with the open source *amdgpu* Linux kernel driver [54]. Our test machine is equipped with a single Samsung 980 PRO 1 TB PCIe 4.0 NVMe SSD [36], which contains both our root partition and our swap space. The system runs the Linux distribution Fedora 40 on kernel version 6.9.6, containing the modifications described in Section 4.3. We choose to perform our benchmarks while running a desktop environment, GNOME Shell [32], as we want to evaluate Suspend-to-CXL from a consumer perspective. The port of coreboot to the ASRock board is forked from the mainline coreboot project at commit `2de0e87`. We build upon this commit for our implementation of Suspend-to-CXL, as detailed in Section 4.4, as well. For conducting our hibernation experiments, we leave configuration options at their default settings, specifically hibernation image size, which is set to about $2/5$ of available system RAM [58], which equates to roughly 6 GiB. Also, we configure hibernation for OS-managed mode, as opposed to S4BIOS mode, for the reasons discussed in Section 2.3.3.

To conduct our measurements, we operate two external devices. The first device is a GPM-8213 power meter from GWINSTEK [10] to measure power draw and analyze energy consumption of our implementation. We place the power meter between our system's power supply and the power outlet to capture the active power draw in Watts of our machine. The second external device we employ is a PiKVM [6], a simple KVM device based on a Raspberry Pi mini-computer [34]. Although our server board features an ASPEED AST2600 Baseboard Management Controller [1], which supports basic KVM functionality such as video capture, USB input, and power button control, the provided shell lacks the automated scripting capabilities we require for our evaluation. For this reason, we use the aforementioned PiKVM, which runs a modified version of Arch Linux, thus allowing convenient script execution. The PiKVM provides access to its API via an HTTP interface, enabling shutting down, powering on, and monitoring VGA output from a script.

In our analysis, we opted to use the median and the Median Absolute Deviation (MAD) instead of the more common mean and standard deviation due to the nature of our data distribution. The mean and standard deviation are sensitive to outliers and skewed data, which can distort results, especially in cases of measurements not following normal distributions. In contrast, the median provides a more robust metric, as it represents the midpoint of the data, unaffected by extreme values. Similarly, the MAD offers a more reliable measure of dispersion for data with outliers or skewness, as it calculates variability based on the median rather than the mean. This choice ensures that our analysis remains representative of typical system behavior, minimizing the influence of anomalous data points and providing a clearer picture of performance characteristics.

5.1.1 Hibernation

When attempting hibernation on Linux with less than half of system memory available, such as in our case with a 12 GiB memory load, Linux is unable to create the hibernation image due to insufficient space in memory for copying pages for compression. To address this issue, we install a second, identical 16 GiB DIMM, increasing the system's total RAM to 32 GiB during our hibernation benchmarks. Since this modification only affects the initial page copying before compression, we expect its impact on the benchmark results, such as a small increase in power draw due to the additional DIMM, to be negligible. For all configuration options discussed below, we aim to replicate the original system configuration with 16 GiB of memory, as outlined in Section 5.1. As described in Section 2.3.3, one such configuration involves setting the `image_size` parameter, which defines the maximum memory footprint of the hibernation image. Preserving the base level of 2/5 of system RAM, i.e., roughly 6 GiB on our original test system, leads to unforeseen effects. Specifically, pages we allocate to fill memory to the target threshold are swapped out before hibernation, as they would not fit inside the image for high memory utilization. The process of swapping out user pages can be prevented by setting `sysctl.vm.swappiness=0` in the Linux command line parameters [55]. In order to receive a full picture of hibernate's capabilities, we conduct two hibernation benchmarks: one with the default setting of 2/5 of memory for `image_size`, allowing for processes to be swapped out, and one with `image_size` set to the entire system RAM capacity and prohibited swapping of processes via `sysctl.vm.swappiness=0`. We refer to these benchmarks as "hibernate-swap" and "hibernate-noswap", respectively, in the following. While forcing Linux to store the entire system memory in the hibernation image aligns more closely with the traditional concept of hibernation, allowing the system to swap out memory-intensive processes resembles our asynchronous resume strategy employed in Suspend-to-CXL. We show that allowing swap before hibernation results in lower resume latencies in Section 5.2. However, unlike our approach, swapping out processes before suspending leads to a significant number of page faults upon system resume, as user processes attempt to access their memory pages. These page faults, in turn, introduce substantial performance penalties, as the system must transfer the data back into memory. Although Suspend-to-CXL also moves pages to persistent storage, it avoids these page fault-related issues by allowing virtual memory mappings directly to CXL memory. Additionally, Linux allows configuring the compression algorithm for the hibernation image between LZO, LZ4, and no compression at all [55]. We choose to keep the default setting of LZO for our benchmarks, as this likely represents the situation most end users experience.

5.1.2 Benchmarking

We conduct benchmarks at system memory usages of $4\,\mathrm{GiB}$, $8\,\mathrm{GiB}$, and $12\,\mathrm{GiB}$. For each benchmark, we complete 100 successful suspend-resume cycles for both Suspend-to-CXL and hibernate. To fill system memory to the desired value, we fetch the current memory utilization by reading the `MemAvailable` line from `/proc/meminfo` via ssh and allocate accordingly by executing a basic C program containing calls to `malloc`. Measuring a single suspend-resume cycle consists of the following steps:

1. Ensure the machine is powered on, outputting video, and responding to SSH requests.
2. Fetch memory information and allocate to target usage.
3. Request target ACPI sleep state via SSH.
4. Wait for the system to stop outputting a video signal.
5. Wait for the system to indicate it is off via monitoring the system LEDs from the PiKVM's HTTP interface.
6. Power the machine back on immediately via the PiKVM's HTTP interface. (not necessary for Suspend-to-CXL)
7. Wait for the system to resume outputting a video signal.
8. Wait for asynchronous migrations to complete.
9. Collect power measurements and timing information.

To identify unsuccessful runs, time limits are introduced at each step. Most notably, we employ these timeouts when waiting for a specific state to be reached, for example when expecting the system to respond to pings over the network or waiting for video output to the PiKVM to stop or resume. As described in Section 2.3.3, hibernate initially loads a fresh instance of the Linux kernel, and performs sanity checks before overwriting itself with the hibernated data. In case these sanity checks fail, the fresh instance is simply kept, resulting in a functional, although not resumed operating system. As this behavior does not result in any externally noticeable failures, we additionally search the kernel log for messages indicating successful hibernation resume when benchmarking hibernate and fail if none are present.

5.1.3 Measuring Power Consumption

We collect power measurements from the GPM-8213 throughout the entire duration of the benchmark, specifically while suspending and resuming the system, and during the brief period the machine is sleeping. We use the power meter to monitor active power, measuring approximately three values per second. The GPM-8213 supports remote control via Ethernet, which we employ to gather data on power usage programmatically [9].

5.1.4 Measuring Time

Benchmarking sleep modes involves significantly different challenges compared to evaluating conventional user applications. Since the timings we need to capture originate from various components, including firmware, the operating system, and external events such as power button presses, our evaluation requires the use of multiple clocks to accurately measure these interactions. We count on three sources for measuring timings during our benchmarks, Linux on the PiKVM, coreboot on the test system, and Linux on the test system. From our script running on the PiKVM, we generate a high-resolution timestamp synchronized to Universal Time via the Network Time Protocol for every action we issue via the PiKVM's HTTP interface, such as waiting for video output or pressing the power button. For benchmarking the initialization code in coreboot, we make use of coreboot's capabilities to generate timestamps and save them to CBMEM. When timestamping is enabled, coreboot generates timestamps on entry and exit of every stage, as described in Section 2.2.1. To enable the creation of timestamps for our sleep implementation as well, we create the timestamp IDs `TS_S2CXL_START` and `TS_S2CXL_END`. On resume, we save the timestamps to CBMEM with a call to `timestamp_add_now` with the corresponding IDs as parameters at the start and end of our resume implementation, respectively. Although using the same mechanism for suspend would be ideal, the timestamps would be lost when the system is reset, as the CBMEM buffer is located in system memory. For this reason, we capture timestamps during suspend by reading the processor's Time Stamp Counter (TSC) register [20, Volume 3B, Section 18.7], and, instead of saving it to CBMEM, we save the value to our header in CXL memory, ensuring the timing value is not lost. Saving these timestamps involves writing only two values to CXL memory, a process that requires negligible time and does not impact our benchmark results. Then, on resume, we simply log the measured suspend timing values via `printk`, and collect them by reading the logs saved in CBMEM from Linux when the system has resumed. Lastly, obtaining certain timestamps from Linux on the test system is necessary. Although reading timestamps from the kernel ring buffer is feasible, timekeeping is halted during sleep, preventing accurate tracking of sleep duration on the test system. Therefore, we focus on gathering timestamps related to the asynchronous migration of pages back to system memory, as this process is initiated only after the system has resumed, ensuring it is not affected by the described timekeeping limitation. A selection of the timestamps and durations we collect is given in Table 5.1. In the following, we reference these times when applicable in order to identify comparable time frames between the different sleep modes we evaluate.

Name	**Description**
$t_{\text{suspend\_request}}$	Suspend request by PiKVM (step 3)
$t_{\text{video\_suspend}}$	Test system stops outputting video signal (step 4)
$t_{\text{power\_off}}$	Test system's ATX LED turns off (step 5)
$t_{\text{video\_resume}}$	Test system resumes outputting a video signal (step 7)
$t_{\text{kernel\_resume}}$	The test system logs that suspend is completed to ring buffer
$t_{\text{async\_complete}}$	The test system logs that all asynchronous migrations back to system memory are completed (step 8)
Δt_{async}	Duration of asynchronously migrating user pages back to system memory in Linux on resume

Table 5.1: Selection of timestamps and durations collected during each benchmark run and their corresponding benchmark steps according to Section 5.1.2.

5.2 Resume Latency

In Section 3.1 we identify resume latency as being the most important metric for system suspend. The subsequent section describes our measurements of resume latency and presents our findings.

For our implementation, two key time durations are of particular interest. The first duration measured extends from the moment the power button is pressed to the point when video output resumes, making the system usable for the end user. The second relevant duration is the time from pressing the power button until the asynchronous migrations are completed. The primary objective of our implementation is to minimize the first duration, as it defines the wake latency of the system, marking the point when the user can begin interacting with their machine.

At $t_{\text{power\_off}}$, when the power LED turns off, as indicated by the PiKVM, we immediately press the power button, allowing the resume latency to be calculated as $t_{\text{video\_resume}} - t_{\text{power\_off}}$. To further analyze resume times, we break down the resume latency by collecting timestamps in coreboot, as detailed in Section 5.1.4. For Suspend-to-CXL, coreboot logs are retrieved from CBMEM to measure the time spent restoring kernel data to system memory during the ramstage in coreboot. To assess the execution times of the various coreboot stages outlined in Section 2.2.1, as well as the overall duration of our Suspend-to-CXL code, CBMEM is accessed again to retrieve the saved timestamps. An overview of resume times is given in Figure 5.1. Resume latencies of Suspend-to-CXL and hibernate-swap remain nearly constant across varying memory usages, whereas resume durations for hibernate-noswap show a steady increase as memory utilization rises. On median, resume latencies for Suspend-to-CXL are at least 78 % lower than for hibernation, on average. Furthermore, the time required to migrate user memory back to system memory after resuming from Suspend-to-CXL also grows with higher memory utilization. The subsequent sections provide a more detailed analysis of our results, focusing on the constituent components of resume times.

5.2.1 Suspend-to-CXL

As shown in Figure 5.1, resume latencies of Suspend-to-CXL do not vary significantly relative to memory usage. This is expected, as the majority of memory is occupied by user processes, whose pages are migrated to CXL on suspend and only moved back asynchronously after the system has resumed. By collecting fine-grained timestamps, we are able to break down the resume process into its constituent durations, visualized in Figure 5.2.

Although not strictly a requirement for sleep modes identified in Section 3.1, a primary objective of our implementation of Suspend-to-CXL is to minimize the amount of memory copied synchronously on resume and to offload the bulk of the copy operations to our asynchronous component to be run when the system is already resumed and responsive to the user as described in Section 3.3.2. Moreover, we aim to keep the amount of data requiring synchronous copying constant, regardless of system memory usage. Suspend-to-CXL achieves these goals by first migrating memory of all user space processes to CXL memory as detailed in Section 4.3.2. As a result, our implementation maintains a low and consistent synchronous copy duration in coreboot during the resume phase, regardless of memory usage, as shown in Table 5.2.

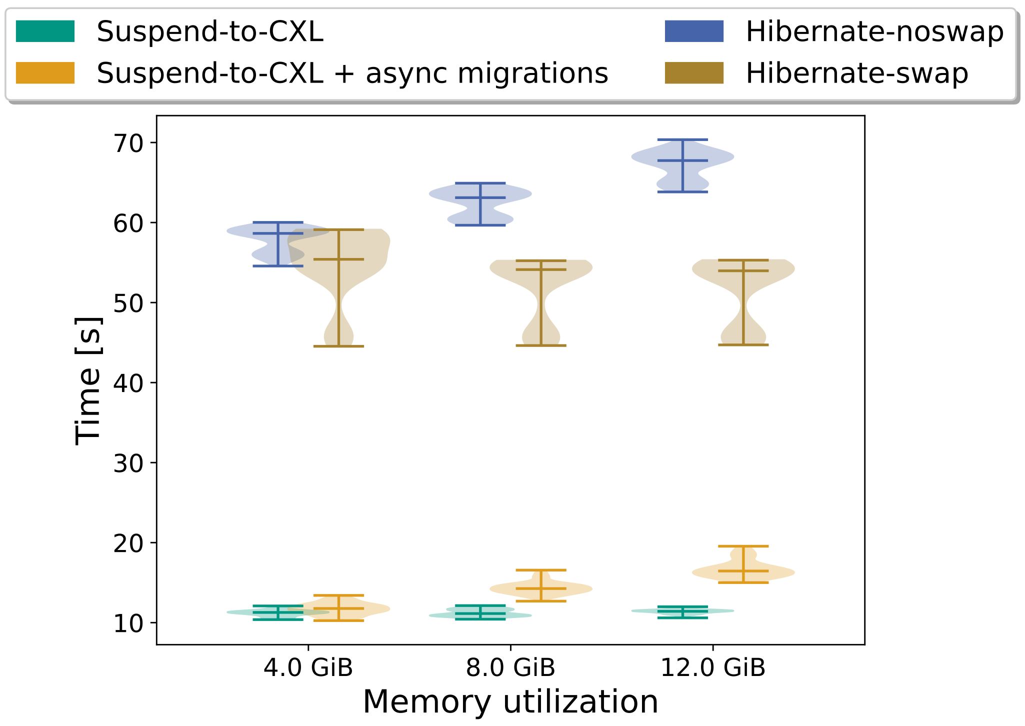

Figure 5.1: Violin plot of resume latencies encountered with Suspend-to-CXL and hibernate. The upper and lower ticks on each component mark the minimum and maximum durations measured, respectively. The ticks inside the components represent the median of resume durations.

Furthermore, Figure 5.2 demonstrates that resume times in general remain constant across varying memory usages. While latencies specifically introduced by Suspend-to-CXL code, namely “s2cxl init” and “s2cxl restore mem” in Figure 5.2, only make up about $4\,\%$ of the total resume duration, Figure 5.2 clearly illustrates that nearly half of the resume delay is caused before coreboot’s bootblock even commences. We refer to this time frame as “platform delay,” as it likely involves low-level platform initialization required for coreboot’s bootblock to be able to begin execution. Moreover, our coreboot component spends minimal time on data structure initialization and traversal, depicted as “s2cxl init” in Figure 5.2, with the majority of its time dedicated to copying memory back to system RAM, which is labeled “s2cxl restore mem” in Figure 5.2. “coreboot misc” encompasses all periods not clearly assignable to a specific stage, such as loading or decompressing the next stage. The use of `MOVDIR64B`, as described in Section 4.4.2, outperforms coreboot’s `memcpy` implementation during resume when moving data from the CXL device to host DDR memory by approximately $10\,\%$. While coreboot’s bootblock consistently completes in under 1 ms, the next stage, the romstage, dominates the time spent in coreboot on resume and accounts for approximately 25 % of Suspend-to-CXL's total resume latency. Due to the bootblock and postcar stages both being very short, we choose to omit them from Figure 5.2, as they would not be visually discernible and do not contribute to the total suspend latency in any meaningful way. The latencies introduced by resume code in Linux prove to be independent of memory usage as well. Our resume component in Linux in particular returns control back to the normal resume recovery procedure almost immediately, as it is solely responsible for registering our asynchronous migration function with the Linux workqueue system, as described in Section 3.3.2. We assert that this function call introduces a delay of under 1 ms, which is insignificant compared to the latency sources discussed earlier.

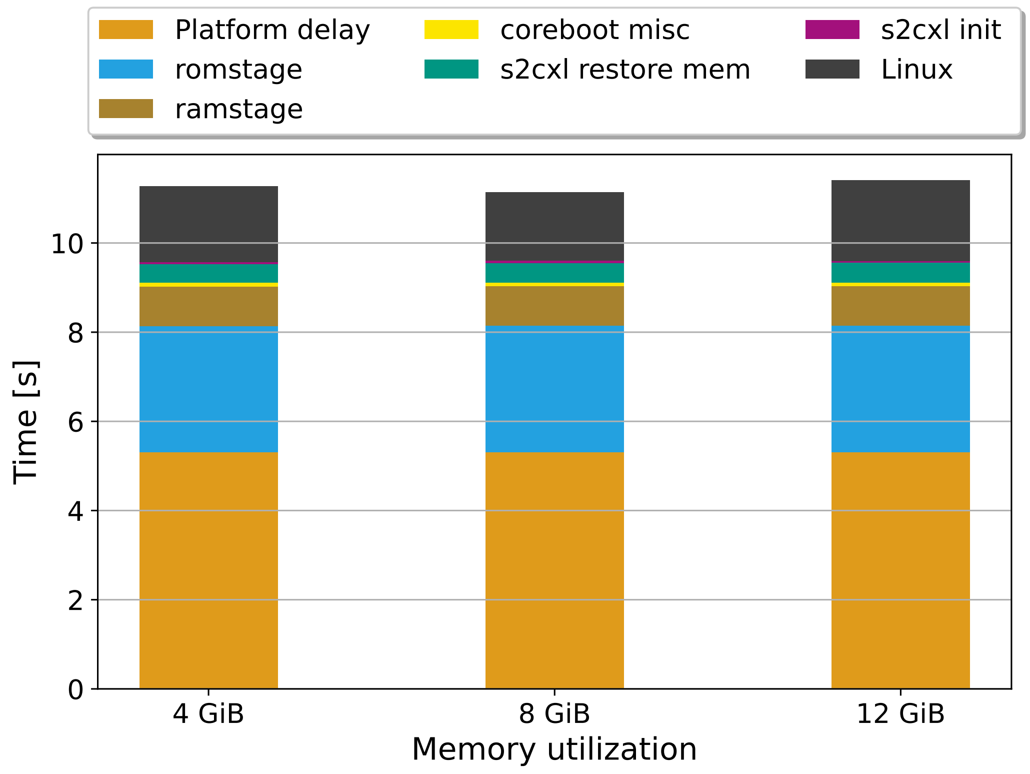

Figure 5.2: Breakdown of resume latency for Suspend-to-CXL at different memory usages. Each component is calculated as the median of 100 successful runs. Resume delays appear mostly independent of total system memory load.

	4 GiB	8 GiB	12 GiB
Copy amount [GiB]	1.485 ± 0.004	1.497 ± 0.006	1.501 ± 0.006
Copy duration [GiB]	0.421 ± 0.003	0.438 ± 0.010	0.441 ± 0.014

Table 5.2: Median and MAD of memory copied back to system memory in coreboot (in GiB), along with the median and MAD of the copy durations (in s) relative to system memory usage.

Asynchronous Migration

Although resume latency is of primary interest to us, we analyze the time until asynchronous migrations complete as well, as this event marks the time at which all applications are able to run without the performance impact induced by accessing CXL memory. As expected, the time from pressing the power button to asynchronous migrations concluding is dependent on the system memory utilization and, as seen in Figure 5.1, follows a similar linear trend to hibernate. The exact durations Linux is busy migrating back user pages in given in Table 5.3. Although three data points do not allow for a meaningful regression analysis, it is safe to assume that the duration of migrating pages back to system memory is linearly dependent on the system memory usage. Our timestamps further indicate that our asynchronous resume function's execution time is dominated by page migrations, while initialization and the priority analysis on processes only requires negligible time. This can be attributed to the simplicity of our priority analysis, which first retrieves and then sorts all user processes, as highlighted in Section 4.3.3.

Duration	4 GiB	8 GiB	12 GiB
Migration duration [s]	1.929 ± 0.038	4.424 ± 0.295	6.891 ± 0.575

Table 5.3: Median and MAD of Δt_{async} after resuming from Suspend-to-CXL (in s). The latency increases in accordance with system memory load.

5.2.2 Hibernation

Evaluating resume latencies for hibernation proved to be significantly more challenging than for Suspend-to-CXL. The reason for this problem mainly lies in the additional sources introducing delays. In particular, we observed that the FSP, as introduced in Section 2.2, resets the CPU of our test system during the romstage. This phenomenon occurs exclusively on cold boots, which is why these effects cannot be observed for Suspend-to-CXL, since it only performs a warm reset due to the limitations explained in Section 4.4.3. In order to be able to accurately account latencies to their sources, we introduce a new delay to our analysis, the "romstage reset delay," spanning from the initial power button press up to the point the FSP performs the aforementioned reset. Unfortunately, this type of CPU reset is very difficult to measure automatically for three reasons. First, the CPU is completely restarted, resulting in traditional timekeeping measures, such as the CPU TSC, mentioned in Section 5.1.4, also being cleared. Second, since this reset happens early in the romstage, system memory is not set up yet, resulting in severely limited means to measure and persist any data on timings. Third, the reset occurs in the FSP binary, which generates neither logs nor timestamps, making it impossible to pinpoint the exact timing of the reset with these methods alone. Toward resolving this problem, we settle on separately measuring the time between initial power button press and the bootblock starting for the second time after the reset with the serial logging capabilities described in Section 4.5. This approach allows the generation of timestamps for incoming log messages, enabling us to obtain an average total delay value. In combination with the platform delay reported by coreboot's timestamps after the system is booted, we are able to reconstruct the "romstage reset delay." Further details on the exact sequence and measurement of events can be found in Figure 5.3.

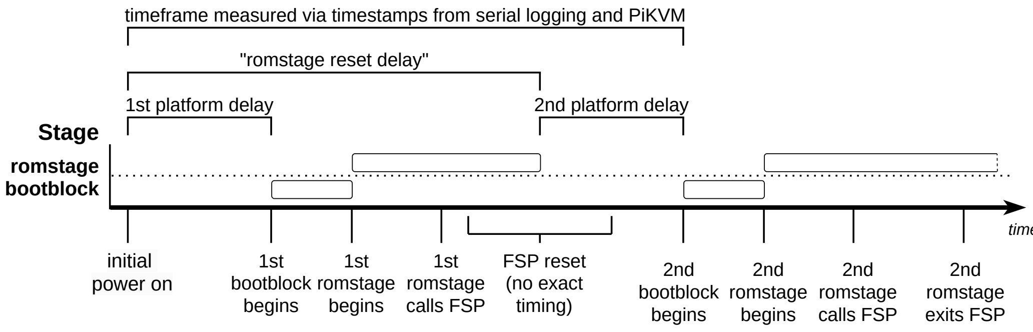

Figure 5.3: Chronology of the romstage reset encountered on our test system, including coreboot stages involved and measurable durations. The second platform delay, bootblock, and romstage durations are reported by the coreboot timestamp system and used in Figure 5.4.

Additionally, hibernation introduces EDK2, whose role is detailed in Section 2.2, as a further source of delay compared to Suspend-to-CXL. Since coreboot performs all platform initialization, EDK2 is mostly tasked with setting up UEFI features, such as identifying boot devices, providing a UEFI shell, and displaying a graphical boot menu. As a result, the execution time of the release build of EDK2 we use is dictated by a built-in 3-second timeout, allowing a user to press the necessary button on the keyboard to enter the boot menu, for example. We decide to assume the delay introduced by EDK2 to be exactly 3 s for our analysis for two reasons. First, capturing exact timestamps for the execution time of EDK2 would have required introduction of another, external time source, such as serial logging timestamps, since EDK2's release build does not expose timekeeping information like coreboot does, as described in Section 2.2.3. Second, although execution time of EDK2 may vary slightly, these variations do not contribute significantly to the resume times of hibernation.

Complementary to the last section, we showcase our results for resuming from hibernation in the following. As mentioned in Section 5.1, we conduct hibernation benchmarks with swapping turned on and off, in the following referred to as “hibernate-swap” and “hibernate-noswap”. Resume latencies for hibernate-swap and hibernate-noswap can both be found in Figure 5.4.

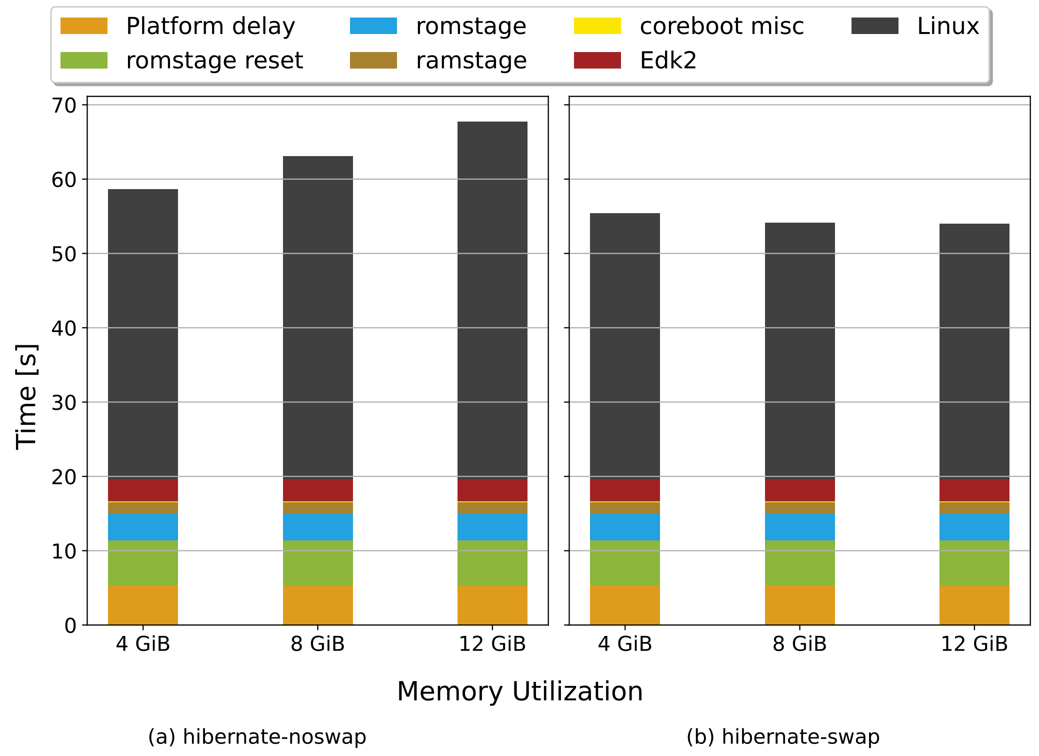

Figure 5.4: Breakdown of resume latency for hibernate-noswap and hibernate-swap, each component calculated as the median of 100 successful runs. Linux dominates resume times for hibernation in general. Allowing the system to swap out processes results in lower resume latencies, regardless of memory usage.

Figure 5.4 demonstrates that, across all memory utilization levels, resume latencies for hibernation are primarily driven by the time spent within the Linux kernel, reaching a maximum of 71 % of the total resume duration for hibernate-noswap at 12 GiB of memory usage. This result coincides with our expectations, as all resume operations are handled exclusively by the kernel without any firmware involvement. As outlined in Section 2.3.3, a new kernel image is initially loaded, which subsequently decompresses and restores the hibernation image. The time required for decompression and loading is directly related to the size of the image and, consequently, the amount of system memory in use at the time of suspending for hibernate-noswap. Allowing the system to swap out processes during hibernation, however, decouples resume performance from system memory usage, as the hibernation image size is constrained by the upper limit for `image_size` defined in Section 5.1. Although resuming the system is accelerated when process swapping is enabled, as can be seen in Figure 5.4, tasks that were swapped out suffer significant performance degradation upon resume due to the need to reload data from the backing storage into system memory on demand. In theory, this backing storage could be any persistent storage medium, which may be substantially slower than the NVMe SSD used in our test environment, resulting in an even greater impact on application performance.

A notable amount of resume latency is caused by the delays the test system experiences. In total, platform delay and romstage reset delay account for up to 21 % of the total resume latency of hibernation (hibernate-swap at 12 GiB load). Similar to Suspend-to-CXL, we choose to exclude the bootblock and postcar stages from Figure 5.4 due to their brevity. After the reset caused by the FSP described earlier, the romstage requires significantly more time to complete when resuming from hibernation compared to Suspend-to-CXL. The timestamps collected from coreboot indicate that this difference can be attributed to the FSP memory setup function call. We presume that more initialization is required to set up system DRAM when waking from hibernation, since the machine has undergone a cold reset during hibernation, whereas Suspend-to-CXL only performs a warm reset, as explained in Section 4.4.3. We expect this difference to disappear when using persistent CXL memory, allowing a complete power off of the system during Suspend-to-CXL sleep and thus resulting in identical romstage latencies for Suspend-to-CXL and hibernate. Figure 5.4 further reveals that resuming from hibernation introduces an added 0.7 s of latency in coreboot's ramstage compared to Suspend-to-CXL. This can be attributed to one of the advantages of our design discussed in Section 3.3.1, where the use of the ACPI wake vector enables a direct jump back to the operating system, thus bypassing the time-consuming payload decompression step.

Moreover, Figure 5.4 emphasizes that permitting the system to swap out processes benefits resume latency greatly. Not only are resume times up to $20\,\%$ lower, latencies stay constant, regardless of current system memory load. However, as mentioned previously, we expect these improved resume latencies caused by swapping out applications to come with the cost of considerable performance impairments for these applications after resuming, since pages must first be loaded from storage when initially accessed. Interestingly, while hibernate-swap with memory loads of $8\,\mathrm{GiB}$ and $12\,\mathrm{GiB}$ exhibits nearly identical performance, resume latency increases when hibernating with only $4\,\mathrm{GiB}$ of memory utilization. We infer that this occurs because Linux opts to compress most of the system memory into the hibernation image, as the threshold of approximately $6\,\mathrm{GiB}$ image size is not exceeded. This leads to hibernate-swap at $4\,\mathrm{GiB}$ exhibiting resume times similar to those of hibernate-noswap. Once the maximum image size would be surpassed, however, it appears that the higher memory usage forces Linux to swap pages more aggressively, resulting in smaller and more consistent image sizes. Additionally, allowing the system to swap out pages before hibernation results in significantly greater variability in resume times, as seen in Figure 5.1. We presume this can be traced back to Linux swapping out different amounts of memory between runs, thus leading to a larger variance in hibernation image sizes, despite identical total system memory load.

5.3 Suspend Latency

While not the primary focus, we also identify suspend latency as a relevant metric for evaluating our implementation in Section 3.1. Our methodology largely mirrors the approach used for measuring resume latency, as described in Section 5.2. Suspend latency is defined as the interval between initiating the suspend command via SSH and the machine signaling it is powered off through its power LEDs and can thus be written as $t_{\text{power\_off}} - t_{\text{suspend\_request}}$. For Suspend-to-CXL, we determine the duration of page migrations by using timestamps obtained from calls to `ktime_get` [56], which we obtain after finishing the benchmark from kernel logs via `dmesg`. For Linux, we differentiate between the time spent migrating pages in our code and the rest of the suspend procedure. “s2cxl init” represents the latency introduced by Suspend-to-CXL code in coreboot, excluding data copying, while “s2cxl save mem” depicts the timespan during which system memory is copied to CXL memory in coreboot.

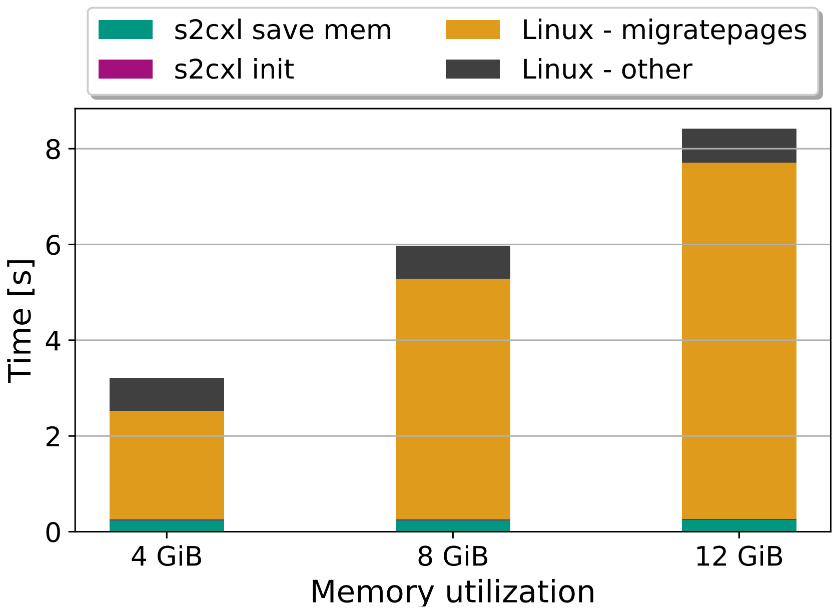

Figure 5.5: Breakdown of suspend latency for Suspend-to-CXL, each component calculated as the median of 100 successful runs. Suspend duration increases with rising memory load.

Sleep Mode	**4 GiB**	**8 GiB**	**12 GiB**
Suspend-to-CXL [s]	3.215 ± 0.035	5.971 ± 0.087	8.417 ± 0.060
Hibernate-noswap [s]	15.347 ± 0.183	20.034 ± 0.269	24.908 ± 0.331
Hibernate-swap [s]	14.665 ± 0.477	17.986 ± 0.167	20.365 ± 0.135

Table 5.4: Median and MAD of suspend durations (in s) for Suspend-to-CXL and hibernate for the different system memory loads used in our benchmarks.

In contrast to resuming, suspending the system with Suspend-to-CXL is strongly influenced by the system's current memory utilization, as shown in Figure 5.5. Suspend latency is primarily determined by user process page migrations, which require more time with growing memory usages. Meanwhile, the durations of other components, including the Linux overhead, which encompasses freezing user space and suspending devices, for example, and our suspend code in coreboot, remain constant, as visualized in Figure 5.5.

Notably, migrating pages back from CXL memory in Linux takes longer than migrating them to CXL memory, which is unexpected given that read operations are typically faster than write operations on most conventional types of storage. At the same time, these results are similar to the findings of Sun et al. [43],

	4 GiB	**8 GiB**	**12 GiB**
Copy amount [GiB]	1.485 ± 0.004	1.497 ± 0.006	1.500 ± 0.006
Copy duration [s]	0.244 ± 0.003	0.245 ± 0.003	0.256 ± 0.008

Table 5.5: Median and MAD of memory (in GiB) copied from system memory to CXL memory in coreboot, along with the median and median absolute deviation of the copy durations (in s).

who report higher bandwidth when transferring data from system DRAM to CXL memory compared to the other way around. Due to time constraints, we were unable to further explore this discrepancy. Regardless, Suspend-to-CXL consistently achieves lower suspend latency compared to hibernation. When resuming from Suspend-to-CXL, using our `memcpy` implementation employing `MOVDIR64B` resulted in shorter copy times when moving data from the CXL device to system memory, as mentioned in Section 5.2. On suspend, the difference between coreboot's `memcpy` and `s2cxl_memcpy` becomes oven more noticeable. Table 5.5 shows that we were able to cut our copy time nearly in half by utilizing `MOVIDR64B`, compared to on resume (Table 5.2). We did not further investigate whether this behavior is expected when using CXL or if it is a property of our device. Similar to resume, we achieve approximately constant copy durations in coreboot across system memory loads, which is expected since the copy amount remains constant as detailed in Table 5.5.

As outlined in Table 5.4, suspend durations for hibernate-swap are at least 4 % shorter than those for hibernate-noswap, though the difference between the two is not as substantial as during resume. This is likely due to the reduced effort needed to compress a smaller amount of memory into the hibernation image. However, it is important to note that hibernate-swap loses its advantage of being independent of system memory usage, as observed during the resume process in Section 5.2. Both hibernation variants and Suspend-to-CXL thus exhibit increasing suspend times as memory load increases.

5.4 Reliability

A crucial metric to determine the usability of sleep mechanisms is their reliability. As argued in Section 3.1, the usability of a system unable to resume from sleep is worse than the usability when unable to sleep entirely. The test system may fail to resume for a variety of reasons, including failures while copying data on suspend, data corruption while the system is sleeping, and failures while restoring data back to system memory.

To measure reliability, we conduct 100 successful benchmark runs for both Suspend-to-CXL and hibernate. To calculate our metric, we divide the number of successful runs by the total number of attempts needed to achieve them. As outlined in Section 4.4.3, our test system is unable to fully power off during sleep, which necessitates the use of a warm reset. A consequence of this reset method is that system memory is retained, even after resetting. This could result in a significant overestimation of Suspend-to-CXL reliability, as the system could theoretically resume successfully without our implementation performing any operations, simply because the system memory remains unchanged from the point of suspending. To address this, modifications to the coreboot code were required for the reliability benchmark. Specifically, to simulate conditions akin to cutting power to system memory, the entire usable memory is zeroed after its contents are copied to the CXL device during suspend. The results for reliability are given in Table 5.6.

Mode	**4 GiB**	**8 GiB**	**12 GiB**
S2CXL	99.01 % (100 / 101)	98.04 % (100 / 102)	97.09 % (100 / 103)
Hibernate	69.93 % (100 / 143)	47.17 % (100 / 212)	48.54 % (100 / 206)

Table 5.6: Reliability of Suspend-to-CXL and hibernate on Linux. We observed no difference between different `image_size` settings, which is why we chose to only evaluate reliability for hibernate-noswap. The hibernation reliability we observe is likely not representative of hibernation in general.

Our implementation of Suspend-to-CXL endured 1 to 3 failures every 100 successful suspend-resume cycles. This indicates that there are no substantial differences in sleeping at different memory usages in terms of reliability. The logs of our benchmark reveal two distinct types of failures of Suspend-to-CXL. In the first scenario, the system fails to transition to the ACPI wake vector from Linux due to improper saving or restoration of memory at that location, causing the system to hang and the benchmark to time out while awaiting the resumption of video output. The second issue encountered pertains to the system successfully resuming execution in Linux; however, the file system recovery process fails, resulting in the file system being left in a read-only state. This scenario is classified as a failure, as it would likely require a system reboot for the user to restore full functionality and perform productive tasks.

For hibernation, we anticipated that most, if not all, runs would complete successfully, as the resume process is entirely managed by Linux without any firmware participation. However, our results suggest the opposite, though we believe these findings do not accurately reflect the reliability of hibernate. Upon analyzing the benchmark logs, we determined that all failures are caused by the sanity checks outlined in Section 2.3.3 failing. Specifically, on resume, the firmware often configures the system memory map in a slightly different manner, which is then stored in the E820 table, leading Linux to abort the resume process on almost every second attempt. Due to time constraints, and as this issue is not directly related to our proposal, we did not further investigate whether this behavior is caused by coreboot or the FSP.

5.5 Energy Consumption

As described in Chapter 1, a sleep implementation must balance wake latency with energy conservation. For this reason, the last requirement we identify in Section 3.1 is energy consumption. For analyzing energy consumption, we measure the power draw of our system with our GPM-8213 power meter as described in Section 5.1.3. We define the base energy consumption of each sleep mode as the sum of the energy it takes to suspend the machine and the amount of energy required to restore video output on resume. The suspend duration is defined as the timespan between requesting suspend and the system indicating power off, while the resume duration encompasses the time from powering on the machine to video output resuming, analogous to the previous sections. Furthermore, we continue measuring the power draw of our test machine for Suspend-to-CXL until all asynchronous migrations are completed. Due to the wider variation in timings for hibernate-swap, as illustrated in Figure 5.1, power consumption also exhibited greater fluctuations, making it challenging to achieve reproducible results. Additionally, the energy consumption of hibernation is mostly dominated by the boot process, leading to similar results for hibernate-swap and hibernate-noswap in terms of energy consumption. For these reasons, we chose to solely focus on hibernate-noswap in this section. Our measurements for Suspend-to-CXL and hibernate are displayed in Figure 5.6. We adjust the $0\,\mathrm{s}$ mark to align with $t_{\text{suspend\_request}}$.

The power draw over time follows a similar pattern for both Suspend-to-CXL and hibernate, with power consumption dropping briefly to approximately $0.12\,\mathrm{W}$ when the system is powered off for hibernate. Since the test system is not placed in S5 for Suspend-to-CXL, as detailed in Section 4.4.3, the power draw stays at a high level of approximately $110\,\mathrm{W}$. When using persistent CXL hardware, we expect the power draw to drop to the same $0.12\,\mathrm{W}$. Subsequently, power consumption rises sharply to approximately $150\,\mathrm{W}$, peaks at $180\,\mathrm{W}$, and then begins to fluctuate. While power measurements were fairly consistent for hibernate, the power draw of the Linux suspend component of our implementation seems to exhibit three distinct levels $161\,\mathrm{W}$, $167\,\mathrm{W}$, and $171\,\mathrm{W}$. While we suspect

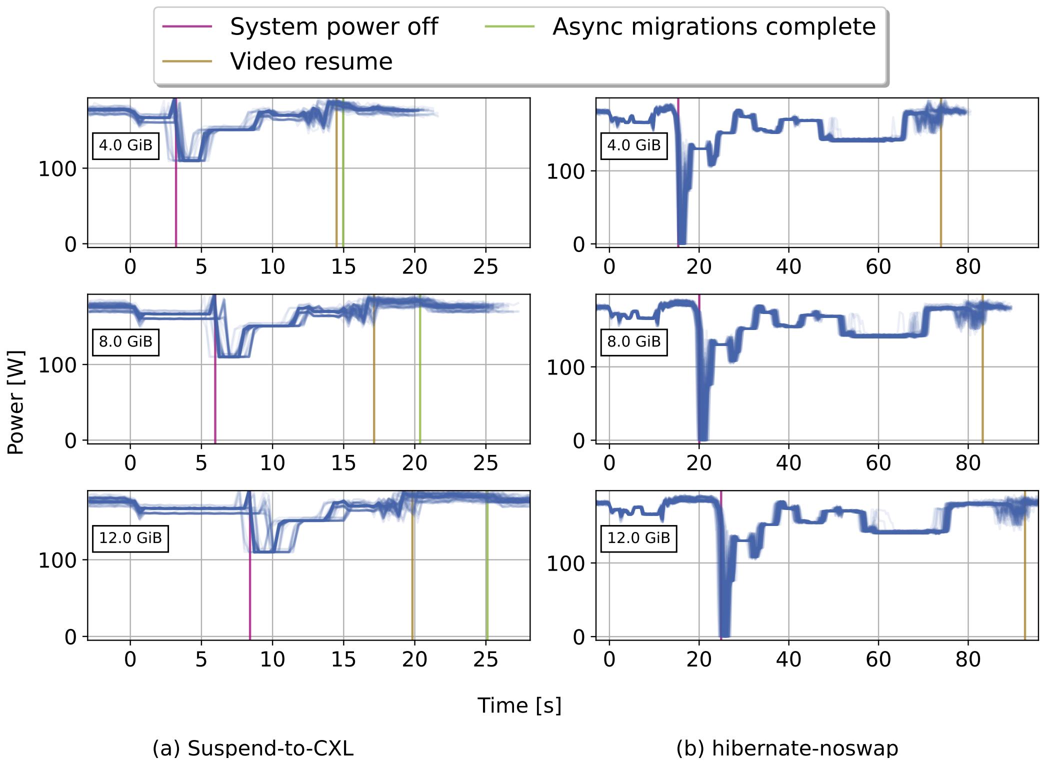

Figure 5.6: Power draw over time of 100 successful runs of Suspend-to-CXL and hibernate for different memory usages (in W), overlapped. The time axis is aligned so that $0\ s$ coincides with the remotely issued sleep request for each run.

this phenomenon can be attributed to operating system power management, we did not have time to further investigate this assumption. From these power measurements, we additionally calculate the total energy consumed to suspend and resume the system for Suspend-to-CXL as well as for hibernate, found in Table 5.7 and Table 5.8, respectively. We arrive at these values by integrating the curve derived from our power measurements over the respective intervals.

As anticipated based on the results from Section 5.2, the energy consumption for Suspend-to-CXL during the suspend phase is highly dependent on memory usage, whereas the energy consumption during the resume phase remains largely constant. Conversely, for hibernate, the energy consumption for both suspend and resume phases is significantly influenced by the system's memory load. Regarding energy consumption during suspend, Suspend-to-CXL outperforms hibernate, reducing expended energy by approximately a factor of 5.6 with $4\,\mathrm{GiB}$

Stage	4 GiB	8 GiB	12 GiB
Suspend [J]	472 ± 3	938 ± 16	1350 ± 7
Resume [J]	1686 ± 58	1680 ± 65	1694 ± 32

Table 5.7: Median and MAD of consumed energy for Suspend-to-CXL (in J). "Suspend" is the energy consumed from requesting suspend to the system being powered off. "Resume" is from system power on to video output resuming. The values are calculated from our 100 successful test runs.

Stage	4 GiB	8 GiB	12 GiB
Suspend [J]	2602 ± 41	3461 ± 42	4351 ± 39
Resume [J]	8722 ± 129	9552 ± 166	10401 ± 161

Table 5.8: Median and MAD of consumed energy for hibernate (in J). "Suspend" is the energy consumed from requesting suspend to the system being powered off. "Resume" is from system power on to video output resuming. The values are calculated from our 100 successful test runs.

of memory usage and by a factor of 3 with 12 GiB of memory usage. An even more pronounced disparity can be observed during the resume phase, where our implementation of Suspend-to-CXL demonstrates an energy consumption that is $6\times$ lower than that of hibernate-noswap with 12 GiB of memory in use.

Despite not being noticeable in our benchmarks, another noteworthy observation influencing energy consumption during extended sleep durations is the increase in power consumption of our test system to approximately 12 W after remaining in the S5 state for around one second. This relatively high power draw, despite the system being powered off, is likely attributed to the Baseboard Management Controller, which remains active in the S5 state to allow remote system startup by an administrator.

5.5.1 Asynchronous Resume

In addition to the results on energy consumption presented in the previous section, we further examine the energy required for asynchronously migrating pages back to system memory for Suspend-to-CXL. To achieve this, we perform an additional 100 modified Suspend-to-CXL runs, ensuring the system is idle and has fully completed the resume process when migrating pages back to system memory. This is accomplished by introducing a 10-second delay in our Suspend-to-CXL resume hook, as detailed in Section 4.3.3, before scheduling the asynchronous resume function for execution. In order to assess energy consumption of the asynchronous migration procedure, we require the baseline idle energy consumption our test system would consume in the same time frame. Toward obtaining this value, we measure a median idle power draw of 136 W with our power meter and calculate the energy consumed by multiplying the idle power draw with each migration duration. Information on the energy consumed during migration can be seen in Table 5.9.

Activity	4 GiB	8 GiB	12 GiB
Idle [J]	287	636	958
Migrating [J]	315	770	1220
Difference (absolute) [J]	28	134	262
Difference (relative)	9.8 %	21.1 %	27.4 %

Table 5.9: Median and MAD of consumed energy during idle and when working on asynchronous migrations (in J). We measure the idle power draw to be 136 W. The difference between “idle” and “migrating” is given absolutely and relatively.

As expected, Table 5.9 shows that the energy required to migrate pages back to system memory depends on system memory load. During the asynchronous migration phase, our test system consumes at least 9.8 % more energy compared to the base idle energy consumption. Even when factoring in the energy cost of asynchronous migrations, Suspend-to-CXL consistently demonstrates superior energy efficiency compared to hibernation at all memory load levels. In particular, Suspend-to-CXL consumes at least 3 × less energy than hibernation during system resume from sleep.

5.6 Discussion

After evaluating our implementation, we present key takeaway points concerned with the performance of Suspend-to-CXL and competing sleep mechanisms. Although the contrast with hibernate on Linux enables a fundamental assessment of the performance aspects of our implementation, a comparison with a sleep mode with similar latency characteristics, such as ACPI S3, would have been valuable to fully identify the performance detriment of offloading data to CXL memory. As explained at the beginning of this section, a comparison with S3 was unfortunately not possible due to the low-level code of the FSP being incapable of recovering from S3 on the platform of our test system. As this shortcoming results from a failure in proprietary code, we regrettably had no direct way of fixing this issue. Nonetheless, the benchmark results for each of the requirements outlined in Section 3.1 provide valuable insights into the characteristics of our sleep implementation.

Among the evaluated metrics, resume latency stands out as the most critical factor for end-user sleep mode performance. In our test setup, Suspend-to-CXL significantly outperforms hibernate in terms of resume latency. While hibernation exhibits median resume times ranging from $54\,\mathrm{s}$ to $68\,\mathrm{s}$, the resume duration for Suspend-to-CXL remains consistently at approximately $11.3\,\mathrm{s}$, regardless of memory usage, observable in Figure 5.4 and Figure 5.2. As mentioned in Section 5.2, nearly half of the total resume time is due to the platform delay occurring before the execution of coreboot's bootblock begins. Based on our experience, this delay is considerably shorter on consumer platforms compared to the server platform used in our benchmarks, suggesting the potential for faster resume times for desktop and laptop users. Although this approximately fivefold reduction in resume latency appears massive, it is important to consider that our test system lacks persistent storage and instead profits from the increased latency and bandwidth of the DDR4 DIMM on our FPGA. Consequently, Suspend-to-CXL likely demonstrates better performance in our controlled environment than it would in a real-world scenario with devices such as Samsung's CMM-H. Additionally, the warm reset employed in our setup significantly reduces latency during the coreboot romstage, further enhancing the observed performance. This advantage is also unlikely to transfer to real-world deployments, where the system is fully powered down during sleep. Nevertheless, we anticipate that Suspend-to-CXL will continue to outperform hibernate even when using persistent CXL storage, as our approach bypasses EDK2 and the entire Linux boot process during resume by leveraging the ACPI wake vector. Moreover, the asynchronous resume component of our implementation is expected to continue improving resume latencies, particularly when utilizing persistent CXL storage, as it avoids the extended read times associated with the sequential approach of hibernate described in Section 2.3.3. Since resuming from Suspend-to-CXL follows the same procedure as ACPI S3, except for the initialization and memory copying functionality we add to coreboot and the enqueueing of the asynchronous migration task in Linux, we expect resume latencies to be very similar to S3.

For suspend latency, Figure 5.5 shows that migrating pages introduces considerable delays for Suspend-to-CXL, especially for high memory usage, where page migration to CXL memory makes up $88.5\,\%$ of the total suspend time. Nonetheless, Suspend-to-CXL outperforms hibernate by a factor of 2 to 5 on suspend duration. While providing moderate performance benefits on resume, Section 5.3 details that using our own `MOVDIR64B`-based `memcpy` implementation cut copy time in coreboot nearly in half on suspend, compared to resume.

Suspend-to-CXL demonstrates reasonable reliability in our test environment, experiencing only 6 failed suspend-resume cycles out of 306 runs. In terms of reliability, it would have been desirable to compare our implementation to ACPI S3, as hibernate's reliability is theoretically expected to approach $100\,\%$, despite our benchmarks suggesting otherwise, making it a suboptimal competitor. Notably, the failure rates of Suspend-to-CXL do not appear to be influenced by system memory usage, which is expected as the amount of memory required by the kernel during resume remains constant across different memory loads. While the evaluation of our other requirements defined in Section 3.1 are impacted in some way by the fact that we do not have access to real persistent CXL storage, the values we determine for reliability are likely not dependent on the concrete device we use as a storage backend.

Our benchmarks establish that substantial amounts of energy can be saved by using Suspend-to-CXL over hibernation. As both implementations fully power off the machine in a real-world setting, and thus draw equal power when sleeping, it is sufficient to solely consider energy consumption on suspend and resume. For example, Suspend-to-CXL on our test system achieves a $3.5\times$ lower energy consumption in total with $12\,\mathrm{GiB}$ of memory used, compared to hibernate-noswap. Despite our benchmarks indicating the potential for significant energy savings, it is important to reiterate that our test setup is not equipped with persistent CXL storage and predicting energy consumption in particular without access to real hardware is difficult. Specifically, reading and writing data to DDR4 DRAM is significantly faster and more energy-efficient than writing to persistent storage. Simultaneously though, implementations of persistent CXL storage such as Samsung's CMM-H may consume less power in general than the FPGA we use for providing the CXL interface for accessing DRAM. In addition to the idle power consumption including the FPGA reported in Section 5.5.1, we conduct additional measurements of the power draw of our test system without the FPGA. From the measured value of $83\,\mathrm{W}$, we infer that the FPGA consumes approximately $53\,\mathrm{W}$, which significantly exceeds the typical power consumption of conventional NVMe SSDs, such as our Samsung 980 PRO, which consumes no more than $9\,\mathrm{W}$ [36].

Chapter 6

Future Work

After giving a detailed insight into the performance characteristics of Suspend-to-CXL in the previous chapter, we describe ideas for possible future enhancements of our implementation in the following.

Commercial CXL Hardware

As discussed throughout this thesis, the implementation and evaluation of Suspend-to-CXL were conducted using a CXL-capable FPGA equipped with a DDR4 DIMM, necessitated by the unavailability of commercially available persistent CXL storage solutions, such as Samsung's CMM-H. This constraint implies that the benchmark results obtained from our test system are not directly comparable to those of other sleep modes, such as Suspend-to-disk, which operate under real-world conditions. As CXL hardware becomes available in the future, an evaluation based on this hardware will prove invaluable to properly assess the characteristics of Suspend-to-CXL.

Priority Analysis

For analyzing the priority of processes to restore to system memory on resume, we identify processes that have recently run on the CPU. While this metric results in interactive applications being migrated back to system memory earlier, it is a very rudimentary algorithm and does not take into account any other metric except the last time the process ran on a CPU. For this reason, a more sophisticated policy involving various thread and process characteristics could be employed to further improve the user experience directly after resume.

Furthermore, a possible enhancement of our method could enable memory migrations at a page-level granularity. This could allow for the identification and subsequent transfer of frequently accessed ("hot") pages back to system memory, irrespective of the process to which they belong. By prioritizing the migration of hot pages, this strategy has the potential to enhance user-perceived performance, as time is not spent moving less frequently accessed ("cold") pages of prioritized applications.

Kernel Page Filtering

As detailed in Table 5.2, roughly $1.5\,\mathrm{GiB}$ of memory are transferred to and from CXL memory in coreboot, regardless of system memory usage. While this value for the identified minimum amount of memory the kernel requires to resume remains constant across system memory loads, it is greater than we initially expected. Aside from identifying used pages according to the Linux page allocator described in Section 3.2.1, we do not employ additional filtering techniques to further reduce the amount of pages to be copied in coreboot. We suspect that this load of $1.5\,\mathrm{GiB}$ can further be reduced, though we did not have time to accomplish this in the context of this thesis.

Accelerator-aided Page Migration

Although not supported on consumer platforms at the moment, server and workstation platforms have begun to support on-die accelerators, such as Intel's Data Streaming Accelerator [15] (DSA), which can be found on their Sapphire Rapids CPUs. In theory, an Intel DSA can be used to perform the page migration component of our sleep algorithm. Sun et al. [43] demonstrate that copying memory in this manner not only permits a reduced CPU load but also an up to $6\times$ improved throughput compared to performing copy operations on the CPU. The page migration mechanism we leverage inside the Linux kernel as described in Section 4.3.2 fundamentally supports different migration modes, including `MIGRATE_SYNC_NO_COPY`, which "will not copy pages with the CPU. Instead, page copy happens outside the `migratepage()` callback and is likely using a DMA engine." [61, `include/linux/migrate_mode.h`] Despite this low-level capability, the high level interface of `do_migrate_pages` we utilize only supports synchronous migrations performed by the CPU. A future implementation could expand the interface to different migration modes and leverage the potential of Intel's Data Streaming Accelerator to achieve page migrations with less CPU overhead.

Comparison with S3

Suspend-to-CXL is designed to directly compete with ACPI S3 on resume latency and energy conservation. However, as mentioned throughout Chapter 5, it was not possible for us to compare Suspend-to-CXL to ACPI S3 on our test system due to the feature not being supported by the FSP on this platform. We are optimistic that as CXL becomes more widely adopted in the future, an evaluation of Suspend-to-CXL on a platform still supporting ACPI S3 will be possible.

Chapter 7

Conclusion

In this thesis we presented Suspend-to-CXL, a novel sleep mode that capitalizes on CXL, a high-speed interconnect built upon PCIe that is able to provide persistent, byte-addressable storage for conventional computer systems. The design of Suspend-to-CXL relies on components in both firmware and the operating system for its sleep procedure. CXL supports exposure of byte-addressable memory to the host system, which allows regular memory pages to be mapped directly to it. Suspend-to-CXL leverages this capability by implementing an asynchronous resume strategy. This approach selectively restores only the essential operating system data needed to resume the system, followed by the retrieval of application data after the system is online, thus accomplishing the primary goal of minimizing resume delays. Further goals we strived to achieve are low suspend latencies, high reliability, and low energy consumption. Apart from the theoretical design of our sleep mechanism we also supplied a practical implementation of Suspend-to-CXL utilizing the open-source firmware coreboot for creating the firmware component and built the operating system component for Linux. We constructed our suspend mode for Intel's Eagle Stream server platform, which supports CXL. Suspend-to-CXL is designed to be compatible with other upcoming CXL-capable x86 platforms, as well. As a storage backend, we equipped our machine with an FPGA containing a DIMM of DDR4 memory, which is exposed as CXL memory to the system. Before suspending the system, we move the pages of user applications to CXL memory and identify the physical memory regions used by the Linux kernel. We modified the suspend code path in coreboot, ensuring execution of our own functionality to move the kernel memory regions to CXL memory as well, subsequently powering off the system. When the system is powered on again, Suspend-to-CXL ensures low resume times by only moving strictly necessary kernel data structures back to system memory and subsequently jumps back to the operating system without having to complete a normal boot cycle. Lastly, Suspend-to-CXL begins asynchronously migrating back processes while they are running on CXL memory.

We evaluated our implementation in terms of the aforementioned goals and compared the results to Linux's implementation of Suspend-to-Disk, called hibernate. We measured a resume latency of approximately $11\,\mathrm{s}$ for Suspend-to-CXL. Due to the asynchronous resume scheme leveraged by our sleep implementation, this latency remains independent of system memory usage. This behavior distinguishes Suspend-to-CXL from conventional persistent sleep modes using a single image containing the entire system memory, such as hibernate, which performs worse as memory loads grow. In comparison, our analysis reveals that resuming from hibernation is approximately $5\times$ slower, and consumes significantly more energy. Although our benchmarks are no replacement for an evaluation on commercial persistent CXL storage, we prove our implementation is capable of suspending and resuming a system reliably and with vastly smaller latencies compared to hibernation. In conclusion, Suspend-to-CXL lays the groundwork for efficiently suspending a system to CXL memory, offering very fast resume durations in particular due to the asynchronous migration strategy employed by our implementation.

Bibliography

[1] ASRock Rack Inc. ASRock SPC741D8-2L2T/BCM mainboard. https://www.asrockrack.com/general/productdetail.asp?Model=SPC741D8-2L2T/BCM, Accessed: 2024-09-01.

[2] Jacob Beningo. *Concepts for Developing Portable Firmware.* Apress, Berkeley, CA, 2017. https://doi.org/10.1007/978-1-4842-3297-2_1.

[3] CLEAResult Consulting Inc. What is 80 PLUS certification program? https://www.clearesult.com/80plus/program-details, Accessed: 2024-09-04.

[4] Yann Collet. LZ4 - Extremely fast compression, 2011. https://lz4.org/, Accessed: 2024-09-21.

[5] Compute Express Link Consortium Inc. *Compute Express Link ™ Specification 3.1.* ©2019-2023 COMPUTE EXPRESS LINK CONSORTIUM, INC. ALL RIGHTS RESERVED, August 2023.

[6] Maxim Deveav. PiKVM - Open and inexpensive DIY IP-KVM on Raspberry Pi. https://pikvm.org/, Accessed: 2024-09-26.

[7] Shawn Embleton, Sherri Sparks, and Cliff Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, New York, NY, USA, 2008. Association for Computing Machinery. https://doi.org/10.1145/1460877.1460892.

[8] flashrom project. flashrom documentation. https://www.flashrom.org/, Accessed: 2024-09-04.

[9] Good Will Instrument Co. Ltd. GPM-8213 Digital Power Meter - User Manual. https://www.gwinstek.com/en-US/products/downloadSeriesDownNew/11551/1553, Accessed: 2024-09-06.

[10] Good Will Instruments Co., Ltd. GPM-8213 Digital Power Meter. https://www.gwinstek.com/en-US/products/detail/GPM-8213, Accessed: 2024-09-05.

[11] Tejun Heo and Florian Mickler. Workqueue - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/core-api/workqueue.html, Accessed: 2024-09-14.

[12] Tejun Heo and Florian Mickler. NUMA Memory Policy, September 2010. https://www.kernel.org/doc/html/latest/admin-guide/mm/numa_memory_policy.html, Accessed: 2024-09-05.

[13] Chien-Chung Ho, Sheng-Wei Cheng, Yuan-Hao Chang, Yu-Ming Chang, Sheng-Yen Hong, and Che-Wei Chang. Efficient hibernation resuming with classification-based prefetching scheme for embedded computing systems. *SIGAPP Appl. Comput. Rev.*, 15(1):33–43, March 2015. https://doi.org/10.1145/2753060.2753064.

[14] Intel Corporation. Intel Firmware Support Package. https://github.com/intel/fsp/, Accessed: 2024-09-04.

[15] Intel Corporation. Intel® Data Streaming Accelerator (Intel® DSA). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html, Accessed: 2024-09-08.

[16] Intel Corporation. Intel® Optane™ Memory - Responsive Memory, Accelerated Performance. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html, Accessed: 2024-09-17.

[17] Intel Corporation. Intel® Xeon® Silver 4410Y Processor. https://ark.intel.com/content/www/us/en/ark/products/232376/intel-xeon-silver-4410y-processor-30m-cache-2-00-ghz.html, Accessed: 2024-09-19.

[18] Intel Corporation. TianoCore EDK II. https://www.tianocore.org/, Accessed: 2024-09-01.

[19] Intel Corporation. *Technical Overview Of The 4th Gen Intel® Xeon® Scalable processor family*, July 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html.

[20] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, July 2024.

[21] Kingston Technology Corporation. KSM48R40BS8KMM-16HMR Memory Module Specifications. `https://www.kingston.com/datasheets/KSM48R40BS8KMM-16HMR.pdf`, Accessed: 2024-09-23.

[22] Andi Kleen and SUSE Labs. numactl - NUMA support for Linux. `https://github.com/numactl/numactl`, Accessed: 2024-09-12.

[23] Shi-wu Lo, Wei-shiuan Tsai, Jeng-gang Lin, and Guan-shiung Cheng. Swap-before-hibernate: a time efficient method to suspend an OS to a flash drive. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 201–205, New York, NY, USA, 2010. Association for Computing Machinery. `https://doi.org/10.1145/1774088.1774129`.

[24] Fabian Meyer. Fast and Power-Efficient System Suspend Using Persistent Memory. Bachelor's thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October 11 2023.

[25] Fabian Meyer and Yussuf Khalil. 82203: mb/asrock: Add SPR 1S server board ASRock Rack SPC741D8-2L2T/BCM. `https://review.coreboot.org/c/coreboot/+/82203`, Accessed: 2024-09-27.

[26] Microsoft Corporation. Modern Standby - Microsoft Learn. `https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/modern-standby`, Accessed: 2024-09-01.

[27] Patrick Mochel and Mike Murphy. sysfs - The filesystem for exporting kernel objects. `https://www.kernel.org/doc/html/latest/filesystems/sysfs.html`, Accessed: 2024-10-07.

[28] Markus F.X.J. Oberhumer. LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`, Accessed: 2024-09-07.

[29] OSDev Wiki. Detecting Memory (x86). `https://wiki.osdev.org/Detecting_Memory_(x86)`, Accessed: 2024-10-02.

[30] David A Pattersson and John L Hennessy. *Computer organization and design RISC-V Edition: The hardware/software interface.* Morgan Kaufmann, 2020.

[31] pico-serprog developers. pico-serprog - Flashrom/serprog compatible firmware for the Raspberry Pi Pico. `https://github.com/stacksmashing/pico-serprog/`, Accessed: 2024-09-04.

[32] The GNOME Project. GNOME Shell, next generation desktop shell. `https://wiki.gnome.org/Projects/GnomeShell`, Accessed: 2024-09-28.

[33] Raspberry Pi Foundation. Pico-series Microcontrollers - Raspberry Pi Documentation. `https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html`, Accessed: 2024-09-26.

[34] Raspberry Pi Foundation. Raspberry Pi computers and microcontrollers. `https://www.raspberrypi.com/`, Accessed: 2024-09-04.

[35] Red Hat Inc. Automatic NUMA Balancing. `https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing`, Accessed: 2024-09-13.

[36] Samsung Electronics America, Inc. Samsung 980 PRO w/ Heatsink PCIe®4.0 NVMe®SSD 1TB. `https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-w-heatsink-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0cw/#specs`, Accessed: 2024-09-04.

[37] Kanoj Sarcar. What is NUMA? `https://www.kernel.org/doc/html/latest/mm/numa.html`, Accessed: 2024-09-03.

[38] Sea Sonic. Seasonic PRIME TX - 650 W power supply. `https://seasonic.com/prime-tx/`, Accessed: 2024-09-14.

[39] SeaBIOS project. SeaBIOS. `https://www.seabios.org/`, Accessed: 2024-09-15.

[40] Rekha Pitchumani Shuyi Pei. CMM-H (CXL Memory Module - Hybrid): Rethinking Storage for the Memory-Centric Computing Era, December 2023. `https://semiconductor.samsung.com/us/news-events/tech-blog/rethinking-storage-for-the-memory-centric-computing-era/`, Accessed: 2024-09-27.

[41] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating system concepts (9th ed.).* John Wiley & Sons, Inc., 2013.

[42] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *The GNU Make Manual.* Free Software Foundation, February 2023.

[43] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, pages 105–121, New York, NY, USA, 2023. Association for Computing Machinery. `https://doi.org/10.1145/3613424.3614256`.

[44] systemd project. systemd - System and Service Manager. `https://systemd.io/`, Accessed: 2024-10-01.

[45] The coreboot project. coreboot. `https://www.coreboot.org/`, Accessed: 2024-09-04.

[46] The coreboot project. coreboot 24.05 release. `https://blogs.coreboot.org/blog/2024/05/23/coreboot-24-05-release/`, Accessed: 2024-09-23.

[47] The coreboot project. coreboot 24.08 documentation. `https://doc.coreboot.org/`, Accessed: 2024-10-02.

[48] The coreboot project. coreboot 24.08 documentation - ABI data consumption. `https://doc.coreboot.org/lib/abi-data-consumption.html`, Accessed: 2024-09-04.

[49] The coreboot project. coreboot 24.08 documentation - coreboot architecture. `https://doc.coreboot.org/getting_started/architecture.html`, Accessed: 2024-10-03.

[50] The coreboot project. coreboot 24.08 documentation - x86 architecture documentation. `https://doc.coreboot.org/arch/x86/index.html`, Accessed: 2024-09-04.

[51] The coreboot project. coreboot repository. `https://review.coreboot.org/plugins/gitiles/coreboot`, Accessed: 2024-09-29.

[52] The Distributed Management Task Force. SMBIOS. `https://www.dmtf.org/standards/smbios/`, Accessed: 2024-09-26.

[53] The kernel development community. Boot time memory management. `https://www.kernel.org/doc/html/latest/core-api/boot-time-mm.html`, Accessed: 2024-09-04.

[54] The kernel development community. drm/amdgpu AMDgpu driver - The Linux kernel documentation. https://docs.kernel.org/gpu/amdgpu/index.html, Accessed: 2024-09-04.

[55] The kernel development community. The kernel's command-line parameters. https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html, Accessed: 2024-09-09.

[56] The kernel development community. ktime accessors. https://www.kernel.org/doc/html/latest/core-api/timekeeping.html, Accessed: 2024-09-17.

[57] The kernel development community. Memory Management APIs - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/core-api/mm-api.html, Accessed: 2024-10-02.

[58] The kernel development community. Swap suspend. https://www.kernel.org/doc/html/latest/power/swsusp.html, Accessed: 2024-09-13.

[59] The Linux Foundation. Kconfig Language. https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html, Accessed: 2024-09-05.

[60] The Linux Foundation. Linux Kernel Documentation - Memory Allocation Guide. https://docs.kernel.org/core-api/memory-allocation.html, Accessed: 2024-09-05.

[61] The Linux Foundation. Linux kernel source tree - 6.9.6 stable release. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v6.9.6, Accessed: 2024-09-04.

[62] The LinuxBoot project. LinuxBoot. https://www.linuxboot.org/, Accessed: 2024-09-04.

[63] TUL Corporation. PowerColor Graphics Cards. https://powercolor.com/, Accessed: 2024-09-14.

[64] UEFI Forum, Inc. ACPI Specification 6.5. https://uefi.org/specs/ACPI/6.5/, Accessed: 2024-09-09.

[65] UEFI Forum, Inc. UEFI Specification 2.10. https://uefi.org/specs/UEFI/2.10/, Accessed: 2024-08-22.

[66] Miquel van Smoorenburg and Adam Lackorzynski. minicom - friendly serial communication program. `https://salsa.debian.org/minicom-team/minicom`, Accessed: 2024-09-04.

[67] Rafael J. Wysocki. Linux Kernel Documentation - System Sleep States. `https://www.kernel.org/doc/html/latest/admin-guide/pm/sleep-states.html`, Accessed: 2024-09-08.

[68] Chenyang Zi, Chao Zhang, Qian Lin, Zhengwei Qi, and Shang Gao. Suspend-to-PCM: A New Power-Aware Strategy for Operating System's Rapid Suspend and Resume. In W. Eric Wong and Tinghuai Ma, editors, *Emerging Technologies for Information Systems, Computing, and Management*, pages 667–674, New York, NY, 2013. Springer New York.