Karlsruhe Institute of Technology

Fast and Power-Efficient System Suspend Using Persistent Memory

Bachelor's Thesis
submitted by

Fabian Meyer

to the KIT Department of Informatics

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

May 9, 2023 – October 11, 2023

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 11, 2023

Abstract

Suspend is an important mechanism to reduce the power consumption of an unused computer system, at the cost of some latency for returning to the working state. The ACPI specification defines several low-power system states such as suspend-to-RAM (S3) and suspend-to-disk (S4), but these compromise either on the power reduction (S3) or wake latency (S4) aspects. In this thesis, we present a novel mode of system suspend based on S3, but using persistent memory to store the working context, called suspend-to-PMem. In contrast to S3, which has to supply uninterrupted power to DRAM, suspend-to-PMem is able to completely power off all devices. We implement our design entirely within the system firmware, using an FPGA-based PCI Express device to improve compatibility between PMem and commodity hardware. We show that our implementation has a lower overall energy consumption than S3 after only one hour in suspend. The wake latency is increased by up to 9 times compared to S3 mainly due to the FPGA's initialization time.

Contents

Abstract **v**

Contents **1**

1 Introduction **3**

2 Background **7**

2.1 Firmware Fundamentals . . . 7

2.2 System Suspend . . . 9

2.2.1 Types of Sleep . . . 10

2.2.2 Power State Transitions . . . 11

2.2.3 Performance Characteristics . . . 12

2.3 Related Work . . . 13

2.4 Coreboot . . . 14

2.4.1 Platform Initialization Stages . . . 15

2.4.2 Platform Specialization . . . 17

2.4.3 Device Communication and Drivers . . . 18

2.4.4 EFI Variables . . . 19

2.4.5 Logging . . . 20

2.5 Intel Optane . . . 21

3 Design **23**

3.1 Requirements . . . 23

3.2 Suspend . . . 24

3.3 Resume . . . 27

4 Implementation **31**

4.1 Firmware Customization . . . 31

4.2 PCIe Device Handling . . . 34

4.3 FPGA Driver . . . 35

4.4 Memory Management . . . 37

4.5 EFI Variables 38
4.6 System Power Off 39
4.7 Kernel Modifications 39
4.8 Development, Deployment, and Recovery 40
4.8.1 Deployment 41
4.8.2 Failure Modes During Development 42
4.8.3 Safeguarding Code Paths 42
4.8.4 Firmware Debugging 43

5 Evaluation 45
5.1 Test Setup and Methods 45
5.2 Reliability 48
5.3 Wake Latency 49
5.4 Power Consumption 50
5.5 Suspend Latency 55
5.6 Discussion 56

6 Conclusion 59
6.1 Future Work 60

Bibliography 63

Chapter 1

Introduction

Reducing the energy consumption of computer systems is becoming an increasingly important task, both in the data center and the consumer space. Data centers aiming to reduce their energy cost can suspend or power off temporarily unused computing hardware [48], for example by consolidating virtual machines [47]. Yet, the latency of waking up inactive hardware must be minimized to avoid performance degradation. In the consumer space, the European Union is even mandating stricter power limits [43], citing the environmental impact of computer systems' electricity consumption. These regulations incentivize device manufacturers to implement low-power idle states. The wake-up latency of such devices is equally important because consumers expect a device to be ready when needed.

Several low-power system states are defined by the Advanced Configuration and Power Interface (ACPI) specification [45, ch. 16.1]. The operating system (OS) can freely choose to enter one of these states by calling into the respective ACPI methods and via CPU instructions. This can be in reaction to a hardware event such as the press of a power button, or due to a software command. Entering into a low-power state allows the system to reduce its power consumption in times of low or zero system load by selectively turning off devices. Yet, exiting a low-power state to resume processing at regular speed is associated with some amount of lost time to re-enable devices and restore system state (wake latency).

It is generally difficult to design a system state that minimizes power consumption and wake latency simultaneously. For example, power state S3, or suspend-to-RAM, has relatively low wake latency as the system context is present in DRAM. Conversely, S3 has to keep main memory powered the entire time. This power state is often entered by laptops while their lid is closed. S3 improves upon battery life compared to the fully-powered state because other devices besides main memory are powered off [48]. Yet, it still drains the battery faster than if main memory was also powered off.

An even lower power state is available in the form of hibernate, or suspend-to-disk, which uses ACPI power state S4. With hibernate, the main memory contents are written to a hibernate file on disk and restored on the next boot. As a result, main memory can be powered off along with all other devices. The ACPI specification defines two modes for entering S4: OS-initiated and firmware-initiated. With OS-initiated S4, the operating system is responsible for saving all context to disk. Likewise, with firmware-initiated S4, the OS delegates this task to firmware, a process known as S4BIOS [45, ch. 16.1.4]. Support for firmware-initiated S4 is optional, and operating systems nowadays generally utilize OS-initiated S4 [18]. On the one hand, hibernate reduces the system power consumption to effectively zero, apart from stray currents. On the other hand, the process of copying from storage to memory increases the wake latency significantly compared to suspend [48]. Specific timings are determined by the size of the hibernate file, which is itself dependent on the size of RAM installed, and the speed of the storage device, such as a hard disk drive (HDD) or solid-state drive (SSD).

In this thesis, we propose a hibernate-inspired extension to system suspend using persistent memory (PMem): suspend-to-PMem. Upon receipt of a request to enter the S3 state, system memory is copied to PMem and restored on wake-up. This allows powering off the system DRAM, as well as the PMem device, for a near-zero power consumption like in hibernate. Yet, we hope to achieve much lower wake latency than with hibernate. Ideally, wake latency would be close to regular (unmodified) suspend-to-RAM.

The proposed design for suspend-to-PMem will be built directly into the firmware, as opposed to the operating system. We argue that this allows for a lower wake latency as the entire resume can happen transparently to the operating system; there is no need to bootstrap the OS at all before memory can be restored. Furthermore, implementing suspend-to-PMem in firmware allows for OS independence. In theory, a commercial system with built-in PMem could have its OS reinstalled, upgraded, or even changed to a completely different OS without affecting its ability to power off main memory on suspend. In practice, we discovered some caveats within the implementation that did necessitate OS kernel modifications as we will show in Section 4.7.

A key goal of our work is the practical implementation and evaluation on commercially available hardware. There are three prerequisites to this goal: first, a commodity persistent memory device; second, a firmware which is easily adapted with custom source code; and third, a hardware that can run the given firmware and also supports the chosen persistent memory.

Persistent Memory. We will be using Intel Optane Persistent Memory [16] ("Optane") for our design, since it is readily available. Optane, which was first introduced commercially in 2017 [4], is a non-volatile storage technology. Therefore, it can be powered off during suspend without data loss. It offers traditional load/store semantics with byte-addressability, similar to DRAM-based memories. For a single Optane DIMM, the maximal read and write bandwidths are $6.6\,\mathrm{GB/s}$ and $2.3\,\mathrm{GB/s}$, respectively [49]. The read performance is good enough to support sufficiently fast copy operations on wake-up.

Firmware. While the source code of commercial firmware is usually not available, the Coreboot project makes an attempt at building an open-source and portable alternative firmware [40]. Coreboot is structured around a multi-stage architecture that performs minimal device initialization before yielding control to a payload such as a bootloader, or directly to an embedded OS kernel. Additional components can be added on top of the base firmware, such as a configuration interface for firmware settings [34]. Most of Coreboot's code is generic and can be ported to a mainboard model by setting the appropriate compile-time flags, but some parts require implementing C functions or ACPI machine language code. These mainboard specialization capabilities are useful for our design since we can hook into existing functions to perform the suspend and restore.

Hardware. Given the choice of PMem and firmware, the hardware choice is severely limited. The firmware should not need to be ported; instead, it should be possible to directly flash an unmodified version of Coreboot such that we have a stable target to test against, and to focus development efforts on the suspend implementation. Since we were unable to obtain Coreboot-supported hardware that is natively capable to interface with Optane, a key part of this thesis will be about integrating Optane with the firmware using additional hardware. Specifically, prior work by Khalil on asynchronous copy offloading for Intel Optane Persistent Memory [24] resulted in the development of an FPGA-based PCIe device that can be integrated into most systems. Their design achieves saturation of Optane's bandwidth at the cost of up to 100 times higher latency. While we plan to use this FPGA implementation to increase Optane hardware compatibility, the additional latency has to be taken into account during the evaluation.

In the following chapters, we describe and evaluate our approach to PMem-backed system suspend in detail. We begin by introducing the relevant technologies and provide an overview of related works in Chapter 2. In Chapters 3 and 4, we describe the theoretical design and practical implementation, respectively. Afterwards, in Chapter 5, we evaluate the implementation's performance on real hardware. Chapter 6 provides insight into potential further research and forms the conclusion of this thesis.

Chapter 2

Background

In this chapter, we set the stage for the rest of the thesis. First, we introduce the fundamentals of system firmware and its interactions with the operating system. Second, we discuss the general behavior of system suspend. This is followed up by an overview of prior art regarding modified system suspend. Then, we provide some background information on the Coreboot firmware, which is used in our implementation. Finally, we introduce Intel Optane as an implementation of persistent memory and discuss its suitability for our design.

2.1 Firmware Fundamentals

A computer system typically consists of hardware, firmware, an operating system (OS), and software application layers. Each successive layer depends on the preceding layers to perform its duties. The firmware is primarily responsible for low-level hardware interactions, which includes setup of the hardware at system power on, known as platform initialization, as well as some general device and power management tasks. The firmware is generally stored "within the computer hardware in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM)" [32, p. 7].

Similarly to how an operating system provides information about the system, as well as run-time services, to its applications, the firmware makes available information about the hardware and the firmware's own capabilities to the OS. Much of this cooperation is guided by the Unified Extensible Firmware Interface (UEFI) [46], Advanced Configuration and Power Interface (ACPI) [45], and System Management BIOS (SMBIOS) [9] specifications. Any references to "firmware" within this thesis are generally intended to mean UEFI-, ACPI-, and SMBIOS-compliant firmware implementations; similarly, we assume any OS to be UEFI-, ACPI-, and SMBIOS-compatible.

The ACPI specification defines a large set of nested data structures collectively known as *ACPI tables*. These tables are created by the firmware and intended to be read by the OS, allowing it to retrieve descriptions of "system information, features, and methods for controlling those features." This includes, among other things, information about hardware devices that "cannot be detected or power managed using some other hardware standard" [45, ch. 5]. In other words, the information contained within ACPI tables abstracts certain information about the system hardware, allowing the OS to be implemented in a more generic way, instead of having to contain code specific to every hardware that it should run on.

ACPI tables may be simple with a fixed structure and provide static information, or they may be dynamic, containing data objects and functions that need to be evaluated. These dynamic tables are specified in a format known as ACPI Machine Language (AML). Usually, firmware authors will write code in the human-readable ACPI Source Language (ASL), which is then compiled to AML byte code [45, ch. 19]. An ACPI-compatible OS includes an AML interpreter, sometimes referred to as an AML virtual machine, to interpret the AML code. AML code may contain complex logic and execution flow control, perform I/O and memory accesses, trigger interrupts, and generally perform many privileged operations [45, ch. 20]. Since all AML is interpreted by the OS, it is possible for the OS to validate and potentially modify the AML instructions before they are executed. This might happen in response to AML code that is found to be generally incorrect, or simply incompatible with the type of OS. The Linux kernel includes runtime features to override individual pieces of AML code and to insert completely new methods [31].

In addition to ACPI, CPU architectures may offer other means of interacting with the firmware from within an OS. The Intel 64-bit architecture supports an operating mode known as System Management Mode (SMM) for such purposes. A special interrupt, the System Management Interrupt (SMI), will cause the processor to enter SMM. SMM executes code located within the firmware's SMI handler within a separate and protected address space. The processor state is restored to its state prior to the SMI when the SMI handler finishes executing [21, vol. 1, ch. 2.1]. By setting additional memory registers before triggering an SMI, the OS can make requests to the firmware such as entering a different power state. The memory location of these registers is defined within ACPI tables.

The OS can further trigger special interrupts known as BIOS interrupts to invoke a firmware operation or retrieve pieces of data. One of these interrupts is BIOS interrupt `15h`. By setting the AX register to hexadecimal value `E820` when invoking interrupt `15h`, the OS can retrieve the firmware's memory table. This table contains a mapping of memory regions, defined by their start address and size, to memory region types. The region type indicates whether the region is reserved by firmware, reserved by a hardware device, etc., or whether it is

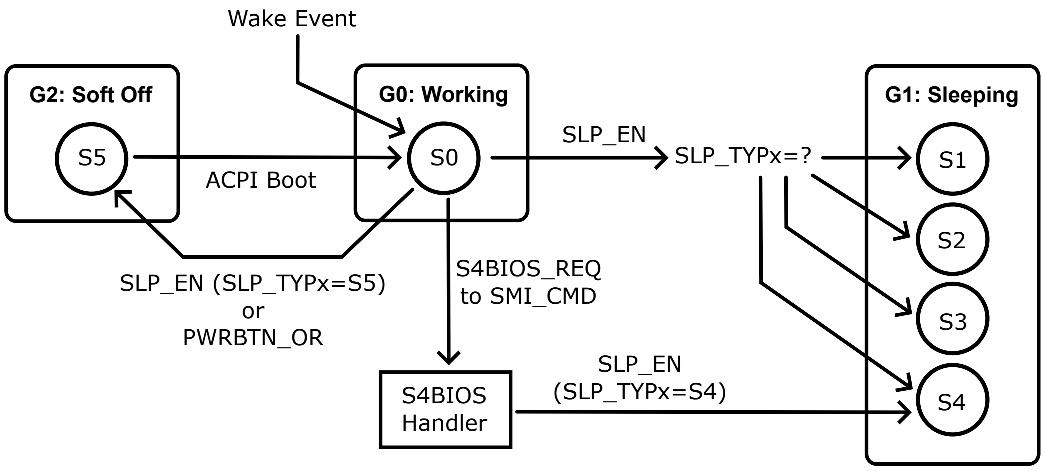


Figure 2.1: ACPI power states G2 ("Soft Off"), G0 ("Working"), and G1 ("Sleeping"). To further differentiate the "Sleeping" state, sleep types S1–S4 are used. Special types S0 and S5 are associated with the "Working" and "Soft Off" power states, respectively. Transitions between states are triggered via external events or internal changes to register values, such as the `SLP_EN` (sleep enable) register. When entering sleep, the `SLP_TYPx` registers indicate the sleep type.

available for the OS to freely use [45, ch. 15]. For reference, the Linux command-line program `dmesg` returns the `E820` memory table as part of its output.

The SMBIOS specification defines further tabular data structures that an SMBIOS-compatible firmware must make available. These tables also contain system information, including information about the hardware and firmware vendors, serial numbers, firmware versions, among many other characteristics [9]. Information like the mainboard serial number are not found within ACPI tables and make SMBIOS an important part of the firmware.

2.2 System Suspend

The behavior and mechanisms related to system power management are defined in the ACPI specification [45, ch. 7, ch. 16]. It specifies different power states and power state transitions, along with the methods of transitioning between power states from within the operating system or as a response to external events, such as the power button being pressed. For the purposes of this thesis, we need to mainly consider the power states and transitions related to sleeping and waking, or suspend and resume, which are shown in Figure 2.1.

An ACPI-compliant computer system can be in one of three main power states, namely G0 ("Working"), G1 ("Sleeping"), or G2 ("Soft Off").

In the G0 state, the system is actively working, although each individual device may be placed in any of its supported power states, including being powered off. According to the ACPI specification, "[i]n the G0 state, [. . .] some work will be taking place in the system" [45, ch. 16.1].

In contrast, the system is not working when in the G1 state. Neither the CPU nor any other devices are operating, "except possibly to generate a wake event" [45, ch. 16.1]. When entering into G1 from G0, the system is said to enter a sleeping state. While sleeping, system context, such as the contents of main memory, is saved by the operating system and/or firmware. Devices are transitioned into lower power states to achieve a reduction in power consumption, however, this depends on the particular device, the targeted sleep state, and whether the device is intended to wake the system. The occurrence of a wake event may transition the system back into G0. The saved system context is then restored and the system resumes its work.

The third power state, G2, is called the "Soft Off" state. In G2, the system is logically off, in that no work is being done and no context is saved. Yet, "power may still be applied to parts of the platform in this state, and as such, it is not safe to disassemble" [45, ch. 16.1]. A system in G2 can transition to G0 by booting.

2.2.1 Types of Sleep

Several sleep types are defined as part of the G1 state, namely S1, S2, S3, and S4; they differ in how system context is saved, and the power states of hardware devices. **S1** sleep preserves all system context except for CPU caches, which are flushed by the OS prior to entering S1. The hardware is responsible for saving "the context of the CPU, memory, and chipset" [45, ch. 16.1.1]. **S2** sleep preserves only the contents of system memory while stopping system clocks and powering off the CPU entirely. On resume from S2, execution starts from the CPU's boot vector, i.e., within firmware. The firmware must re-initialize the CPU, cache, and memory controller, before jumping to the ACPI wakeup vector located within the OS [45, ch. 16.1.2]. **S3** sleep is similar to S2, but is intended to place hardware devices in lower power states than S2 would. Therefore, S2 can support more device wake events than S3, but S3 consumes less power [45, ch. 16.1.3]. S3 is also known as suspend-to-RAM or simply suspend [18]. **S4** sleep is "the lowest-power, longest wake-latency sleeping state supported by ACPI" [45, ch. 16.1.4]. When entering S4, the OS and firmware are intended to power off all devices, including system memory. The memory contents are to be saved by copying them to non-volatile storage, such as a hard disk drive (HDD) or solid-state drive (SSD), prior to powering off the memory. S4 sleep can be initiated directly by the OS, in which case the OS must save the memory context and restore it when waking. Alternatively, the ACPI specification defines an S4BIOS transition that is optional for firmware to implement. For S4BIOS, the OS invokes a firmware command, causing the firmware to perform entry into S4 via its S4BIOS handler, in which case the firmware is responsible for saving the memory context and for restoring memory when waking. S4 is also known as suspend-to-disk or hibernate [18].

In addition to S1–S4, the ACPI specification defines **S0** and **S5**. These identifiers do not refer to sleep states, but represent the power states G0 and G2, respectively. In other words, a transition to S0 is equivalent to placing the system into the "Working" state, while a transition to S5 is identical to placing the system into the "Soft Off" state.

2.2.2 Power State Transitions

The ACPI standard defines, as part of its Hardware Specification [45, ch. 4], several power management-related control registers. For our purposes, the most relevant of these registers is the PM1 control register, located in system I/O or memory space and discoverable via the "FADT" ACPI table (Fixed ACPI Description Table) [45, ch. 4.8.3.2]. PM1 bit 13 is called the `SLP_EN` (sleep enable) bit; setting `SLP_EN` to 1 starts the transition into a sleeping state. PM1 bits 10–12 are called the `SLP_TYPx` fields and indicate the type of sleep to enter. Each sleep state (S0–S5) is associated with an AML data object (`\_S0`–`\_S5`) that resolves to the 3-bit value that, when written into `SLP_TYPx`, causes the associated sleep state to be entered.

As shown in Figure 2.1, several power state transitions are possible. For instance, the system can transition from a "Working" state (G0 / S0) to a "Soft Off" (G2 / S5) state in two ways. The first method involves setting the `SLP_EN` bit to 1 while the `SLP_TYPx` bits are set to the value of `\_S5`. The second method involves a power button override, i.e., performing a long-press of the power button to force a system power off (indicated as `PWRBTN_OR`). The transition to G0 is hardware-initiated and happens via booting. A transition into G1 is possible by setting `SLP_EN` with `SLP_TYPx` set to one of `\_S1`, `\_S2`, `\_S3`, or `\_S4`. Alternatively, when the firmware supports the S4BIOS transition, triggering an SMI with the SMI command register `SMI_CMD` set to the value of `S4BIOS_REQ` will cause the firmware to save system memory to non-volatile storage before transitioning to S4 [45, ch. 16.1]. `S4BIOS_REQ` is a field found in the "FADT" ACPI table; a value of zero indicates that S4BIOS is not supported [45, ch. 5.2.9].

In this work, we are mainly concerned with the S3 sleep, or suspend-to-RAM. When the OS intends to enter S3, it begins by freezing user space processes, before traversing the device tree to let each device driver handle the transition to suspend, saving device context if needed and placing the device into a lower-power state. Every CPU except the boot CPU will be disabled [18, 44]. Some devices cannot be power-managed by the OS; additionally, the firmware needs to prepare for the transition. The OS invokes the ACPI methods `_PTS` (Prepare to

Sleep) and `_GTS` (Going to Sleep), which perform platform-specific operations to achieve these tasks. Afterwards, the OS saves some final context and stores the ACPI wakeup vector, such that it is able to resume later. The OS then sets `SLP_EN` with a `SLP_TYPx` of `\_S3` [45, ch. 6.5]. Generally, the southbridge will trigger an SMI, known as a Sleep SMI, in reaction to these register writes [44]. This lets the firmware power off any remaining device, such as the boot CPU.

2.2.3 Performance Characteristics

A system in the "Working" state (G0 / S0) is going to have the highest power consumption out of all power states, because devices will be in higher device power states compared to G1 or G2. The advantage of G0 over other states is the immediate ability to do work.

In contrast, "Soft Off" (G2 / S5) will essentially power off all devices and a complete ACPI boot is required to enter G0, at which point the system is able to do work. For that reason, G2 offers the lowest power consumption out of all power states, but has a considerable wake latency.

The power consumption and wake latency of the G1 "Sleeping" state depends entirely on the sleep type. S1 sleep generally powers off very few devices, resulting in little reduction of the system power consumption, with the benefit of a low wake latency. S2 sleep powers off most devices and loses all system context except for system memory, therefore requiring another platform initialization on wake, resulting in more power savings than S1 at the cost of higher wake latency. S3 sleep (suspend-to-RAM) performs similarly to S2, with potentially slightly lower power consumption and slightly higher wake latency due to lower device power states compared to S2. S4 sleep (suspend-to-disk, hibernate) powers off all devices including system memory. It is the most similar to S5 ("Soft Off") in that regard and essentially requires a full boot to resume. The power consumption while in S4 is minimal at the cost of significantly higher wake latency [45, ch. 16.1.4]. While the wake latencies of S1–S3 depend mostly on the devices installed in the system, the wake latency of S4 is also dependent on the size of system memory and the speed of the disk to which the hibernate file is saved.

Xi et al. [48] further explore the suspend and resume process, particularly of S3 sleep, in data center environments. Their work investigates the performance of power state transitions from the level of the operating system and a hardware perspective. Crucially, the influence of firmware on power state transition latencies is missing from their analysis. The authors show that the presence of additional hardware such as network cards, graphics subsystems, and USB host devices can significantly increase suspend and wake latencies. Additionally, operating systems differ in their implementations of resuming from suspend, and there are large performance opportunities in parallelizing device wake-up.

2.3 Related Work

There exists some previous work on modifying system suspend to improve upon power consumption, latency, and other metrics.

A similar idea to the one described in this thesis, namely suspending to Phase-Change Memory (PCM), was proposed by Zi et al. [50]. The authors implemented their design in software only, using a modified version of the QEMU system emulator. When the operating system sends a sleep command, QEMU transfers the DRAM contents to a simulated PCM device. When a wake-up event is received, the contents of the simulated PCM are transferred back into DRAM. The specifics of the PCM simulation are missing from the paper, including the performance behavior that was modeled. Due to hardware unavailability, no real-world energy measurements were performed, either, and there is no statement how much (if any) additional power requirements of the PCM memory were taken into account. Instead, the authors compute the theoretical energy consumption of their system based on the full load, idle, and sleep time percentages. They claim energy savings of 80.3% for a system that is fully loaded 10% of the time, idle 2% of the time, and sleeping 88% of the time, assuming power consumptions of $95\,\mathrm{W}$ when fully loaded, $6.25\,\mathrm{W}$ when idle, and $2.5\,\mathrm{W}$ when sleeping. The source of these numbers is unclear and a consumption of $2.5\,\mathrm{W}$ while sleeping is surprising, as we would assume a suspend to non-volatile storage, such as PCM, to fully power off the system. Meanwhile, latency of the wake-up or "resume" phase is increased from $16\,\mathrm{s}$ to $41\,\mathrm{s}$ compared to regular suspend-to-RAM, although the paper neglects to specify the amount of system memory at which these measurements were taken. Another series of experiments shows that the wake latency depends strongly on the amount of system memory, reaching up to $66\,\mathrm{s}$ at $1\,\mathrm{GiB}$ of system memory. Additionally, CPU performance is reduced by up to 6% at run-time due to kernel modifications. The paper lacks an explanation for how a modified suspend can lead to regressions in run-time performance. For their experiments, the authors mention that "there is a few program running in the guest OS" [sic.], but control neither the exact CPU load during idle phases, nor the amount of memory utilized. Furthermore, the experimental results indicate a wake latency that is consistently more than $2\times$ larger than the associated suspend latency. The applicability of this result to hardware storage devices is questionable, as the write performance of non-volatile storage is generally lower than its read performance [33, 49]. As such, we would expect the overhead during suspend to be higher than the overhead during resume. Overall, while we agree with the premise of suspending to non-volatile storage for improved power consumption, more work is needed to achieve a realistic design and obtain real-world performance and energy measurements.

Kim et al. [25] propose an instant on/off system based on a combination of DRAM and non-volatile RAM (NVM). While the kernel resides entirely in

NVM, only some user processes are allowed to reside in NVM. On power-off, the processes in DRAM are lost, but the kernel, CPU registers, and non-code pages of user processes that were saved to NVM remain available after an instant off/instant on cycle. This approach results in an *instant off* latency that is 17-45$\times$ lower than shutdown or suspend, and an *instant on* latency that is 161-244$\times$ lower than boot or resume. The paper demonstrates that a suspend-like state using non-volatile storage can lead to good performance. However, it should be noted that the inability to retain all user processes may limit the applicability of this strategy in some scenarios.

In [14], Ho et al. propose to split the hibernation disk image into multiple parts according to a priority analysis. The most urgent memory pages such as kernel memory are loaded immediately after restore, at which point interaction with the system is already possible. Then, the other parts of the hibernation image are loaded in sequence to minimize page faults. This technique results in a nearly constant resume latency, as opposed to basic hibernate, for which the resume latency grows linearly with the number of processes. We could investigate if incorporating a similar priority analysis into our basic design, either as an extension or in future research, would similarly improve wake-up latency.

Swap-before-hibernate [27] is a hibernation procedure which uses flash storage and a multi-step suspend to speed up the restore. The authors determine that most memory pages are clean disk-backed pages and need not be written to secondary storage at all. Of the remaining memory pages, those that are swappable can be written to a swap file on flash storage. Only non-swappable pages are written to the hibernate file. Resuming from this state requires about 20% of the time required by regular suspend-to-disk on Linux. We may use the insights about clean disk-backed pages to avoid unnecessary work in our design, however, our approach is based on suspend instead of hibernate. Also, we do not explore adding a swapping step to our persistent memory approach because this can only lead to a speed-up when swap storage is indeed faster than PMem.

2.4 Coreboot

We will be implementing our approach within the Coreboot firmware, or, more specifically, within the Dasharo distribution of Coreboot. In this section, we provide an overview of Coreboot, focusing on the relevant aspects for our work.

Coreboot is a modular, open-source firmware implementation that enjoys broad compatibility with various CPU architectures [34, 40]. Notably, it supports both 32-bit and 64-bit variants of x86 CPUs from Intel and AMD, as well as ARM CPUs. It is worth highlighting that Coreboot recently implemented 64-bit execution, which is enabled for certain mainboards [30]. However, in most cases, Coreboot is still limited to 32-bit execution. Additionally, at the time of writing, even with 64-bit execution enabled, certain parts of Coreboot such as the System Management Interrupt handler are still constrained to 32-bit. Since the operating system sets up the CPU again upon handover from the firmware, the OS is not affected by whether Coreboot runs in 32-bit or 64-bit mode.

Coreboot is inherently generic and concerns itself almost exclusively with platform initialization. Section 2.4.1 describes this process in more detail. Following platform initialization, Coreboot launches a preconfigured binary known as the *payload* [39]. A common payload is TianoCore EDK II (EDK II, edk2), which is Intel's reference implementation of UEFI [19]. EDK II, which is an abbreviation for EFI Development Kit, provides UEFI platform services as well as user-facing features, such as a UEFI shell [15, ch. 5].

Dasharo is a commercial Coreboot distribution developed by 3mdeb [1]. Compared to upstream Coreboot, it shares much of the same code, with some changes and additions. As an example, Dasharo includes a graphical user interface for changing UEFI settings. Most Dasharo variants include an EDK II payload and are configured to boot into operating systems installed on storage media. 3mdeb further performs validation of their code against a limited set of platforms [2].

2.4.1 Platform Initialization Stages

Figure 2.2 shows the boot process of Coreboot on x86 architectures starting at hardware power on. The following description of the stages is adapted from the Coreboot documentation on architecture [35]. After a CPU reset, the platform first enters into the *bootblock* stage. The bootblock is written in assembly language. It performs the most basic platform initialization, such as setting the stack pointer, initializing timers, and switching the CPU from 16-bit real-mode to 32-bit protected mode. Notably, main memory is not initialized in the early stages of Coreboot. The bootblock sets up the CPU cache to be used as RAM (Cache-as-RAM, CAR) for the heap and stack [28]. The stages following the bootblock may be compressed with LZ4 or LZMA compression. Each stage decompresses the following stage, if needed, before loading it.

If verified boot is enabled, the next stage after the bootblock is the *verstage*:

> "The verstage is where the root-of-trust starts. It's assumed that it cannot be overwritten in-field (together with the public key) and it starts at the very beginning of the boot process. The verstage installs a hook to verify a file before it's loaded from CBFS or a partition before it's accessed. The verified boot mechanism allows trusted in-field firmware updates combined with a fail-safe recovery mode." [35]

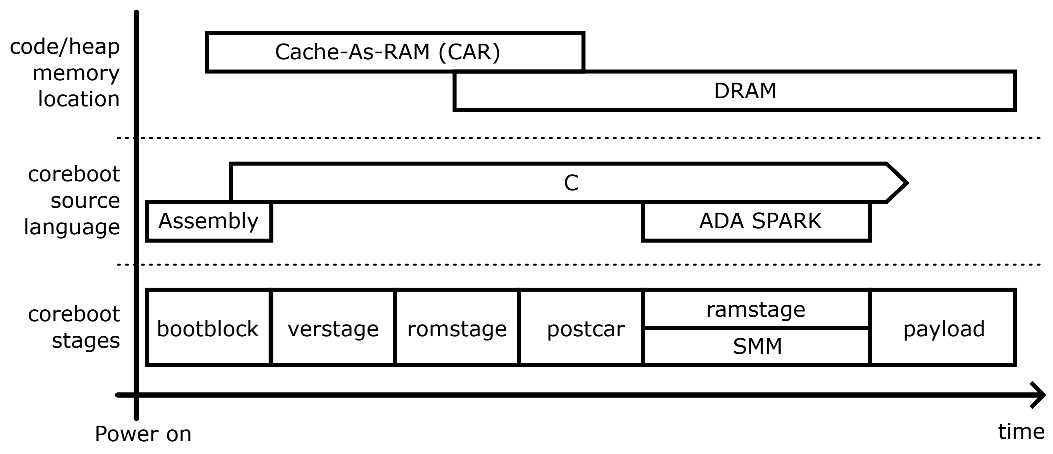


Figure 2.2: Coreboot stages on x86. Platform initialization starts in the bootblock which is written in Assembly language. For transitioning to code written in C and ADA SPARK, the CPU cache is set up as a RAM substitute. The romstage and postcar switch from Cache-As-RAM to proper DRAM. The ramstage performs final initialization tasks before handing over control to the payload. Reentry into the firmware is possible in System Management Mode (SMM), which executes in an environment similar to that of the ramstage.

The next stage is the *romstage*, which performs "early device" initialization. This includes initialization of the DRAM. On some platforms, most commonly Google Chrome OS devices [13] [36], USB controllers are initialized during this stage. Certain console outputs and debug devices also need to be part of early initialization to be useful while developing Coreboot; this topic is further explored in Section 2.4.5. Early initialization of devices is distinctly different from later device initialization, which is performed in a more modular way via drivers and can make use of higher-level primitives. In contrast, early initialization has to perform low-level I/O operations to set up hardware.

The romstage then loads the *postcar* stage, which moves execution to regular DRAM. At this point, the CPU cache is no longer used as RAM.

The final stage of Coreboot is the *ramstage* (RAM stage). This stage performs the bulk of the platform initialization work such as setting up on-chip devices, PCI, graphics, and TPM devices. It also configures the CPU such that it is able to enter into System Management Mode (SMM). SMM is not executed as part of platform initialization but may be executed in response to a System Management Interrupt (SMI). The ramstage writes ACPI tables, SMBIOS tables, and other data into RAM for the payload or operating system to access.

Finally, the ramstage loads the payload, which may be any arbitrary executable. Common payloads include EDK II, SeaBIOS, and GRUB2 [39]. EDK II was introduced in the introductory paragraph of Section 2.4. SeaBIOS is "an open source implementation of a 16bit X86 BIOS." It is the default BIOS for the QEMU and KVM virtualization softwares, but can also run on real x86 hardware when used with Coreboot [8]. GRUB2 by the GNU project is a boot loader, suited to booting into a diverse set of operating systems [11]. Instead of loading an operating system via a boot loader such as GRUB2, it is also possible to load the Linux operating system directly as a Coreboot payload.

All Coreboot stages are compiled and linked separately and can access a different set of library functions. This complicates writing unified code that works in multiple stages, such as in the ramstage and in System Management Mode. Coreboot internally makes heavy use of preprocessor macros to differentiate stages and available library functions. Following the bootblock, most of Coreboot is written in C language code. There are, however, some exceptions. For instance, Coreboot's graphics initialization library *libgfxinit* is written in SPARK, a subset of the Ada language with formal verification features [38].

2.4.2 Platform Specialization

Specializations of Coreboot for different hardware are added directly to the Coreboot source tree. The `src` directory contains several sub-directories for specialized implementations of different components:

- `src/arch` - processor architecture (x86, ARM, etc.);
- `src/cpu` - CPU brand and model;
- `src/ec` - mainboard embedded controller;
- `src/mainboard` - mainboard;
- `src/northbridge` - CPU northbridge;
- `src/southbridge` - CPU southbridge;
- `src/soc` - system-on-chip;
- `src/arch` - Super I/O controllers.

The combination of modules compiled and linked into a firmware binary is decided by an elaborate system of GNU `make` configuration files. The starting point for adapting a Coreboot build is a `.config` file in the root directory. In this file, global build flags can be set, such as the type of mainboard and CPU specializations to link against, whether to build with Trusted Platform Module (TPM) support, whether to enable boot media locking, and many more. Each included module can make further choices. For example, a given mainboard may force a certain CPU architecture, which may in turn toggle support for the Timestamp Counter (TSC) register.

The abstract parts of Coreboot are defined in header files. Implementation files make heavy use of weak linkage to provide default implementations, allowing any of the specialized modules to override them. For instance, the x86 System Management Mode (SMM) trap handler only performs basic SMM setup before handing over control to specialized implementations. Many similar examples exist, such as functions for customizing boot memory regions, restoring devices on S3 resume, among others.

2.4.3 Device Communication and Drivers

Coreboot handles communication with many different types of devices over different communication channels, such as:

- system bus
- system management bus (SMBus)
- PCI and PCIe
- General Purpose I/O (GPIO)
- generic memory-mapped devices
- I^{2}C, UART
- XHCI and USB

The communication with these devices is generally architected as two layers of abstraction. The first layer implements low-level communication primitives; it is used directly by "early device" initialization before and during the romstage. The second layer consists of "devicetree" definitions that are used to generate device information as part of the ACPI tables, and also for use by Coreboot itself. Each mainboard will contain a set of devicetree entries that define its associated devices. An example of such a device tree definition, taken from the Coreboot documentation [37], is shown in Listing 2.1. It would be beyond the scope of this thesis to explain every part of that entry. However, note the `chip` declaration in the example, which indicates that the devicetree entry has an associated structure located at `drivers/i2c/generic/chip.h`. When in the ramstage, Coreboot will attempt to detect the presence of a matching device. If found, ACPI table data will be generated for the device, containing information from the devicetree entry as well as the chip structure [37]. Furthermore, Coreboot will initialize the chip via its driver. This allows for device operations to be performed directly within the firmware.

When adding support for new hardware, one must consider the Coreboot stage during which the hardware will be accessed. If the hardware will be needed solely within the ramstage, creating a devicetree entry with an associated driver is a simple method to add support. Device detection will be handled by existing logic within Coreboot. Operations such as enabling or disabling the device will be called by Coreboot at the appropriate time. If the hardware is to be used in

Listing 2.1: An example devicetree entry.

```
device pci 15.0 on
    chip drivers/i2c/generic
        register "hid" = ""ELAN0000""
        register "desc" = ""ELAN Touchpad""
        register "irq" = "ACPI_IRQ_LEVEL_LOW(GPP_A21_IRQ)"
        register "detect" = "1"
        register "wake" = "GPE0_DW0_21"
        device i2c 15 on end
    end
end # I2C #0
```

earlier boot phases or within System Management Mode, a device driver cannot be used. Then, one must manually execute the lower-level communication primitives within platform specialization functions to set up and tear down the device.

2.4.4 EFI Variables

The term "EFI variables" refers to non-volatile key-value storage managed by the firmware. The UEFI Specification defines related runtime services as well as the variable format [46, ch. 8.2]. The prototype of the UEFI Runtime Service function `SetVariable` is shown in Listing 2.2. Each variable is identified by a combination of an EFI GUID (Globally Unique Identifier) [46, appendix A] and a null-terminated string representing the variable name. The data can be any binary buffer. The UEFI specification reserves several variables with "architecturally defined meanings" under a fixed vendor GUID, but allows for arbitrary additional variables, as long as they are saved with a different vendor GUID or GUIDs [46, ch. 3.3].

Listing 2.2: Prototype of the UEFI function `SetVariable`.

```
typedef EFI_STATUS SetVariable (
  IN CHAR16   *VariableName,
  IN EFI_GUID *VendorGuid,
  IN UINT32   Attributes,
  IN UINTN    DataSize,
  IN VOID     *Data
);
```

In Coreboot, EFI variables are usually stored on the CMOS chip. As such, power needs to be supplied by a CMOS battery for the EFI variables to be persisted, but a continuous supply of mains power is not necessary.

Notably, the Linux kernel exposes EFI variables via a virtual filesystem [22]. On Ubuntu 22.04, this filesystem is mounted at `/sys/firmware/efi/efivars` in read-write mode. This way, firmware behavior can be modified within the OS, without needing to implement a graphical menu as part of the firmware itself.

2.4.5 Logging

Coreboot reserves a region in main memory, called *cbmem*, to store console messages and other data about the state of the system [41]. While the cbmem region is reserved by firmware, the operating system is allowed read access. The cbmem console is implemented as a ring buffer and while it may be truncated due to limited capacity, it is not automatically emptied. As such, console logs from previous system boot-ups may be present, as long as power to main memory was not lost. A `cbmem` command-line utility exists to read the stored log messages from within an operating system. Unfortunately, this type of log output is not useful for debugging a system that fails to boot into the operating system for that reason.

Several other console outputs besides cbmem are configurable when compiling Coreboot [42]:

- Serial: Coreboot can perform input and output via certain non-USB serial ports, though serial ports on add-in cards like PCIe serial cards have only limited support, and few modern mainboards are produced with on-board serial ports.
- EHCI debug port: A limited set of chipsets are supported for debugging Coreboot via the debug port of an EHCI controller. Every USB 2 host controller is an EHCI controller [6], but the EHCI debug port is an optional feature; furthermore, specialized software and/or hardware are required to receive data from the system's debug port.
- spkmodem: The console can be output via the computer's sound card and the signal interpreted by specialized software running on another computer connected to the audio output.
- Network: For NE2000-compatible (legacy) Ethernet cards, Coreboot can send the log files via network to another networked system.
- POST card: Power-On Self Test (POST) cards are built into some mainboards or can be added via PCI. These cards can display a single byte that Coreboot outputs on I/O port 80.

In summary, Coreboot debug methods and access to its console output is unfortunately quite limited. This can partly be explained by it being a firmware; that is, executing with much of the system hardware still uninitialized. Notably, any console output that requires additional hardware cannot be used to debug situations where booting fails before the point at which the hardware would be set up. Furthermore, some work towards adapting existing log methods to newer hardware, such as modern network cards, could prove to be beneficial.

2.5 Intel Optane

The brand name *Optane* refers to two product lines by Intel, which are both based on the *3D XPoint* storage technology. *3D XPoint* was developed jointly by Intel and Micron and first announced in 2015. In 2017, Intel introduced the first Optane SSDs [4]. In 2019, they followed up with *Intel Optane Persistent Memory*, which is available in a DIMM format similar to traditional DRAM modules [16, 20]. The Optane DIMM connects directly to the processor's integrated memory controller via the memory bus [49]. Unlike DRAM, however, Intel Optane Persistent Memory is non-volatile, that is, no data is lost when the storage device is powered off. In this thesis, we consider only the DIMM variant, and any mentions of Optane shall refer to *Intel Optane Persistent Memory*.

Performance. Yang et al. [49] provide a thorough performance analysis of Optane DIMMs. For a single first-generation Optane DIMM with $256\,\mathrm{GB}$ capacity, they measured a maximal read bandwidth of $6.6\,\mathrm{GB/s}$ and a maximal write bandwidth of $2.3\,\mathrm{GB/s}$. The achievable bandwidth for a given workload depends on many factors such as the number of bytes read or written per access. According to the manufacturer, the bandwidth is also different between capacities [16], and we can assume between different product generations. In this thesis, we will be working with a single $128\,\mathrm{GB}$ Optane DIMM of the PMem 100 series, which is a first-generation Optane DIMM.

Hardware compatibility. Not all CPUs readily support Optane DIMMs, but it is possible to integrate Optane into systems lacking native support using additional hardware. For instance, Khalil [24] have implemented a solution based on a field-programmable gate array (FPGA) PCIe card. Specifically, they used an Intel Stratix 10 DX FPGA [17] to interface with Optane. Their design achieves saturation of Optane's bandwidth at the cost of up to 100 times higher latency. In essence, by using an intermediate hardware device between the CPU and Optane, better hardware compatibility can be achieved at the cost of some performance.

Chapter 3

Design

In the previous chapter, we have introduced related works for improving upon system suspend, both from a power consumption and from a wake latency perspective. We have also provided prerequisite knowledge regarding the architecture of the open-source Coreboot firmware. Finally, we have discussed Intel Optane Persistent Memory as a form of DRAM-like, non-volatile storage, and the usage of an intermediate device to improve compatibility between mainstream systems and Intel Optane Persistent Memory.

Building on this foundation, we present our design for an alternative implementation of system suspend, called suspend-to-PMem. Although suspend-to-PMem is based on S3 sleep, we deviate from S3 as defined in the ACPI specification by offloading DRAM contents to Optane while in suspend. This allows the system to be powered off entirely, thereby minimizing power consumption compared to regular S3. The design is specifically intended to work at the firmware layer and uses an FPGA as the intermediary for interfacing with Optane.

3.1 Requirements

We identify three main aspects that we consider mandatory for a successful implementation of suspend-to-PMem:

Reliability. Suspend-to-PMem is intended to serve as a replacement for regular S3 sleep. As such, the same outward behavior must be observed, otherwise any operating system code dependent on that behavior may malfunction. Crucially, suspend-to-PMem needs to fully persist main memory upon suspend entry, then perform a full restore upon wake, and finally resume execution from the operating system wake vector. Any data loss could lead to a system fault or crash, such that the hardware has to be manually power-cycled, negating any of the benefits.

Low wake latency. For suspend-to-PMem to serve as a competitive suspend mode even when suspending for short time periods, it should be relatively fast to resume to the operating system. Optimally, the time between the wake signal and completed wake-up would not exceed the time required by S3 for the same operation.

Low power consumption. Suspend mainly exists to reduce power consumption while the system is not utilized. During regular S3, only very few components continue to be powered. Most importantly, as main memory contents are kept in volatile storage, this storage must be powered. The same is not true when offloading main memory to persistent storage: In that case, all components can be turned off without data loss. This trivially reduces the power consumption during the suspended state. Yet, there is additional work done when entering and exiting suspend to perform the copy and restore, which also involves additional hardware. An implementation should ideally reduce the overall energy consumption for the entire store-shutdown-restore-resume process compared to unmodified S3.

Further metrics besides the three main requirements can be used to qualitatively and quantitatively measure an implementation's efficacy.

In addition to wake latency, the suspend latency, i.e., the time between the suspend request and system shutdown, may be considered. We do not prioritize the suspend latency as much as the wake latency because as a device enters suspend, it is supposedly not utilized any more, and its operator may not care about its state as much. On the other hand, a device waking from suspend is likely intended to be used immediately and any latency should be minimized.

Moreover, we aim to achieve complete transparency to the operating system. Only the firmware layer should be concerned with the mode of suspend, leaving the operating system kernel and all user-space applications unaffected. This approach greatly improves compatibility with different operating systems, as it obviates the need to significantly change their code in addition to the firmware code.

Since we are working at the firmware level, some platform-specific code will be required. The scope of our design will be limited to 12th generation ("Alder Lake") Intel Core CPUs, i.e., a 64-bit x86 platform. Similar designs may be possible for different vendors and CPU architectures.

3.2 Suspend

The first phase to consider for our design is the phase of entering into a suspended state. Fundamentally, the following steps must be performed:

1. Detect that the system is about to suspend.
2. Find and initialize the PMem device.
3. Copy the entirety of system memory to PMem.
4. Backup the ACPI wakeup vector to be restored on resume.
5. Set a variable to inform the following boot about the presence of suspended memory.
6. Power off the system.

The remainder of this section will provide detailed descriptions of each step in order of execution.

Detecting suspend. There are multiple candidate methods for detecting that the system is about to suspend. Note that suspend is triggered by the operating system, and as such, the firmware cannot actively monitor this situation. Instead, we require some kind of hook to be executed by the OS. This leaves at least the following options:

- Adapt the ACPI machine language code for `_PTS` or `_GTS`.
- Implement the suspend logic in System Management Mode and execute in response to a System Management Interrupt.

Each option has some benefits and drawbacks. Interacting with PMem hardware from within AML code to perform a full-memory store may be quite complex. Additionally, as it is the operating system's task to execute AML, the code may not be executed exactly in the intended way or at all. There is no guarantee that the system will actually enter suspend when `_PTS` is called as the OS may still abort the suspend. While `_GTS` may be suitable since it is called later, it is deprecated since ACPI 5.0A [45, ch. 16.1].

An implementation inside the SMI trap handler is theoretically ideal, as regular C code can be used, which is executed in the firmware itself. Additionally, code running in SMM has higher privileges than the OS. It is trivial to access firmware memory to gather additional information about the state of the system, such as memory regions and devices. Yet, since an SMI is the final step of entering suspend, much of the hardware will already be disabled and may need to be reinitialized. There is also no support for 64-bit addressing in Coreboot's SMM, so memory accesses within the trap handler are limited to the lower $4\,\mathrm{GiB}$ region.

For our design, we chose to focus on SMM mostly due to the ability to program in C. The FPGA device hosting Optane, which is connected to the system via PCIe, is able to access all of the system memory using Direct Memory Access (DMA). The task of storing all memory to the device will consequently not be hindered.

Device initialization. When the SMI trap handler is executed and we have determined that this was due to a request for S3 sleep, we first need to initialize the PMem device. This entails locating the PCIe device address based on the FPGA's vendor and device identifiers. Any PCI-to-PCI bridges on the path to the device must be initialized for this to succeed. The FPGA uses memory-mapped I/O which means that its Base Address Register (BAR) must be mapped. Due to Coreboot's SMM being constrained to 32-bit addressing, the BAR must be located entirely within the lower 4 GiB of memory.

Then, the FPGA registers located within the BAR region must be initialized. This includes the setup of two command queues, implemented as ring buffers, and located separately in main memory. One of the queues (the write queue) represents pending store requests, while the other (the read queue) represents pending load requests.

Storing memory. By updating the write or read queue and updating a queue end pointer within the FPGA registers, the FPGA performs a store from main memory to Optane or a load from Optane into main memory, respectively. Each request can be for up to 1 MiB of data. The addresses must be 64 byte-aligned. When suspending, we need to treat main memory as a series of 1 MiB regions and create a store request for each. In case the write queue length is insufficient for the amount of chunks left to write, we must wait for the FPGA to signal that at least one of the prior requests has been served before creating another.

ACPI wakeup vector. The ACPI wakeup vector is located within the APCI tables. It indicates the address within the OS from which to resume execution when waking from suspend. The OS writes this value before entering suspend such that the firmware is able to read it when resuming. In our case, the ACPI tables will be recreated by Coreboot at the next boot. Hence, to be able to resume, we need to backup the ACPI wakeup vector within PMem.

Marking as suspended. Any system boot could potentially be a resume, and we need a method of marking whether this is the case. This can be done with a custom EFI variable. We let a Boolean value "1" denote a suspend state, and either the absence or a Boolean value "0" denote that there is no memory to restore. After all storage and the ACPI wakeup vector have been saved to PMem, we create or update the EFI variable such that it contains Boolean value "1".

Powering off. The goal of this thesis is to fully power down the system during suspend. We do this after all previous steps have been taken, that is, all memory is saved, the ACPI wakeup vector is backed up, and an EFI variable is set to indicate this fact. To achieve this, we need to power off all devices including PCIe devices and bridges. Finally, the CPU must be placed into power state S5 (soft power-off), at which point the system will be effectively shut down.

3.3 Resume

The second phase to consider is the boot process following a suspend. This results in the following high-level algorithm to be executed as early as possible during boot:

1. Read the variable that is set by suspend to determine whether memory needs to be restored.
2. Reset the variable such that subsequent boot processes do not resume from the same memory.
3. Find and initialize the PMem device.
4. Copy PMem contents to system memory, except for firmware memory regions.
5. Restore the ACPI wakeup vector.
6. Jump to the wakeup vector, i.e., into the suspended operating system.

The remainder of this section will provide detailed descriptions of each step in order of execution.

Reading suspend state. At the end of suspend entry, we set an EFI variable to denote that the system is now suspended. This variable needs to be read at boot to differentiate between a resume and a non-resume boot. If the variable is present and set to “1”, we continue with the remaining steps. Otherwise, i.e., if the variable is absent or set to “0”, the boot continues as a regular boot instead.

Resetting suspend state. The following boot should not be treated as a suspend exit, too, unless there was another suspend entry immediately preceding it. We reset the EFI variable to “0” accordingly. It would again be set to “1” in case of another suspend; otherwise, it will stay as “0” and not interfere with the regular boot process.

The reset must always happen immediately after the variable has been read to prevent a possible failure condition. Suppose the reset was positioned later in the resume algorithm. Any failure between reading the variable and resetting it could crash the system. In this situation, the variable would still read as “1” on the next boot attempt, possibly triggering the same failure again. This condition is prevented by resetting the variable early, allowing for a second boot attempt to not hit the faulty code path.

Furthermore, it is imperative to not execute the system from the same memory image, unless the system is entirely side effect-free or all side effects are idempotent. This is because resuming from the same image twice may cause the same side effect twice. For any non-idempotent side effect, by definition its second execution will result in a different outcome than the first. This can lead to inconsistent system states, crashes, and potentially data loss.

Device initialization. Like in suspend entry, we also need to interact with PMem during suspend exit. The same steps as described in 3.2 are performed to find and initialize the FPGA via PCIe and create its command queues.

Loading memory. This step represents the bulk of the resume algorithm. Ideally, resuming from suspend-to-PMem would lead to memory contents that are exactly identical to memory contents resulting from regular S3 resume, given identical memory contents at suspend entry. This is to satisfy our "transparency" design requirement. At least, memory regions belonging to the operating system and userspace must be fully restored to ensure a valid system state when resuming execution within the OS. Some memory regions belonging to the firmware cannot be reasonably restored to the same state they were in when entering suspend. For example, the code performing the restore algorithm resides in one of these regions, as does its stack and heap, and any device information managed by the firmware.

To perform a memory load while not interfering with the firmware execution, we utilize the firmware's memory table (see Section 2.1). We assume that the memory table covers the entire physical address space while containing no overlapping entries. Now, we iterate over its entries. Regions marked as belonging to firmware or marked as hardware-reserved are skipped. Regions marked as available for use by the operating system will need to be restored.

To restore a region, we write instructions into the FPGA's write queue such that it loads the associated data from Optane and places it into the region. Specifically, we need to treat the region as a series of 64 byte-aligned chunks up to 1 MiB in length, due to the FPGA's requirements. For each chunk, we create a load request. In case the read queue length is insufficient for the amount of chunks left to load, we must wait for the FPGA to signal that at least one of the prior requests has been served before creating another.

Restoring the wakeup vector. As described in the previous step, we do not restore memory regions belonging to firmware as part of this algorithm. Notably, the ACPI tables are located within firmware regions, and as such, the ACPI wakeup vector set by the OS is not automatically restored. Yet, we have prepared for this fact by saving the ACPI wakeup vector separately within PMem, as described in Section 3.2, and can read it back such that a jump into the operating system is possible.

Additionally, the ACPI wakeup vector read from PMem must be written into the ACPI tables by the firmware as part of this step. Because regular S3 keeps main memory intact, the OS is allowed to assume that writing the ACPI wakeup vector once into the ACPI tables is sufficient for any number of transitions to S3. In case of another suspend, it may not write its ACPI wakeup vector again. In this situation, our suspend algorithm would read an incorrect value and be unable to complete the resume from this second suspend.

Jumping to the OS. The resume algorithm ends with an exit from firmware code. The firmware places the processor and memory controller into the expected state for a resume, and finally executes a jump to the ACPI wakeup vector [45, ch. 16.1.3.1].

This procedure effectively skips part of the boot process, and instead loads a previously stored operating system back into memory, and resumes execution there. The hardware will have been initialized in a similar way to a regular boot or regular resume. The ACPI, SMBIOS, and other tables will have been created as well, as implied by the need of the OS to access this data at runtime. Yet, the firmware would usually continue by loading a payload into memory and start executing it. This is not needed as the system should resume from where it left off instead of booting a new instance of the OS.

Chapter 4

Implementation

In this chapter, we present a practical implementation of suspend-to-PMem, taking into consideration the requirements and theoretical design discussed in Chapter 3. We implement our functionality into version 1.1.1 of Dasharo's Coreboot distribution for MSI Z690-A PRO mainboards [3]. All source code references within the following paragraphs refer to parts of that distribution, unless otherwise noted.

To interface with Optane, we make use of the FPGA designed by Khalil [24] and adapt its existing Linux kernel driver to function at the firmware layer. An Optane DIMM is installed directly on the FPGA, which is itself connected to the system via PCIe. It is possible to map the FPGA into a region of memory such that several of its internal registers can be read and written by performing regular reads and writes to memory addresses. Furthermore, the FPGA expects memory to be allocated for command queues, which are used to submit requests to the FPGA. One queue contains read requests, i.e., requests to copy from PMem to main memory; a second queue contains write requests, i.e., requests to copy from main memory to PMem. By writing the queue start addresses and sizes into the FPGA's registers, along with keeping a pointer to the current tail element of each queue, the FPGA is able to process the read and write requests on behalf of the CPU. As the FPGA is only able to transfer up to $1\,\mathrm{MiB}$ of data per request, larger transfers must be split in multiple chunks. To avoid running out of queue space, the FPGA's registers must be regularly polled for the amount of completed requests, such that the associated elements can be invalidated and later reused.

4.1 Firmware Customization

The first step in implementing suspend-to-PMem functionality within Coreboot is determining where the additional code should be inserted. According to the fundamental design, we need to hook into the sleep SMI handler as well as the boot process. Some more minute aspects of the implementation require additional insertion points.

As described in Section 2.4.2, Coreboot contains a large number of weakly-linked functions that hook into the firmware processes to allow for platform-specific behavior. Some of these functions are well-suited as insertion points for suspend-to-PMem functionality; other parts of the implementation require direct modification of the firmware code.

We now discuss each relevant code location in turn.

`mainboard_smi_sleep`. Coreboot implements a generic sleep SMI handler function for Intel SoCs in `smihandler_southbridge_sleep`, located at `src/soc/intel/common/block/smm/smihandler.c`. This function is responsible for detecting the target sleep type (S0–S5) from hardware registers, then entering it. In between detecting and entering sleep, however, a call to `mainboard_smi_sleep(u8 slp_typ)` is made. We implement this hook to serve as an entry point to the "suspend" part of our design. We simply check for a request to enter S3 by comparing `slp_typ` against the `ACPI_S3` constant, and execute our suspend logic in case of equality; for any other sleep type, we simply return, allowing the default behavior to run.

Our suspend logic fundamentally consists of the steps described in Section 3.2. Additionally, a mechanism is implemented using EFI variables that allows suspend-to-PMem to be disabled and regular S3 to be executed, which is useful for performing experiments, and is explored in more detail in Section 4.5. We also need to initialize a heap memory allocator, which is further described in Section 4.4. Then, the FPGA connected to Optane is located on the PCIe bus. It is initialized and the entirety of system memory is written to PMem. We also define an "info block" containing the ACPI wakeup vector, which we obtain by walking the ACPI tables, and save this to PMem as well.

Finally, an EFI variable is written to indicate that the system is suspended, and it is powered off.

Boot state machine. In Section 2.4.1, we introduced the different platform initialization stages. We implement the "resume" part of our design as part of the ramstage to take advantage of existing device initialization mechanisms and to have access to information such as the memory table, which is only constructed in the ramstage. The ramstage is implemented within `src/lib/hardwaremain.c` in the form of a state machine. Its sequencing is shown in Figure 4.1.

From the state diagram, we can determine that our resume handler must be inserted immediately after `write_tables`. If it was inserted earlier, the memory table would be unavailable; if it was inserted later, time would be wasted loading a payload that would be overwritten by the memory restore. Unfortu-

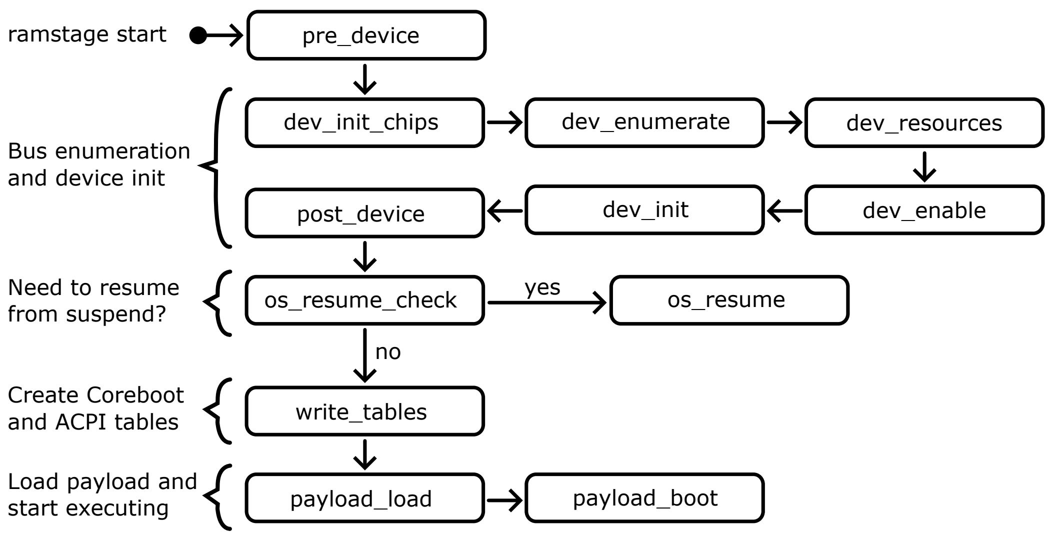


Figure 4.1: The ramstage state machine. Execution begins in `pre_device` and ends in `payload_boot`, at which point the payload takes over. Because our resume implementation needs access to the Coreboot memory table, it must be inserted after `write_tables`. `os_resume_check` and `os_resume` are part of regular S3 and not relevant for suspend-to-PMem.

nately, no weakly-linked function is available for that exact point during the execution, necessitating a direct modification of the existing code. We chose to call the resume handler in the beginning of `payload_load`. The resulting state function is shown in Listing 4.1. First, the EFI variable that determines the suspend state is checked by calling `s3pmem_should_resume`. If this call indicates a pending resume, the firmware is prepared for hand-over to the OS by calling `arch_bootstate_coreboot_exit`, before proceeding with the custom resume logic in `s3pmem_boot_resume`. The resume logic consists of the steps defined in Section 3.3. We use `bootmem_walk` to iterate the memory regions in ascending order and load every region with the `BM_MEM_RAM` tag, indicating a memory region belonging to the OS. We also walk the ACPI tables again to find the wakeup vector location and set it to the value stored in the "info block" during suspend.

`bootmem_platform_add_ranges`. This function is called as part of platform initialization and is used to register custom memory ranges. Its usage as part of this work will be further explored in Section 4.4.

`mainboard_suspend_resume`. While we do not use this weakly-linked function in our implementation, we include it here, as it may be relevant for adjacent or future work. As shown in Figure 4.1, if `os_resume_check` determines that

Listing 4.1: Adapted payload loading state to perform a restore from PMem if necessary.

```
static boot_state_t bs_payload_load(void *arg)
{
    if (s3pmem_should_resume()) {
        arch_bootstate_coreboot_exit();
        int ret = s3pmem_boot_resume();
        if (ret) {
            printk(BIOS_ERR, "s3pmem: Failed OS resume, attempt cold boot\n");
        }
        // ret == 0 means we successfully resumed, so we won't reach here.
    }
    payload_load();
    return BS_PAYLOAD_BOOT;
}
```

the device needs to resume, it sets `os_resume` as the next state. `os_resume` then calls `acpi_resume`, which allows the platform to prepare for resume via `mainboard_suspend_resume` before jumping to an Assembly routine that performs the jump to OS code. Because this hook requires all memory contents to be present already, it is only relevant for regular S3, but not suspend-to-PMem.

4.2 PCIe Device Handling

In Section 2.4.3, we have seen that there are multiple levels of abstraction for adding hardware support to Coreboot. While it would be more convenient to use a PCIe device driver to instantiate the FPGA, this functionality is only available during the ramstage, but not in SMM. As we perform the suspend entry in SMM, our implementation is constrained to lower-level PCIe primitives.

PCIe enumeration. To communicate with a PCI/PCIe device, it must first be located on the PCI/PCIe bus. We assume that any bridges in front of the device have already been enabled by Coreboot when booting into a resume, or by Linux when entering suspend. We define a function `pcie_locate_device` that iterates over the PCI address space until it finds a device matching the expected vendor and device identifiers; for our FPGA, these are `0x3345` and `0x0001`, respectively. Coreboot's functions `pci_s_read_config16(dev, PCI_VENDOR_ID)` and `pci_s_read_config16(dev, PCI_DEVICE_ID)` can be used to obtain both identifiers, where `dev` refers to a PCI address. The device may then need to be enabled by updating its command register to contain the "master" and "memory" bits.

These allow the device to become the bus master, i.e., write to the PCI bus, and to use memory-mapped I/O [29, ch. 7.5.1.1].

Memory-mapped I/O. We exclusively use memory-mapped I/O (MMIO) instead of port-based I/O to communicate with the FPGA. This is done by configuring a specific Base Address Register (BAR) with the start address of a memory region. The FPGA's internal registers can then be read and written simply by reading from or writing to locations within that memory region. For our FPGA, the relevant BAR is BAR0, and it uses 64-bit addressing with memory prefetching. While a single BAR is normally 32 bits wide, 64-bit addressing is possible by combining BAR0 and BAR1 into a double-wide register [29, ch. 7.5.1.2]. In our case, Coreboot is limited to 32-bit execution and therefore 32-bit memory accesses, so that the upper half is simply set as all zeroes.

Notably, the BAR size is determined by the device, and the start address must be an integer multiple of that size. It is possible to determine the BAR size by writing all ones into the register and reading it back; the lower bits will be all zeroes, indicating the address alignment [29, ch. 7.5.1.2.1]. In our case, we expect a BAR0 size of $4\,\mathrm{MiB}$ which implies that the lower 22 bits must be zeroes. Any other result is treated as a failure and the suspend or restore is aborted.

We verify the state of MMIO by reading a static value from the device registers, more specifically the DMA engine's version number, to ensure the device is set up correctly.

Disabling devices. When powering off the system as the last step of suspending it to persistent memory, all remaining PCI and PCIe devices must be disabled. We define a function `pcie_busmaster_disable_on_bus` that recursively visits each PCI/PCIe bus. For every device on the bus, the "master" bit is unset such that the device performs no further writes to the bus.

4.3 FPGA Driver

The FPGA's behavior was designed and implemented by Khalil [24], along with a Linux kernel driver and userspace library. Much of the functionality needed for suspend-to-PMem is already implemented as part of this kernel driver. However, adapting the existing code for use at the firmware layer was a major undertaking due to the wildly different API surfaces of Linux and Coreboot.

Adapting the Linux device driver. The FPGA's Linux driver is implemented as an out-of-tree kernel module and is composed of three parts: a PCIe driver, a character device driver, and a memory management unit (MMU) driver [24].

We need neither the MMU nor the character device driver for suspend-to-PMem because transfers are initiated by a single thread and for large memory regions.

The remaining pieces include code to initialize the FPGA's BAR, data structures to work with its registers, methods for probing its physical and virtual functions, and code for writing to and reading from the device. The latter aspect is further split into the setup of read and write request queues, methods for adding requests to these queues, and polling the queues to ensure requests are fully handled.

There is no direct equivalent for kernel functions such as `pci_resource_len`, `pci_enable_device`, or `pci_iomap` [23, `include/linux/pci.h`, l. 1248, l. 1948] [23, `include/asm-generic/pci_iomap.h`] within Coreboot. These are used by the Linux driver to determine the PCIe BAR size, enable the device for bus mastering, and perform MMIO mapping, respectively.

Therefore, much of the device initialization required a fully custom implementation using low-level primitives as described in the previous section. Fortunately, most of the data structures could be used with only minor changes and the introduction of a few additional `typedef` statements. While the Linux driver is protected against concurrent accesses via locking, Coreboot neither provides suitable locking mechanisms, nor are they necessary. Both during boot and within SMM, execution is limited to a single processor and is never done concurrently. The remaining code had to be fully refactored to not use any PCIe virtual functions, not rely on Linux data types such as hashmaps or scatterlists, and execute in a fully synchronous manner. A custom memory allocator is implemented to satisfy requests for heap memory, which is further described in Section 4.4.

Queue setup. The FPGA uses separate queues for read and write requests, with each queue implemented as a ring buffer. We allocate both queues on heap memory with an allocation size of $1\,\mathrm{MiB}$, resulting in a capacity of approximately $32,000$ elements at a size of $32\,\mathrm{B}$ per element. Some capacity is lost due to a peculiarity in the FPGA's DMA engine, whereby every 128th element becomes a link element and must always point to its immediate successor instead of carrying request data. Every non-link element consists of source address and destination address fields, as well as a field indicating whether the element is valid or invalid, among others. An element is considered valid when it represents a pending request and invalid when it does not, e.g., because it was already processed. After a queue is allocated and its link elements are initialized, its start pointer and size are assigned to registers within the FPGA's BAR0 region. At that point, requests can be created by writing to elements within the queue.

Data transfer. Read and write requests are implemented in an analogous manner. A read, in this case, refers to a data transfer from Optane into main memory, while a write refers to a data transfer from main memory to Optane. Both reads and writes are handled entirely by the FPGA such that the full 64-bit address space is available even if the CPU is in 32-bit protected mode. None of the transfer, except for creating the requests, is handled by the CPU.

The index of the first free element within a queue is stored on the heap next to its elements. This allows for the address to be computed in $\mathcal{O}(1)$ time. Care must be taken to always advance this index beyond any link elements and wrap around to the beginning of the queue once the end is reached. Additionally, writes to the first $1\ \mathrm{GiB}$ are likely to fail, as this region is reserved by Optane. To accommodate, we offset each device address by that amount. After a transfer request is created within the queue, we write the new queue tail index to a register within the FPGA's BAR space. The FPGA monitors this register and reacts to the change by processing the queue.

Queue polling. To synchronize the FPGA's transfer speed and the creation of new requests, the FPGA exposes a pointer to the last element that it has processed. When polling, each element between the previous completion pointer and the new completion pointer is marked as invalid. This frees the elements for re-use. When transferring large amounts of data, such as a transfer of the entire contents of system memory, it is imperative to poll the queue between the creation of requests to ensure that all requests can be processed.

4.4 Memory Management

We need to perform some amount of memory management, both to reserve a region for memory-mapped I/O (MMIO), and to allocate temporary data structures for working with the FPGA. When working with memory, it must be considered that Coreboot is constrained to 32-bit execution. As a result, all data structures and MMIO regions must be placed within the lower 4 GiB.

Coreboot's boot state machine transitions through the `write_tables` state right before loading the payload. As part of this state, a call to the weakly-linked `bootmem_platform_add_ranges(void)` is made, allowing platform-specific memory regions to be added. We implement this function and reserve two regions via calls to `bootmem_add_range(u64 start, u64 size, BM_MEM_RESERVED)`.

Dynamic allocation. We implemented a simple memory allocator within the $4\ \mathrm{MiB}$ region starting at address `0x400000` ($4\ \mathrm{MiB}$), which was otherwise unused as reported by Coreboot's memory table. A capacity of $4\ \mathrm{MiB}$ fits all temporary data as well as the FPGA's read and write queues, which occupy slightly more than $1\ \mathrm{MiB}$ each. When initialized, the memory allocator places a struct at the beginning of the region containing a start address pointer, the capacity, and consumed amount. Invoking `s3pmem_malloc(u32 size)` returns a 64-byte-aligned pointer and increments the consumed amount, or returns `NULL` if `size` is too large to fit. `s3pmem_malloc_zero(u32 size)` allocates memory that is 0-filled. All data is ephemeral and required at once; therefore, no deallocation is needed.

Memory-mapped I/O (MMIO). A dedicated memory region is reserved for the FPGA's MMIO such that there is no overlap with any data that should be stored on PMem. The FPGA requires $4\,\mathrm{MiB}$ of memory, the start address of which is written to its Base Address Register 0 (BAR0). PCI requires BAR0 to be aligned to the size of the region. Indeed, we were able to reserve $4\,\mathrm{MiB}$ of memory at address `0x40000000` ($1\,\mathrm{GiB}$).

4.5 EFI Variables

EFI variables are used to store two important pieces of information:

- `S3PmemEnabled`: whether to use suspend-to-PMem instead of regular S3
- `SuspendToPmem`: whether the system was suspended and memory contents are waiting to be loaded from Optane

While EFI variables can store arbitrary binary data, our implementation only requires Boolean data for the aforementioned settings. As such, we have implemented utility functions to simplify working with EFI variables to store Boolean data. `int efivars_set_bool(const char *name, bool value)` is used to write a Boolean value to the variable of the given name, creating it if necessary. `int efivars_get_bool(const char *name, bool *value)` is used to read a Boolean value of the given name and store it at the memory location pointed to by `value`. Fortunately, Coreboot's low-level primitives for working with EFI variables are part of the compilation targets for every stage including SMM, using an identical interface throughout.

`S3PmemEnabled` is read at the start of the SMI handler. If `efivars_get_bool` returns a non-zero value, the variable is likely to be missing, which is the case on the first suspend after the firmware is flashed. In this case, suspend-to-PMem defaults to being enabled and the variable is created with a value of `true`. If the variable is present but `false`, the sleep SMI handler returns and Coreboot performs a regular S3. If it is `true`, however, our suspend implementation executes. At the end of our implementation, the EFI variable `SuspendToPmem` will be set to `true`, too. On the next boot, this variable is read to detect whether our resume implementation needs to be executed instead of performing a regular boot into the payload.

4.6 System Power Off

As the final step of entering suspend, the system needs to be powered off, that is, placed into the S5 “soft off” state. The steps of our Intel x86 platform-specific `power_off` procedure are shown in Listing 4.2. As this procedure is executed as part of the SMI handler for S3 sleep, the previous power management registers must be overwritten with a request to enter S5 instead. Coreboot’s SMI handler has already disabled further SMIs from reaching the firmware at this point. Therefore, setting `SLP_EN` does not trigger another SMI, but lets the hardware receive and act on the S5 request. The CPU is placed into a loop executing the `hlt` instruction until it is powered off.

Listing 4.2: Powering off an Intel x86 platform.

```c
void power_off(void)
{
    // Prevent GPIO events from waking the system
    pmc_clear_all_gpe_status();
    pmc_disable_all_gpe();

    // Disable all PCI busmastering
    pcie_busmaster_disable_on_bus(0, smihandler_soc_disable_busmaster);

    // Disable the watchdog timer
    if (is_wdt_enabled())
        wdt_disable();

    // Clear any previously requested SMI sleep state and request S5
    pmc_clear_pm1_status();
    pmc_disable_pm1_control(~0);
    pmc_enable_pm1_control(SLP_EN | (SLP_TYP_S5 << SLP_TYP_SHIFT));

    // Halt the CPU
    halt();
}
```

4.7 Kernel Modifications

Some modifications of the Linux kernel are necessary to achieve a reliable suspend. This is due to certain implementation details of the FPGA that we describe in the following paragraphs. Such modifications would not be needed with a pure Optane implementation.

Power management. When entering suspend, the Linux kernel checks for the "power management" capability on each PCIe device, and if the capability is present, places the device into a low power state. While the FPGA does report this capability, it does not implement any power management-related functionality. As a result, when the kernel attempts to power-manage the FPGA, the FPGA enters a failed state. This must be prevented such that the FPGA can be accessed during the sleep SMI handler. Additionally, none of the PCI bridges in front of the FPGA may be power-managed.

When transitioning the system to a lower power state, the Linux kernel calls `pci_target_state(struct pci_dev *dev, bool wakeup)` to "find an appropriate low power state for a given PCI device" [23, `drivers/pci/pci.c`, l. 2631 ff.]. We patch it to match `dev->vendor` and `dev->device` against the known vendor and device identifiers of either the PCI host bridge, the PCI-to-PCI bridge in front of the FPGA, or the FPGA itself. In case of a match, a fixed power state of `PCI_D0`, or fully powered, is returned; otherwise, the function is allowed to continue [29, ch. 5.3.1].

BAR0 remapping. The FPGA exposes 511 "VFs", or virtual functions, each with their own registers. The combined BAR space of all VFs exceeds $4\,\mathrm{GiB}$, which Coreboot is unable to reserve. The Linux kernel detects this and performs a remapping of BAR space. This places the physical BAR0 beyond the $4\,\mathrm{GiB}$ address boundary that is accessible in 32-bit SMM, preventing any communication with the FPGA.

On boot, the kernel calls `pci_assign_resource(struct pci_dev *dev, int resno)` to assign or potentially re-assign memory regions to PCI devices [23, `drivers/pci/setup_res.c`, l. 325 ff.]. We patch it to match the vendor and device identifier against the FPGA's vendor and device identifier. In case of a match, return `0`, indicating that the existing resource assignment is accepted; otherwise, the function is allowed to continue. The kernel will then simply keep the BAR assignment previously set by Coreboot although the VFs are incorrectly mapped.

4.8 Development, Deployment, and Recovery

In this section, we explore some peculiarities of the development process along with methods of debugging and recovery that were discovered during this thesis, in the hope that they may help others also working on Coreboot.

4.8.1 Deployment

Building a Coreboot image from source consists of many steps, and we will not explore all of them in detail, as that would be beyond the scope of this work. In any case, a set of shell scripts were developed as part of this thesis that automate the building and flashing procedure almost entirely. We recommend against building and flashing manually as it is very error-prone and can potentially lead to a damaged mainboard. The tools and development guide included with the Dasharo distribution of Coreboot are also a good starting point, however, one will most likely need to adapt them to the platform at hand. A general summary of the steps required for building an image follows.

- Ensure all dependencies are installed and present. Refer to the Dasharo guide to determine the requirements.
- Remove any output from previous builds, if necessary.
- Copy the mainboard-specific config into the root directory. For example:
  `$ cp configs/config.msi_ms7d25_ddr5 .config`
- Update config settings as needed. For instance, disable boot media locking, set the presence of an SMI handler, and enable cbmem logging during SMM.
- Update git submodules used for 3rd-party code.
- Run Dasharo's docker image to execute `make` to build the image.
- Use the included *cbfstool* to update SMBIOS data within the image.
- Use the included *vboot* utility to sign the image.

To deploy a Coreboot image onto the target system's EEPROM or flash storage, the *flashrom* command-line program [10] is used. UEFI "Secure Boot" must be disabled to allow for re-programming of the firmware. When flashing a mainboard for the first time, it is highly recommended to create a backup of both the previous firmware image as well as the SMBIOS serial number and system UUID. The respective command lines are shown below.

```
flashrom -p internal -r backup.rom
dmidecode -s baseboard-serial-number >serial_number.txt
dmidecode -s system-uuid >system_uuid.txt
```

Before replacing the current firmware, a "dry run" of flashrom can validate the system state for flashing. This is done by running flashrom as `flashrom -p internal`, where `-p internal` specifies that the internal programmer located on the mainboard itself shall be used. flashrom may indicate errors such as an unsupported chipset, read-only EEPROM regions, among others. In such cases, it is unsafe to flash; otherwise, one may continue, and perform either a full flash when flashing a mainboard for the first time, or perform a much faster partial flash if only some code has changed. The respective command lines are shown below.

```
# full flash of the entire EEPROM
flashrom -p internal -w coreboot.rom --ifd -i bios

# partial flash, only update code regions
flashrom -p internal -w coreboot.rom --fmap -i RW_SECTION_A -i RW_SECTION_B
```

4.8.2 Failure Modes During Development

When developing firmware on real hardware, one also has to consider potential failure modes. During the development of our implementation, we encountered multiple failures that can be placed into two broad categories.

Flashing error. This failure occurs when the flashing software or hardware cannot correctly write the new firmware image into EEPROM; we encountered this multiple times while working on this thesis. After the write-erase cycle is done, flashrom performs a verification of the written sections. This step may finish with an indication that some bytes were not correctly written. While the check is prone to produce false-positives in case writing succeeded but reading failed, it should be treated with care, since any actual failure to write may prevent the system from booting. In the case of flashing errors, we were always able to re-run flashrom with the same image to achieve a successful flash.

Programming error. This failure occurs when the firmware code is changed in such a way that it consistently triggers a fault, infinite loop, or any other kind of bug such that booting into a working operating system for re-flashing is made impossible. This situation is very easy to create especially when working directly with the code responsible for boot, which was required for this project. In such a case, the EEPROM cannot be flashed via the internal programmer and an external programmer has to be used instead. This programmer has to be physically placed firmly on the EEPROM chip, connected to a secondary system, and used to flash back the original vendor firmware image. We found this process to be significantly more time-consuming and error-prone than using flashrom with the internal programmer. It usually required multiple attempts with each attempt taking approximately 15 minutes. While our mainboard provided a “USB flash-back” feature intended to enable flashing via a USB thumb drive inserted in a dedicated USB port, we did not get this to work.

4.8.3 Safeguarding Code Paths

Because programming errors can lead to a significant time investment for restoring the mainboard to working condition, or even result in irreparable damage to the mainboard in case of improper operation of the external programmer, we recommend introducing safeguards into all code under test. This would be any Boolean expression that can be made to evaluate to either `true` or `false` on different boots, without requiring that the previous boot attempt succeeded. By introducing conditional statements, any added or changed functionality can be made to execute only if the safeguard expression evaluates to `true`; otherwise, the unmodified, known-good code would be executed instead. If the code under test contains programming errors that would usually prevent the system from booting, and as such prevent flashing of a non-faulty firmware image, letting the safeguard evaluate to `false` and rebooting will execute the known-good code path. Then, the fault can be corrected and another firmware image can be flashed.

Ideally, the safeguard expression would be deterministic and controllable by the human operator. This could be any input to the system that can be read in the relevant firmware stage. It may be difficult, however, to obtain an external input early in the boot process when only a limited set of devices are initialized. In that situation, a random bit generator can be used.

We investigated the operations available at early boot stages to identify a source of randomness. Our findings revealed that the second least-significant bit of the CPU's timestamp counter (TSC) had an approximately 50% chance of being 0 or 1. Surprisingly, we did not observe a similar distribution on the last bit, which always read as 1. While using a random bit like this may require multiple reboots before the code under test is executed, this can still save time compared to using an external programmer or having to order replacement hardware.

4.8.4 Firmware Debugging

In Section 2.4.5, we looked at each of Coreboot's primary console outputs and logging capabilities. We found none of the hardware outputs particularly useful due to severely limited support for recent devices. The test system's mainboard did not include an on-board serial output. As a result, we exclusively referred to cbmem to retrieve log messages.

The `console/console.h` header provides a `printk` function that is widely used across the Coreboot codebase for appending log messages [3, `src/include/console/console.h`, l. 55]. Its usage is similar to `printk` contained within the Linux kernel interface in that it receives a log level, a format string, and a variadic number of format parameters. An example for its usage is shown in Listing 4.3. We found `printk` critical during the implementation of suspend-to-PMem for the ability to retrieve almost any amount of system information and filter it on the command line using tools such as `grep`. cbmem can survive system resets as long as main memory is not explicitly cleared or unpowered as part of the reset.

While developing the resume phase, there was a need to receive debug output

Listing 4.3: Logging with `printk`.

```c
#include <console/console.h>

void foo(int *p)
{
    printk(BIOS_INFO, "%s(%p):␣value=%d\n", __func__, p, *p);
    // prints: "[INFO] foo(0x...): value=..."
}
```

even when the jump into the operating system failed or the OS refused to proceed from there. An investigation of alternative output methods revealed a colored framebuffer set up by Coreboot and located at address `0x80000000`. By looping over the framebuffer in 4-byte increments, which corresponds to neighboring pixels on the screen, we could output different-colored stripes to indicate how far execution had progressed before failing. A benefit of this approach is its trivial portability to Assembly language for use in the Linux kernel. Listing 4.4 shows sample code to set an area of the framebuffer to a blue color. The relevant Linux kernel source file, in this case, is `arch/x86/realmode/rm/wakeup_asm.S`, with further wakeup logic located in `arch/x86/realmode/rm/trampoline_64.S` [23]. The framebuffer debugging strategy is no longer useful once the kernel has set up paging as it relies on a fixed memory location.

Listing 4.4: Writing to the framebuffer from Linux to show a blue stripe on screen (Assembly language).

```asm
        mov $0x2000, %cx
        movl $0x80000000, %eax
fbloop1:
        movl $0x000000FF, (%eax)
        add $4, %eax
        dec %cx
        jnz fbloop1
```

Chapter 5

Evaluation

In this chapter, we provide a framework for evaluating suspend-to-PMem on hardware and use it to quantify the success of our implementation. In Chapter 3, we formulated three main requirements for the implementation: reliability, low wake latency, and low power consumption. First, we describe the hardware and test setup for evaluating suspend-to-PMem. Then, for each of the requirements, we state the measurement process in detail as well as giving an analysis of the results. Finally, we will discuss some secondary quantitative and qualitative criteria for evaluation.

5.1 Test Setup and Methods

For the test system, we used an MSI Z690-A PRO WIFI model mainboard with an Intel Core i7-12700K CPU (8 "performance" cores with 2× hyperthreading + 4 "efficiency" cores) and 4× 8 GB DDR5-5600 RAM (CL40-40-40). This specific mainboard was chosen because of pre-existing support by the Dasharo distribution of Coreboot [2]. The power supply is a be quiet! System Power 9 CM 500W and is 80 PLUS Bronze rated. 80 PLUS is a "performance specification and certification program for internal power supply units (PSUs)" [5]. An 80 PLUS Bronze rating, in the case of a 230 V internal PSU sold in the EU, indicates an efficiency of 85% at 20% load, 88% at 50% load, and 85% at 100% load, respectively.

On this system, we installed Ubuntu 22.04 LTS with Linux kernel v6.1.33 patched according to Section 4.7. The firmware has been built and flashed using the shell scripts developed as part of this thesis (see Section 4.8.1). Swapping of memory pages to disk is disabled.

In addition to the test system, we used additional equipment for controlling the benchmark operations and for performing measurements. First, we attached a PiKVM to the test system. PiKVM is a Raspberry Pi microcomputer-based hardware kit capable of virtually pressing power and reset buttons, reading LED states, capturing graphics output, among other features [7]. In addition to a graphical user interface, PiKVM offers a query and command interface via HTTP and WebSocket. Second, we connected a GWINSTEK GPM-8213 Digital Power Meter to the test system's power supply. This power meter is capable of measuring both apparent power (in VA) as well as active power (in W) [12]. According to our own measurements, the power meter's measurements update approximately three times per second. They can be read via a USB, RS232, or LAN interface.

To conduct the experiments, coordinate devices, and collect measurement data, a virtual server located on the same VLAN was used. We will refer to this virtual server as the "controller." We determined the network latency between the controller and all other relevant devices to be consistently lower than $1\,\mathrm{ms}$.

Benchmark runs. We performed 4 benchmarks for regular S3 and 4 benchmarks for suspend-to-PMem. The benchmarks were performed with system memory usages of $1\,\mathrm{GiB}$, $4\,\mathrm{GiB}$, $8\,\mathrm{GiB}$, and $16\,\mathrm{GiB}$, representing approximately 3%, 12.5%, 25%, and 50% of total system memory. A particular memory usage was achieved by first retrieving the current memory usage from `/proc/meminfo`, then allocating and filling memory pages according to the missing amount.

Each benchmark targets 100 successful iterations, or "runs," of a suspend-resume cycle. A run contains the following steps to be executed by the controller:

1. Ensure the system is powered on and working, that is, it responds to network requests.
2. Ensure the system's memory usage is equal to the targeted memory usage.
3. Trigger a suspend entry by executing `systemctl suspend` via SSH.
4. Wait for the power LED to indicate system power off. This is detected via a low-latency WebSocket connection to PiKVM.
5. Wait 5 seconds.
6. Press the power button via the PiKVM HTTP interface.
7. Wait for the power LED to indicate system power on. This is detected via a low-latency WebSocket connection to PiKVM.
8. Wait for the system to come online, that is, start responding to network requests.
9. Retrieve kernel and cbmem logs.

A run is successful when all of the steps are executed successfully. A step may fail when either the request to the device fails, such as failure to connect to the test system via SSH, or a timeout is hit. For instance, we only wait for up to $60\,\mathrm{s}$ for the system to come online before considering that step, and therefore the entire run, as failed.

Timing measurements. For our timing measurements, we had to rely on time information from multiple sources. On the firmware side, we introduced calls to `printk` (see Section 4.8.4) with timestamps, to be read via cbmem log retrieval. These timestamps are derived from the CPU's TSC (Time Stamp Counter) register, and although they have a high resolution of microseconds or better, are only useful for computing durations and the relative time of events. For suspend entry, we collect TSC timestamps at the beginning of our sleep SMI handler, when copying starts, and when copying ends. Because the system is powered off afterwards, the cbmem log would be lost. We store the timestamps as part of the info block that contains the ACPI wake vector, which is read back during resume, and print the timestamps then. For resume, we collect TSC timestamps at the beginning of our resume subroutine, after the info block has been read, and after all memory has been restored. The info block is very small and fits in a single 64-byte read request. Hence, we can assume that the time at which it was read closely corresponds to the time at which the FPGA was fully initialized.

We retrieve additional timestamps from the Linux kernel logs of the test system. By executing `journalctl -kb -o short-iso-precise`, we are able to retrieve log lines with absolute timestamps in microsecond resolution. The log lines containing `PM: suspend entry` and `PM: suspend exit` messages are particularly relevant. The former marks the handover from the kernel to firmware when suspending. The latter marks the jump back into the kernel on resume.

Furthermore, we can associate each action taken by the controller with a high-resolution timestamp. We made sure that the test system's and controller's system clocks were synchronized. The list of all timestamps and durations collected is shown in Table 5.1. Throughout the following sections, we will refer to the timestamps via the symbols defined therein.

Symbol	**Definition**
T_{start}	(3.) Controller requests suspend
T_{ksuspend}	Kernel logs `PM: suspend entry`
T_{poweroff}	(4.) PiKVM reports system power LED as off
$T_{\text{powerclick}}$	(6.) Controller presses the power button via PiKVM
T_{poweron}	(7.) PiKVM reports system power LED as on
T_{kresume}	Kernel logs `PM: suspend exit`
T_{online}	(8.) Test system starts responding to ping
Δt_{store}	Duration of writing to PMem on suspend
Δt_{load}	Duration of reading from PMem on resume
Δt_{init}	Time between resume start and completed info block read

Table 5.1: The timestamps and durations collected. Timestamps are shown in the order of events. Numbers refer to the steps of a benchmark run.

Power measurements. For each benchmark run, we collected a list of power measurements using the GPM-8213 Digital Power Meter at intervals of approximately $0.3\,\mathrm{s}$, which is close to the maximum data rate we could achieve. Each data point was individually associated with a high-resolution timestamp before being stored by the controller. We determined that there was no significant time offset between a change in power consumption and a change in the value reported by the power meter besides the $0.3\,\mathrm{s}$ resolution.

5.2 Reliability

According to the first requirement as defined in Section 3.1, the reliability of our implementation must be evaluated. The term "reliability" refers to the ability to enter and resume from a suspended state, restoring the operating system to working condition with identical memory contents compared to the memory contents at suspend entry. In the case of suspend-to-PMem, reliability is determined by the ability to fulfill the designed behavior according to Sections 3.2 and 3.3. Non-reliability may arise when any step of the suspend or resume algorithms fail. In that case, the implementation may hang and not reach power off on suspend entry; or, the implementation may fail to restore the operating system to working condition with identical memory contents on resume. The reliability of regular S3 suspend must also be measured to obtain a baseline reliability value.

Results. We compute the reliability as the number of successful runs divided by the number of total runs.

The results for regular S3 are shown in Table 5.2. Overall, reliability did not differ much between memory usages, with 3 to 4 failures per 100 successful runs. The percentage of successful runs to total runs varies between 96% and 97%. In every failure case, the log files indicate a timeout while waiting for the system to come back online, although the power LED has shown a powered system state for at least $60\,\mathrm{s}$. We manually observed a few of these failures and are able to confirm that the system was powered, outputting an empty image to the display, indicating a failure in firmware.

Table 5.3 shows the results for suspend-to-PMem. Again, the reliability is mostly constant across memory usages, although it is higher than with regular S3 at 10 to 14 failures per 100 successful runs. The percentage of successful runs to total runs varies between 88% and 91%. An investigation of the failure cases shows identical behavior to regular S3, with the system failing to come back online, while being powered and outputting an empty image to the display.

1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
100/103 (97.1%)	100/103 (97.1%)	100/103 (97.1%)	100/104 (96.2%)

Table 5.2: Reliability of regular S3 suspend for different memory usages. (#Success / #Runs)

1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
100/110 (90.9%)	100/113 (88.5%)	100/112 (89.3%)	100/114 (87.7%)

Table 5.3: Reliability of suspend-to-PMem for different memory usages. (#Success / #Runs)

5.3 Wake Latency

The second requirement, as defined in Section 3.1, is low wake latency. Therefore, the wake latency when using suspend-to-PMem must be measured and compared to the wake latency of regular S3.

Results. We compute the wake latency as $\Delta t_1 = T_{\text{kresume}} - T_{\text{poweron}}$. We can also compute the time it takes for the system to come back online, that is, start responding to `ping` requests. This result is of interest because the kernel needs to be fully initialized to reach this state. We compute this additional latency as $\Delta t_2 = T_{\text{online}} - T_{\text{kresume}}$. From the Coreboot logs, we obtain Δt_{init}, i.e., the time required to initialize the FPGA, and Δt_{load}, i.e., the time required to copy memory contents from PMem to RAM. Trivially, since these operations are part of the total wake process, it follows that $\Delta t_1 > \Delta t_{\text{init}} + \Delta t_{\text{load}}$. Only successful runs are considered because failed runs will be missing at least one of these values.

The results for regular S3 are shown in Table 5.4 and the results for suspend-to-PMem are shown in Table 5.5. The system memory usage did not significantly influence the wake latency of either mode. However, the wake latency of suspend-to-PMem is consistently higher than regular S3, with a slowdown between $8.71\times$ and $9.11\times$. The time for the system to come online (Δt_2) is not significantly different between suspend modes and memory usages.

Figure 5.1 further shows time spent on different parts of the resume process. The chart was created by splitting each run into its time components, then graphing the median of each component. We can clearly see that the FPGA initialization (Δt_{init}) along with copying data between PMem and RAM (Δt_{load}) are responsible for most of the total wake latency. The additional overhead, calculated as $\Delta t_1 - \Delta t_{\text{init}} - \Delta t_{\text{store}}$, is very similar to the total wake latency of regular S3. Notably, the large initialization time is a result of the FPGA's architecture, as it has to perform a "memory training" step each time.

	1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
Δt_1	2.23 ± 0.32	2.27 ± 0.30	2.25 ± 0.28	2.17 ± 0.39
Δt_2	2.50 ± 1.39	2.34 ± 1.18	2.52 ± 1.56	2.54 ± 1.37

Table 5.4: Resume times for regular S3. Shown are the times between power on and kernel resume (Δt_1) and between kernel resume and system online (Δt_2) for different memory usages.

	1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
Δt_1	19.63 ± 0.38	19.72 ± 0.35	19.62 ± 0.34	19.78 ± 0.35
Δt_2	2.93 ± 1.97	2.48 ± 1.35	2.50 ± 1.24	2.91 ± 2.24
Δt_{init}	9.06 ± 0.01	9.06 ± 0.01	9.06 ± 0.01	9.06 ± 0.01
Δt_{load}	8.11 ± 0.00	8.11 ± 0.00	8.11 ± 0.00	8.11 ± 0.00

Table 5.5: Resume times for suspend-to-PMem. Shown are the times between power on and kernel resume (Δt_1), between kernel resume and system online (Δt_2), and the time components of initializing PMem (Δt_{init}) and restoring RAM (Δt_{load}), for different memory usages.

5.4 Power Consumption

The third and final requirement, as defined in Section 3.1, is low power consumption. While regular S3 powers off most components, it keeps main memory powered. Suspend-to-PMem is able to power off the entire system at the cost of some additional energy to transfer data between RAM and PMem. For this reason, we need to evaluate both the energy required to suspend and to resume, as well as the power consumption in the suspended state. Importantly, the device may stay suspended for any amount of time. At some point, the energy saved while suspended will likely offset the energy required to enter and exit from the suspended state, resulting in a "break-even" point.

Results. We measured the power consumption simultaneously to the other measurements as described in Section 5.1. From this data set, we are able to approximate both the energy consumption as well as the average power consumption for any given time interval. We note that power is the amount of energy transferred or converted per unit time. As a result, integrating the power measurements over a given interval yields the energy consumed. The average power consumption can then be computed by dividing the energy consumed by the length of the interval.

Figure 5.2 visualizes the power measurements for each benchmark that was performed. The charts show all of the runs' power measurement curves overlaid

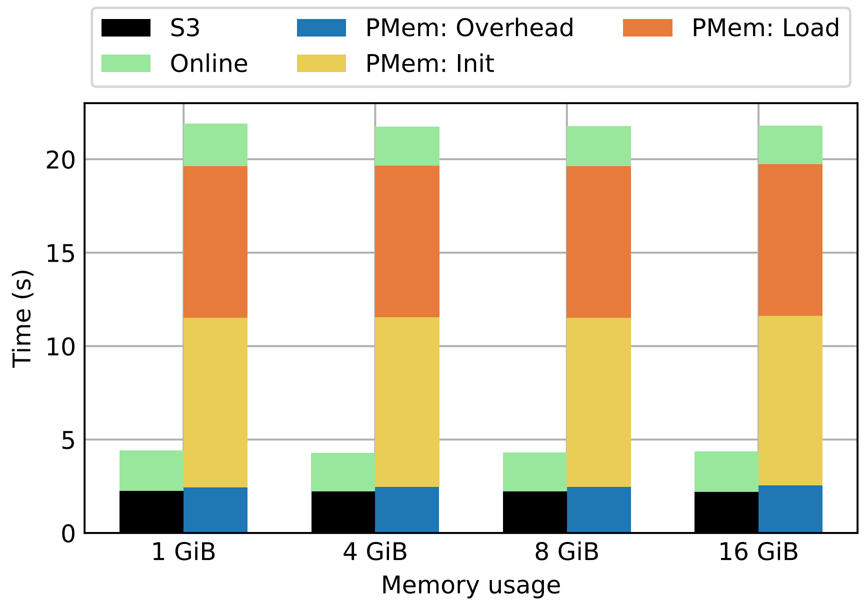
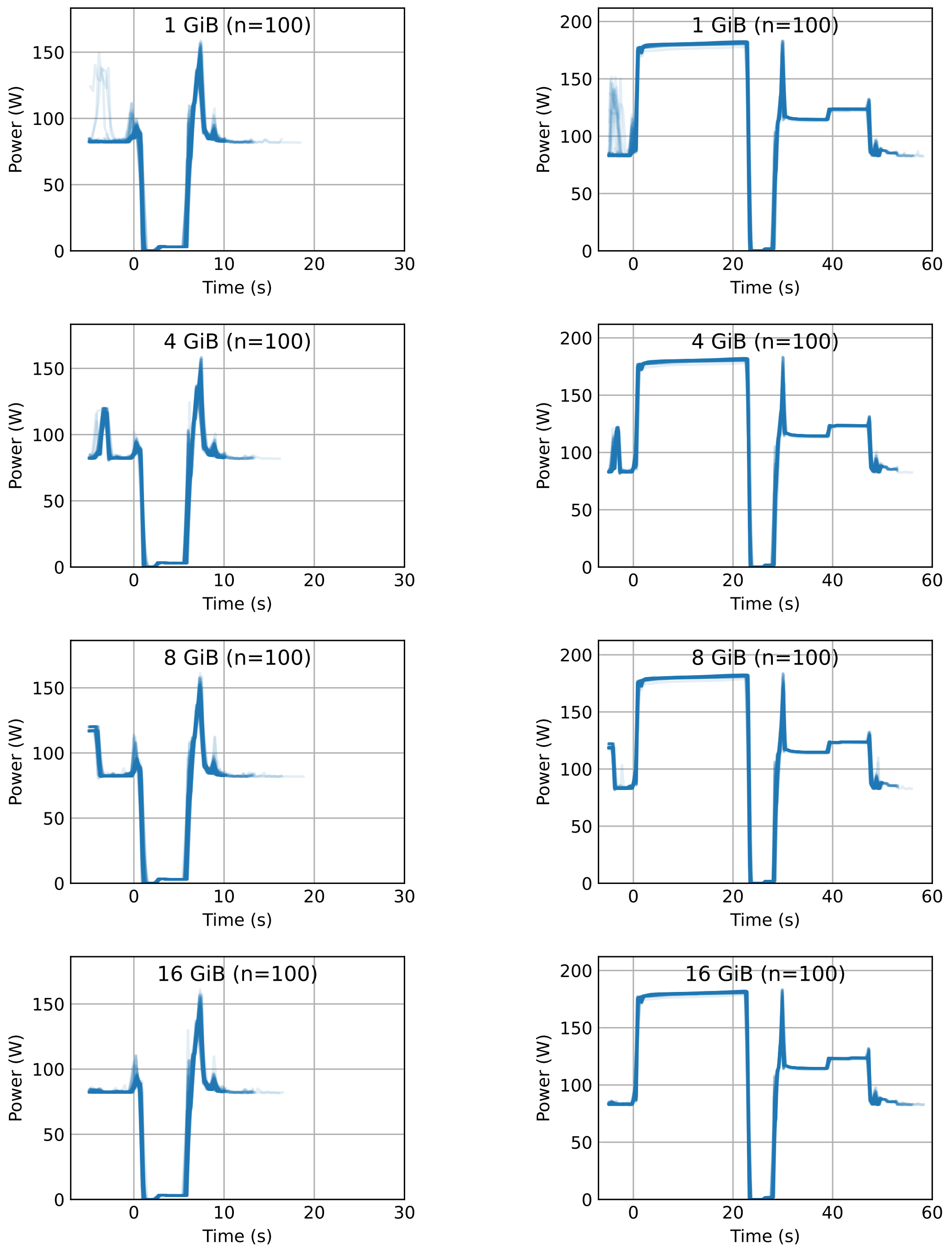


Figure 5.1: Wake latency components. Shown is the median of each contributing timespan. "S3" is the median wake latency of regular S3. "Online" represents additional time until the system responds to ping requests. "PMem: Init" is time spent initializing PMem, "PMem: Load" is time spent restoring RAM, and "PMem: Overhead" is any remaining part suspend-to-PMem's median wake latency.

on top of each other. Every run's power measurement curve was offset along the time axis such that time 0 aligns with the request to initiate suspend (T_{start}). It can be seen that the power measurement curves and timings are very consistent across runs for any given benchmark. As expected, unmodified S3 has only a slight peak in energy consumption before entering a low power state. In contrast, suspend-to-PMem spends a significant amount of time consuming much more power than while the system is idle. Yet, it reaches a lower power consumption than S3 while the device is suspended. When resuming, both S3 and suspend-to-PMem start with an initial peak in power consumption before leveling off. Suspend-to-PMem's power consumption at resume is further split into two clearly visible sections. As mentioned in Section 5.3, we assume that the first, slightly lower-power section represents memory training of the FPGA, while in the second section, the system is actively restoring system memory from PMem.



(a) Unmodified S3

(b) Suspend-to-PMem

Figure 5.2: Suspend power consumption over time for different memory usages. Time 0 marks the suspend request. S3 enters a low power state quickly. Suspend-to-PMem requires more energy initially, however, it achieves a better reduction.

Phase	**1 GiB**	**4 GiB**	**8 GiB**	**16 GiB**
Suspend	23 ± 15	26 ± 14	35 ± 15	24 ± 17
Resume	212 ± 34	205 ± 31	212 ± 30	204 ± 43
Sum	234 ± 37	232 ± 33	247 ± 31	228 ± 46

Table 5.6: Average energy consumption in Joule for different memory usages (regular S3 suspend).

Phase	**1 GiB**	**4 GiB**	**8 GiB**	**16 GiB**
Suspend	4024 ± 29	4031 ± 27	4048 ± 25	4000 ± 24
Resume	2289 ± 41	2293 ± 35	2291 ± 36	2301 ± 33
Sum	6313 ± 53	6325 ± 46	6339 ± 46	6301 ± 42

Table 5.7: Average energy consumption in Joule for different memory usages (suspend-to-PMem).

We also computed the total energy consumption during the suspend and resume phases of each run. Tables 5.6 and 5.7 show the averages of these values for regular S3 and suspend-to-PMem, respectively, along with their sums. For suspend, the time between the initiation of suspend and reaching power off was used for integration. For resume, the time between pressing the power button and reaching kernel resume was used. A few observations can be made about these results. For one, the energy consumed during S3 resume is about one order of magnitude higher than during S3 suspend, while suspend-to-PMem actually consumes less energy during resume than during suspend. Still, suspend-to-PMem consumes between $25.67\times$ and $27.58\times$ more energy than regular S3 in total.

Tables 5.7 and 5.8 show the approximate suspend power consumption for both modes. These results were computed by integrating the power consumption during the suspended state and dividing by the length of that interval. The beginning of suspend is marked by the time at which the system is powered off, while the end is marked by the time at which the power button is pressed to initiate resume. To remove artifacts caused by imprecise timings, only the middle 50% of the interval were used for the calculation. Looking at the results, it can be seen that regular S3 consumes, on average, between $2.03\,\mathrm{W}$ and $2.19\,\mathrm{W}$ while suspended. Suspend-to-PMem consumes much less power during the same phase, at $0.23\,\mathrm{W}$ to $0.33\,\mathrm{W}$.

Figure 5.3 shows the energy consumption over time for both modes of suspend. For short suspend durations, regular S3 has a lower total energy consumption than suspend-to-PMem because the energy cost of entering and exiting from suspend

1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
2.03 ± 0.18	2.19 ± 0.17	2.12 ± 0.14	2.08 ± 0.23

Table 5.8: Average power in Watts while suspended for different memory usages (regular S3 suspend).

1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
0.33 ± 0.10	0.30 ± 0.09	0.31 ± 0.10	0.23 ± 0.07

Table 5.9: Average power in Watts while suspended for different memory usages (suspend-to-PMem).

is much lower. However, suspend-to-PMem has a lower power consumption during suspend. From this, we can calculate the break-even point for each pair of benchmarks, where a pair is just two benchmarks with equal memory usage, one for regular S3 and one for suspend-to-PMem. This results in a break-even point ranging from $54\,\text{min}$ to $60\,\text{min}$. In other words, it is safe to assume that for a suspend duration of more than one hour, suspend-to-PMem will be more energy efficient than S3.

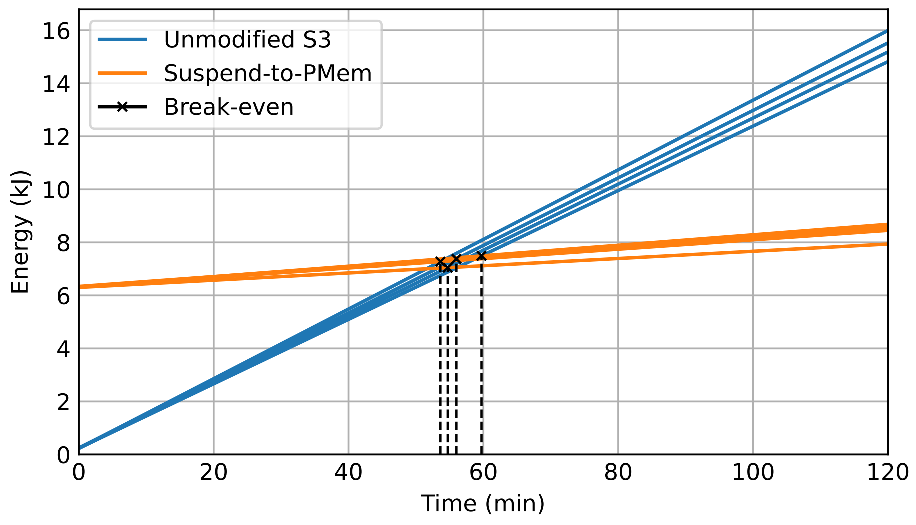


Figure 5.3: Energy consumption over time for different memory usages. The break-even point is marked for each pair of benchmarks. The break-even point is the time at which the energy consumption of suspend-to-PMem is equal to that of regular S3.

5.5 Suspend Latency

We consider the suspend latency, that is, the time between the suspend request and system power off, as a secondary performance characteristic. A device is usually suspended while not in use, and as such, its state might not be as important as a device that is needed again, and is therefore resuming. Still, it is possible that the device is told to suspend just before it is needed again; in that situation, a long time spent suspending would be detrimental, as this will further delay the time until a successful resume.

Results. We computed the suspend latency as $\Delta t_s = T_{\text{poweroff}} - T_{\text{start}}$. For suspend-to-PMem, we could also measure the time required to copy memory contents into PMem, denoted as Δt_{store}. As storing memory is one part of the full resume process, it follows that $\Delta t_s > \Delta t_{\text{store}}$.

	1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
Δt_s	0.63 ± 0.02	0.63 ± 0.02	0.63 ± 0.02	0.62 ± 0.02

Table 5.10: Suspend times for regular S3. Shown are the times between requesting suspend and power off (Δt_s) for different memory usages.

	1 GiB	**4 GiB**	**8 GiB**	**16 GiB**
Δt_s	22.94 ± 0.06	23.02 ± 0.04	23.04 ± 0.07	22.87 ± 0.03
Δt_{store}	22.32 ± 0.05	22.39 ± 0.01	22.42 ± 0.07	22.24 ± 0.02

Table 5.11: Suspend times for suspend-to-PMem. Shown are the times between requesting suspend and power off (Δt_s), as well as the time component of preserving RAM (Δt_{store}) for different memory usages.

Tables 5.10 and 5.11 show the suspend latency for both modes of suspend. The timings are nearly identical for all memory usages, though a slowdown between $36.63\times$ and $36.86\times$ can be observed for suspend-to-PMem as compared to regular S3. Δt_{store} contributes majorly to this difference, with an average 97.27% of the total suspend latency spent on PMem writes.

5.6 Discussion

The goal of this thesis was to design and implement a novel mode for system suspend using persistent memory. By storing the contents of main memory in PMem while suspended, the system can be powered off entirely, reducing the overall energy consumption compared to traditional S3 suspend.

As described in Section 3.1, we consider three key requirements for a successful implementation of suspend-to-PMem: reliability, low wake latency, and low power consumption. Additionally, aspects such as the suspend latency and transparency to the operating system can be evaluated. Based on the experimental results and the implementation at hand, we believe suspend-to-PMem to be a viable alternative to existing modes of system suspend, and consider our implementation to be successful according to the requirements.

In Section 5.2, we evaluated the reliability of suspend-to-PMem in comparison to S3. While the percentage of successful benchmark runs is slightly reduced for suspend-to-PMem, regular S3 did not achieve full reliability, either. This is a surprising result, as we would expect S3 to be very well-tested on all system components, including the CPU, mainboard, firmware, and operating system, due to its prevalence. Hence, there may be other factors at play, such as a combination of system components that do not integrate properly. One must also consider that the usage of an intermediate device, more specifically the FPGA, as an interface to Optane introduces substantial complexity in terms of configuring PCIe and initiating data transfers. We believe that the reliability of suspend-to-PMem may be improved by using a more direct connection between the CPU and persistent memory.

The wake latency, i.e., the time required to resume from a suspended state, is clearly dominated by the FPGA initialization and PMem reads, as previously shown by Figure 5.1. The FPGA's initialization time is mostly due to a "memory training" step. If it were possible to avoid memory training whenever the system is powered on, the wake latency could be reduced by as much as 46%. There is also clearly some overhead present for restoring DRAM contents from PMem. In Section 2.5, we briefly discussed Optane's performance, citing a read bandwidth of up to $6.6\,\mathrm{GB/s}$ for a single Optane DIMM with a capacity of $256\,\mathrm{GB}$. Assuming a similar read bandwidth for our $128\,\mathrm{GB}$ capacity DIMM, a $32\,\mathrm{GiB}$ read is expected to take $5.2\,\mathrm{s}$. As shown in Table 5.5, we observed load times of $8.11\,\mathrm{s}$, i.e., a 55% increase over the theoretical duration; however, the amount read is slightly less than $32\,\mathrm{GiB}$ because firmware memory regions are not restored. Still, the overall wake latency is consistently lower than $20\,\mathrm{s}$, which is likely sufficient for many applications.

The power consumption, as evaluated in Section 5.4, is where suspend-to-PMem truly shines. While suspended, regular S3 consumes, on average, between

2.03 W and 2.19 W. In contrast, suspend-to-PMem consumes only 0.23 W to 0.33 W. Unfortunately, the energy requirements associated with entering and exiting suspend are much higher in the case of suspend-to-PMem. This is, however, entirely offset when the break-even point, shown in Figure 5.3, is reached; for a suspend duration exceeding just one hour, suspend-to-PMem consumes less total energy than S3. There is potential to improve this even further by avoiding memory training and reducing data transfer overheads. If possible, the FPGA may be eliminated entirely from the design to limit the energy overhead of PMem to just Optane. Additionally, Table 5.7 has shown that writing to PMem is the main contributor to suspend-to-PMem's energy consumption. Our current implementation copies the entire DRAM contents, even including unused regions. By parsing the page tables when entering suspend, it may be possible to avoid storing unused regions, reducing the amount of data written and thereby improving the total energy consumption.

We can estimate the improvements in break-even times based on these ideas. Figure 5.2 shows that the power consumption is constant during suspend entry and nearly constant during resume. Therefore, we can compute an approximate reduction in energy consumption by considering the reduction in suspend and resume latencies. First, avoiding memory training saves 9.06 s according to Table 5.5. Based on Table 5.7, this reduces the total energy consumption by up to 1063 J, resulting in a break-even point between 44 min and 49 min. Second, if we assume zero-overhead transfer, a 32 GiB write to Optane would take just 14.94 s [49], while a 32 GiB read would take just 5.2 s as discussed previously. This optimization alone would result in a break-even point between 39 min and 44 min. Third, we may consider parsing page tables to reduce the amount of data written and read. At 25% memory usage, this optimization alone would result in a break-even point between 15 min and 17 min. Now, if all optimizations were combined, we would expect to see a break-even point of 31 min to 35 min at 100% memory usage, or only 12 min to 13 min at 25% memory usage. With all optimizations, the overall wake latency may be as low as 7.65 s at 100% memory usage or 3.75 s at 25% memory usage. This represents slowdowns of approximately 3.5× and 1.7×, respectively, when compared to regular S3.

We also provided some further evaluation criteria in Section 3.1. The suspend latency, for which experimental results were provided in Section 5.5, is consistently at around 23 s for our implementation. This represents a nearly 37× slowdown compared to regular S3. However, we believe this to be an acceptable compromise, as there is little difference between a system currently entering suspend and a system that is already suspended from the perspective of the system's operator. Furthermore, suspend-to-PMem offers much better resilience against power loss compared to traditional S3. As PMem does not need to be continuously powered, there is no data loss in case of a power outage while the system is suspended, unlike with regular S3, which requires an uninterrupted supply of power to DRAM. Finally, our implementation achieves good transparency to the operating system, which is entirely unaware of the suspend mode. Two minor modifications of the Linux kernel were described in Section 4.7, but these are only needed to correct problems with the FPGA architecture. The FPGA incorrectly reports its capability to be power-managed via PCIe and exposes too many PCIe virtual functions to fit within the memory region reserved by Coreboot. Any aspect concerning the suspend itself is achieved solely through additions in firmware.

Chapter 6

Conclusion

Suspending computer systems while they are not utilized can reduce the overall energy consumption over the system's lifetime. This is important for data centers and private consumers alike, as it allows for cost savings and may lead to a reduced environmental impact. The time required to wake a device from suspend is also important to avoid a performance degradation.

In this thesis, we have proposed a novel suspend mechanism called suspend-to-PMem. When entering suspend, the contents of main memory are copied to a persistent memory device based on Intel Optane Persistent Memory, and the system is fully powered off. A system suspended in this mode resumes by booting, but instead of loading a new instance of the operating system, the previous system state is restored from Optane.

We implemented this design as part of the open-source Coreboot firmware, using an FPGA-based PCI Express device to interface with Optane. To this end, we integrated a Linux device driver into the firmware layer, adapting it to a vastly different execution context. The device setup at the PCIe layer is implemented using low-level PCIe primitives. As part of our work, we further developed methods for automatically building and flashing Coreboot to a device, and established processes to overcome the constraints in debugging system firmware.

By evaluating the implementation on real hardware, we were able to show that suspend-to-PMem is a viable suspend strategy. When considering the energy consumption, this initial implementation already consumes less energy than traditional S3 after one hour in suspend. We found that there is significant potential for future improvements regarding power consumption, and also wake latency, by implementing suspend-to-PMem without an intermediate device.

6.1 Future Work

To conclude this thesis, we want to provide some ideas to further extend suspend-to-PMem and improve its performance.

Integrating directly with Optane. In Section 5.6, we have already stated that much of the performance overhead is caused by using an FPGA as the interface to Optane. For instance, the FPGA has to perform memory training on every resume. It also increases the data transfer latency [24]. Along with the simple presence of an additional device in the system, these aspects currently result in a significant energy consumption for entering and exiting suspend. However, the FPGA served an important role in our work, as it enabled the use of Optane on consumer hardware. Furthermore, Coreboot only has limited support for 64-bit execution, such that memory accesses are mostly constrained to the lower 4 GiB. This did not cause problems for us, as the FPGA performs all data transfers between DRAM and Optane on behalf of the CPU, and is capable of accessing the full 64-bit address space.

Copying used pages only. Our current resume implementation fully restores every non-firmware memory region. As a result, the wake latency is effectively the same across different memory usages, as shown in Section 5.3. We believe that suspend-to-PMem's wake latency can be substantially improved by parsing the operating system's page tables to only restore pages that are in use. When this parsing step is performed at suspend entry, the suspend latency also decreases, as unused pages need not be written to PMem in the first place. Finally, as shown by Table 5.7, writing to PMem is the main contributor to suspend-to-PMem's energy consumption. Reducing the amount written may lead to a much earlier break-even point when comparing suspend-to-PMem's overall energy consumption to that of S3.

On-demand paging. In Section 2.3, we already looked at work by Ho et al. [14] related to restoring only "urgent" pages at resume. Essentially, the memory pages can be further categorized into three categories: disk-backed and clean, non-disk-backed and urgent, non-disk-backed and non-urgent. Clean, disk-backed pages do not need to be restored at all [27]. Urgent pages such as kernel memory would be restored exactly like in our current implementation. Non-urgent pages, such as pages of userspace applications, can be restored on-demand by the OS. This could be achieved by marking these pages as backed by the PMem and exposing it as a storage device to the OS. They would then only be restored once a page fault occurs, decreasing the initial wake latency.

Suspend-to-idle. Operating systems may use a different suspend mode called suspend-to-idle (S2I) that is almost entirely software-based, giving them more control compared to ACPI-based sleep [18]. We did not investigate S2I as part of this thesis as it relates only minimally to system firmware. With S2I, the OS sets up address range monitoring, then executes an `MWAIT` instruction to allow the processor to enter a low-power state. It resumes processing when some predefined event occurs, such as an interrupt, or on a write to the monitored address range [21, vol. 2B, ch. 4, pp. 160-162]. The OS can use S2I to emulate S3 sleep, in which case the outward behavior would appear similar, but suspend-to-PMem would not be triggered.

We believe that suspend-to-PMem could also be implemented as part of an operating system, allowing for its use alongside S2I-based sleep. Notably, the OS cannot power off the entirety of DRAM when using such a modified S2I design, as that would necessitate at least a partial boot to resume. Lee et al. [26] have recently explored a mechanism by which the OS is able to power-manage DRAM sub-arrays individually. For S2I-based suspend-to-PMem implemented entirely within the OS, a viable strategy may entail offloading as many memory pages as possible onto PMem when entering suspend, then powering off unused parts of DRAM. Any pages that must stay present in DRAM should be consolidated such that the minimum amount of sub-arrays has to be supplied with power.

Bibliography

[1] 3mdeb Sp z o.o. Dasharo homepage. URL: https://www.dasharo.com/. Accessed: 2023-09-30.

[2] 3mdeb Sp z o.o. Supported hardware: Overview. URL: https://docs.dasharo.com/variants/overview/. Accessed: 2023-09-30.

[3] The Coreboot Project 3mdeb Sp z o.o. Release msi_ms7d25_v1.1.1 - Dasharo/coreboot. URL: https://github.com/Dasharo/coreboot/releases/tag/msi_ms7d25_v1.1.1. Accessed: 2023-10-09.

[4] Ars Technica. Intel's first Optane SSD: 375GB that you can also use as RAM. URL: https://arstechnica.com/information-technology/2017/03/intels-first-optane-ssd-375gb-that-you-can-also-use-as-ram/. Accessed: 2023-06-22.

[5] CLEAResult Consulting Inc. What is 80 PLUS? URL: https://www.clearesult.com/80plus/. Accessed: 2023-10-02.

[6] Intel Corporation. *Enhanced Host Controller Interface Specification for Universal Serial Bus*, March 2002. URL: https://www.intel.com/content/www/us/en/products/docs/io/universal-serial-bus/ehci-specification-for-usb.html. Accessed: 2023-09-27.

[7] Maxim Devaev. PiKVM - Open and inexpensive DIY IP-KVM on Raspberry Pi. URL: https://pikvm.org/. Accessed: 2023-10-02.

[8] SeaBIOS developers. SeaBIOS. URL: https://www.seabios.org/SeaBIOS. Accessed: 2023-10-02.

[9] DMTF. *System Management BIOS (SMBIOS) Reference Specification (Version 3.7.0)*, July 2023. URL: https://www.dmtf.org/sites/default/files/standards/documents/DSP0134_3.7.0.pdf. Accessed: 2023-10-02.

[10] flashrom developers. flashrom documentation. URL: https://www.flashrom.org/. Accessed: 2023-10-08.

[11] Free Software Foundation, Inc. GNU GRUB Manual 2.06. URL: `https://www.gnu.org/software/grub/manual/grub/grub.html`. Accessed: 2023-10-02.

[12] Good Will Instrument Co., Ltd. GPM-8213 Digital Power Meter: User Manual. URL: `https://www.gwinstek.com/en-US/products/downloadSeriesDownNew/11551/1553`. Accessed: 2023-10-02.

[13] Google. Developer Information for Chrome OS Devices. URL: `https://www.chromium.org/chromium-os/developer-information-for-chrome-os-devices/`. Accessed: 2023-10-01.

[14] 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`.

[15] Intel Corporation. EDK II Minimum Platform Specification. URL: `https://tianocore-docs.github.io/edk2-MinimumPlatformSpecification/draft/`. Accessed: 2023-10-01.

[16] Intel Corporation. Intel® Optane™ Persistent Memory Data Sheet. URL: `https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html`. Accessed: 2023-06-23.

[17] Intel Corporation. Intel®Stratix® 10 DX FPGA. URL: `https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/dx.html`. Accessed: 2023-10-04.

[18] Intel Corporation. The Linux Kernel: System Sleep States. URL: `https://docs.kernel.org/admin-guide/pm/sleep-states.html`. Accessed: 2023-06-27.

[19] Intel Corporation. What is TianoCore? URL: `https://www.tianocore.org/`. Accessed: 2023-10-01.

[20] Intel Corporation. Intel Optane DC Persistent Memory Readies for Widespread Deployment. URL: `https://newsroom.intel.de/news/intel-optane-dc-persistent-memory-readies-for-widespread-deployment/#gs.gyg13n`, October 2018.

[21] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, March 2023.

[22] The kernel development community. efivars - a (U)EFI variable filesystem. URL: `https://docs.kernel.org/filesystems/efivarfs.html`. Accessed: 2023-09-27.

[23] The kernel development community. Linux kernel stable tree v6.1.33. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v6.1.33`. Accessed: 2023-10-09.

[24] Yussuf Khalil. FPGA-accelerated non-volatile memory access. Master's thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October 2022.

[25] Hyojeen Kim, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Building fully functional instant on/off systems by making use of non-volatile RAM. In *2011 IEEE International Conference on Consumer Electronics (ICCE)*, pages 675–676, 2011.

[26] Seunghak Lee, Ki-Dong Kang, Hwanjun Lee, Hyungwon Park, Younghoon Son, Nam Sung Kim, and Daehoon Kim. Greendimm: Os-assisted dram power management for dram with a sub-array granularity power-down state. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 131–142, New York, NY, USA, 2021. Association for Computing Machinery. `https://doi.org/10.1145/3466752.3480089`.

[27] 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`.

[28] Yinghai Lu, Li-Ta Lo, Gregory R. Watson, and Ronald G. Minnich. CAR: Using Cache as RAM in LinuxBIOS. Technical report, Advanced Computing Laboratory, Los Alamos National Laboratory, 2006.

[29] PCI-SIG. *PCI Express® Base Specification Revision 4.0 Version 1.0*, September 2017.

[30] Martin Roth (The Coreboot Project). Announcing coreboot release 4.19. URL: `https://blogs.coreboot.org/blog/2023/01/28/announcing-coreboot-release-4-19/`, January 2023. Accessed: 2023-09-25.

[31] Zhang Rui. Linux ACPI Custom Control Method How To. URL: `https://docs.kernel.org/firmware-guide/acpi/method-customizing.html`. Accessed: 2023-10-07.

[32] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts, 9th Edition*. Wiley, 2012. `https://books.google.de/books?id=9VMcAAAAQBAJ`.

[33] Shihao Song, Anup Das, Onur Mutlu, and Nagarajan Kandasamy. Improving phase change memory performance with data content aware access. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2020, pages 30–47, New York, NY, USA, 2020. Association for Computing Machinery. `https://doi.org/10.1145/3381898.3397210`.

[34] Jiming Sun, Marc Jones, Stefan Reinauer, and Vincent Zimmer. *Building coreboot with Intel FSP*, pages 55–95. Apress, Berkeley, CA, 2015. `https://doi.org/10.1007/978-1-4842-0070-4_4`.

[35] The Coreboot Project. Coreboot 4.21 documentation: Coreboot architecture. URL: `https://doc.coreboot.org/getting_started/architecture.html`. Accessed: 2023-09-25.

[36] The Coreboot Project. Coreboot 4.21 documentation: Distributions. URL: `https://doc.coreboot.org/distributions.html`. Accessed: 2023-10-01.

[37] The Coreboot Project. Coreboot 4.21 documentation: Driver Devicetree Entries. URL: `https://doc.coreboot.org/drivers/dt_entries.html`. Accessed: 2023-10-05.

[38] The Coreboot Project. Coreboot 4.21 documentation: libgfxinit - Native Graphics Initialization. URL: `https://doc.coreboot.org/gfx/libgfxinit.html`. Accessed: 2023-10-07.

[39] The Coreboot Project. Coreboot 4.21 documentation: Payloads. URL: `https://doc.coreboot.org/payloads.html`. Accessed: 2023-09-25.

[40] The Coreboot Project. Coreboot for end users. URL: `https://www.coreboot.org/users.html`. Accessed: 2023-06-28.

[41] The Coreboot Project. Coreboot wiki: Cbmem console. URL: `https://www.coreboot.org/Cbmem_console`. Accessed: 2023-10-03.

[42] The Coreboot Project. Coreboot wiki: Console and outputs. URL: `https://www.coreboot.org/Console_and_outputs`. Accessed: 2023-09-27.

[43] The European Commission. Regulation: Ecodesign regulation for computers and servers (EU) No 617/2013. Official Journal of the European Union, URL: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32013R0617`, 2018.

[44] Ubuntu Wiki. Kernel Reference: S3. URL: `https://wiki.ubuntu.com/Kernel/Reference/S3`. Accessed: 2023-10-04.

[45] UEFI Forum, Inc. *ACPI Specification 6.5*. URL: `https://uefi.org/specs/ACPI/6.5/`. Accessed: 2023-06-28.

[46] UEFI Forum, Inc. *UEFI Specification 2.10*. URL: `https://uefi.org/specs/UEFI/2.10/`. Accessed: 2023-09-27.

[47] Yefu Wang and Xiaorui Wang. Performance-controlled server consolidation for virtualized data centers with multi-tier applications. *Sustainable Computing: Informatics and Systems*, 4(1):52–65, 2014. `https://www.sciencedirect.com/science/article/pii/S2210537914000031`.

[48] Sam Likun Xi, Marisabel Guevara, Jared Nelson, Patrick Pensabene, and Benjamin C Lee. Understanding the critical path in power state transition latencies. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 317–322. IEEE, 2013.

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association. `https://www.usenix.org/conference/fast20/presentation/yang`.

[50] 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.