Karlsruhe Institute of Technology

# Whole Process Persistence with coreboot

Bachelor's Thesis
submitted by

Max Streicher

to the KIT Department of Informatics

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

September 18, 2023 – February 15, 2024

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

Karlsruhe, February 15, 2024

# Abstract

This thesis introduces a novel approach to preserving data processed and stored by software applications, addressing the risk posed by computer crashes that can erase the state of applications and lead to data loss. The concept of making applications persistent across system crashes without having to change or recompile them, termed Whole Process Persistence (WPP), is introduced in "Zhuque: Failure is not an Option, it's an Exception", where the authors introduce a runtime relying on persistent memory in the system [14]. Motivated by the need to ensure data integrity against unexpected system failures, this thesis explores the feasibility of WPP in environments lacking native persistent memory. We developed a system to achieve WPP in the absence of traditional persistent memory solutions, such as Intel's discontinued Optane brand, by modifying the Linux kernel and firmware of the system [27]. This adaptation enables the saving of PTE, PCB, and virtual memory regions onto an FPGA device accessible via PCIe when the system experiences a power loss.

By modifying system components, the thesis outlines a methodology for persisting the entire state of a process, aiming to mitigate the impact of crashes. The practical implementation of WPP, tested on a system configured with a 12th generation Intel x86 CPU demonstrates the technical viability of this approach.

The implementation reveals a significant performance overhead, with synthetic load tests indicating more than 35 times increase in execution time and a worst time slowdown of more than 90% on other processes running on the same machine. Real world benchmarks are depending heavily on how much memory a process allocates, simple redis benchmarks are not experiencing a noticeable slowdown. The process of persisting is able to copy $1.3\,\mathrm{GB/s}$ of allocated memory to the FPGA device, necessitating the need for an external UPS.

# Contents

# Chapter 1

# Introduction

Today, many critical workloads are processed by computers. These include, but are not limited to banking, managing data in the health sector, and powering consumer services [1]. While many client devices feature a battery, servers do not. There are several approaches to mitigate data loss when a system crashes due to power loss. Solutions include general patterns like atomic operations up to "whole system persistence" enabled by persistent memory [22]. Persistent memory is a new form of system memory closing the gap between high-performance volatile memory and persistent storage [21]. This type of memory makes it possible for applications to transparently persist all their memory contents thus preventing data loss. The primary implementation of persistent memory is Intel Optane, which is available in DIMM format to replace traditional DRAM [27]. While possible, swapping out the complete memory with NVDIMMs comes with a performance penalty.

System memory that consists of traditional and persistent memory can only be used in a meaningful way by software that was modified for this memory configuration. Hodkins [14] et al. propose "whole process persistence" for systems with persistent memory. The proposed system tracks the process at runtime and places all private memory regions on persistent memory, making them crash-tolerant. While this approach only comes with minor performance penalties and does not need software changes, it needs persistent memory in the system. Since the only provider of persistent memory, Intel, discontinued its Optane product, approaches reliant on persistent memory are not fit for use in the future [27].

Achieving whole process persistence, where a process is persisted and then recreated in the same state it was in before a shutdown or power loss, involves preserving several properties of the process. This includes all memory allocated to the process, various kernel-tracked properties such as file descriptors, and the layout of the process's virtual memory space. Additionally, the CPU's state, including the program counter and registers, must also be saved.

In this thesis, we propose a system that tracks the state of a process like the

approach described in "Zhuque: Failure is Not an Option, it's an Exception [14]." In contrast to the runtime proposed in the paper, we implement whole process persistence without being reliant on persistent memory in the host system.

To accomplish whole process persistence, the kernel actively monitors and records the memory allocations of a process during its runtime, relaying this information to the system firmware. Upon receiving a simulated power loss signal, the process is immediately paused by halting all its threads, and further details from the Process Control Block (PCB) are transmitted to the firmware. Subsequently, the firmware takes responsibility for preserving all the acquired data by saving it to a storage device connected through Peripheral Component Interconnect Express (PCIe). In addition to this, the contents of all mapped memory pages are also stored on the same device, ensuring a snapshot of the process's state. After the system restarts, the operating system kernel can restart the previously persisted binary at the exact state it was stopped when the power loss signal was received.

### Overview

Chapter 2 of this thesis introduces foundational topics like system firmware, x86 System Management Mode (SMM), coreboot, Intel Optane Persistent Memory (PMem), the Field Programmable Gate Array (FPGA) device used to access Optane, and Linux's memory and process management, which we need to modify for our solution. Chapter 3 provides background on previous research in this area. In Chapter 4, we outline the system's requirements and describe our method for tracking process memory allocations and process recreation after power loss.

Chapter 5 details the implementation, focusing on modifications in the Linux kernel and system firmware. Chapter 6 evaluates the solution, discussing the test environment, challenges, limitations, and performance impacts.

In Chapter 7, we conclude the thesis, summarizing key findings and exploring potential research directions.

# Chapter 2

# Foundations

In this chapter, we establish the background information for this thesis. The chapter introduces system firmware, including interfaces like Unified Extensible Firmware Interface (UEFI) and Advanced Configuration and Power Interface (ACPI) emphasizing coreboot as an open-source implementation. We introduce the SMM of the x86 ISA, one example for non-volatile memory, *Intel Optane Persistent Memory*, and look into the memory and process subsystems of the Linux kernel that are the base for our modifications to the kernel.

## 2.1 System Firmware

On modern computers, the system firmware is a layer bridging the gap between device hardware and the operating system kernel. It is the first software component to be executed when a system boots and performs initialization of the mainboard, the CPU, and DRAM. In most cases, firmware resides on a flash chip on the mainboard. After the basic components are initialized the firmware executes a piece of software that implements one of the firmware interfaces Basic Input/Output System (BIOS) or UEFI [56].

The firmware not only initializes the hardware on startup but remains active and is needed to perform hardware-related tasks like power and thermal management. To communicate with the operating system, the ACPI standard is used [30, 52].

Since firmware hides away the differences of individual mainboard and CPU models, it has to be customized for each mainboard and CPU model. Most implementations are closed source which makes it infeasible to modify them [40].

For our modifications to the firmware, we need to have access to a chunk of system memory. To reserve a portion of memory for this, we modify the e820 parser in the kernel [47]. The e820 interface is included in the ACPI specification and is used to forward information about the system memory to the kernel [52]. Some parts of memory are not usable for the kernel because the regions are memory-mapped I/O devices or reserved by the firmware for internal use.

### x86 System Management Mode

The x86 and x86_64 ISAs specify several operating modes [16]. The processor starts in and executes the operating system in the *kernel mode* ring 0. Other processes not related to the kernel run in ring 3 which gives them no direct access to hardware resources. This allows the operating system to provide an abstraction layer over the hardware and prevents processes in the userspace from interfering with the kernel. While ring 0 is designed to be used by operating system kernels, it has to be possible for the system firmware to be invoked even after the initialization phase in the boot process is over. Therefore the x86 / x86_64 ISA specifies the SMM. The SMM is entered when a System Management Interrupt (SMI) occurs. While the hardware is being initialized, the firmware is able to specify when an SMI should be triggered. The SMM is completely transparent to the kernel, so tasks that run in SMM have to have a short runtime to prevent latency spikes in the kernel. When a SMI is triggered, the processor temporarily halts its current tasks, saving its state (processor's context), and switches to a distinct operating environment in a new address space. This shift activates the system management software executive, or SMI handler, which resides within a protected physical memory region known as SMRAM, where it is placed by the firmware at the beginning of the boot process. The SMI handler executes specialized tasks like power management for disk drives or monitors or suspending the system. Once these operations are complete, the handler issues a `RSM` ("resume") instruction [16]. This instruction reloads the processor's saved context and transitions it back to its previous operating mode (protected or real mode), allowing the interrupted application or operating system to continue where it left off.

### ACPI

SMM predates ACPI, originally handling low-level functions like thermal management and CPU performance directly in firmware [52]. ACPI introduced a shift, enabling the OS to manage power, thermal properties, and performance through ACPI Machine Language (AML) code provided by the firmware. This theoretically reduces the need for SMM in modern systems.

The division of responsibilities between firmware and OS is defined by the firmware developer. This can range from comprehensive management by the OS, guided by detailed AML code, to a minimal role where the OS prompts the firmware to handle events via SMIs. An example is the power button: initially, firmware handles shutdowns directly; after the OS takes over, it manages shutdown procedures based on ACPI signals, allowing for orderly shutdowns and system management [52].

## 2.2 Coreboot

Coreboot is an open-source system firmware implementation designed to perform platform initialization across various mainboards and CPU architectures. As the first code executed on a CPU after a system reset, coreboot configures the DRAM, and initializes connected devices, including PCIe devices. It starts the chipset (called Platform Controller Hub on Intel systems) along with all its attached devices and setups the ACPI table [17]. Adhering to the principle of separation of concerns, coreboot does not incorporate a bootloader or BIOS; rather, it is solely utilized for initializing the underlying hardware [34].

### Boot Stages

The boot sequence of coreboot is structured into several stages, each compiled as separate binaries. The sequence begins with the *bootblock*, responsible for setting up a C programming environment and implementing cache-as-RAM (CAR) for memory management.

Following the *bootblock*, the *verstage* initiates a secure boot process, establishing a root-of-trust essential for maintaining firmware integrity. Next, the *romstage* prepares the system for device initialization by setting up DRAM. This is succeeded by the *postcar* stage, which marks the transition from CAR to regular DRAM operation.

The *ramstage* is tasked with the main device initialization and configuring various system tables. Concluding the coreboot process, the payload is executed.

A coreboot payload is a software component that coreboot executes after it has initialized the hardware of the system. Unlike traditional BIOS or UEFI, which have a fixed set of functionalities, coreboot is designed to be minimal and modular, handing off control to a payload for specific tasks after the system is initialized.

Examples of coreboot payloads include *SeaBIOS* [7], which provides a traditional BIOS interface for booting operating systems, and TianoCore [51], an open-source implementation of UEFI that allows coreboot to boot systems requiring UEFI support. Other payloads like *GRUB*, a bootloader, can be used for directly

loading and managing different operating systems [6]. Dasharo, the distribution of coreboot used in this thesis, includes *TianoCore edk2* as payload [36].

### Logging

Coreboot offers multiple output channels for conveying logs to developers [35]. Given that we do not alter the firmware's boot stages, our interest lies exclusively in logs generated after the system has booted. The available tool, *cbmem*, enables us to display firmware logs in the Linux environment that are retrieved from a memory region that is shared with, and exclusively writable by, the firmware.

## 2.3 Intel Optane

Intel Optane is a non-volatile memory technology, distinct from traditional RAM and NVRAM. It is based on a non-volatile phase-change memory technology called 3DXpoint developed by Intel and Micron [13]. This technology is particularly interesting because of its fast, byte-addressable and persistent nature.

Regarding its applications, Optane can be used in two primary ways:

- As standard NVMe PCIe SSDs: In this format, Optane SSDs were broadly compatible with various systems, not requiring special drivers or CPU support.
- As memory or onboard acceleration devices: Optane could also function as non-volatile main memory (NVDIMM) or for caching and accelerating tasks. This application, however, necessitates hardware that explicitly supports Optane [15]. Within the memory/storage hierarchy, Optane occupies a unique position. It does not fit into the traditional categories of RAM or flash SSDs; instead, it forms its distinct category within this pyramid [21].

In 2022, Intel announced the discontinuation of Optane persistent memory [27].

### FPGA Device

For this thesis, we require a storage device that is accessible from SMM and with a capacity that at least matches the size of our main memory. To minimize complexity for the code running in SMM, the device should be byte-addressable and offer low latency. We use a PCIe device containing Optane non-volatile memory based on an Intel FPGA accelerator, designed to address the performance challenges of Intel Optane Persistent Memory [53]. The FPGA device supports a write bandwidth of up $12\,\mathrm{GiB/s}$. This device empowers us to fully utilize the bandwidth of the

PMem with just a single CPU thread, thanks to its support for "asynchronous copy offloading." This feature allows the CPU to initiate subsequent commands to copy memory sections to the FPGA without waiting for the completion of previous ones, thereby preventing CPU stalls that could occur due to numerous, yet small, page transfers.

### PCIe

The PCIe bus is a high-speed serial computer expansion bus standard. PCIe is used for connecting high-speed components in computers and servers, including graphics cards, solid-state drives (SSDs), and network interfaces.

It operates through serial connections with data transferred over point-to-point data lanes. These can be combined to create wider interfaces, enhancing bandwidth. It is scalable, with configurations from x1 to x32 lanes. PCIe 3, which we are using to connect to the FPGA device, is capable of transmitting $8\,\mathrm{Gbit/s}$ per lane.

Base Address Registers (BARs) are crucial components within the PCIe specification, serving as the mechanism through which a PCIe device communicates its memory and I/O requirements to the host system. When a PCIe device is added to a system, it must be configured to interact with the system's memory space. BARs are used by the device to inform the OS how much memory space it needs and at what address this space should be accessible. The system's BIOS or firmware typically handles the allocation of address space to each device during the system boot-up process.

While there are several types of BARs, we only use Memory Mapped I/O (MMIO) BARs. These are used for mapping device memory into the system's memory address space. MMIO BARs can specify whether the device requires a 32-bit or 64-bit address space.

## 2.4 Linux

In implementing this thesis, we have chosen to use the Linux kernel. It is well-documented and comes with support for a wide range of devices. Using Linux allows us to modify the kernel and implement changes in core components like the memory subsystem.

### Virtual Memory Management

In the realm of virtual memory management, on x86_64 architectures, memory is organized in a paged format. This eliminates the need for contiguous physical memory and allows for efficient and flexible memory utilization. Paging involves

the division of virtual memory into discrete blocks, known as pages. On x86_64 systems, the default settings set the size of a memory page to 4096 bytes, and the operating system maintains a page table to map these virtual pages to physical memory frames [16].

In the x86_64 architecture, the virtual memory is managed through a four-level page table structure as illustrated in 2.1. This structure includes the Page Global Directory (PGD), Page Upper Directory (PUD), Page Middle Directory (PMD), and Page Table Entries (PTE). The PGD, at the top level, points to PUDs, which further divide the virtual address space. PUDs point to PMDs, and PMDs lead to the PTEs, the final level where actual virtual-to-physical address mappings are stored [16].

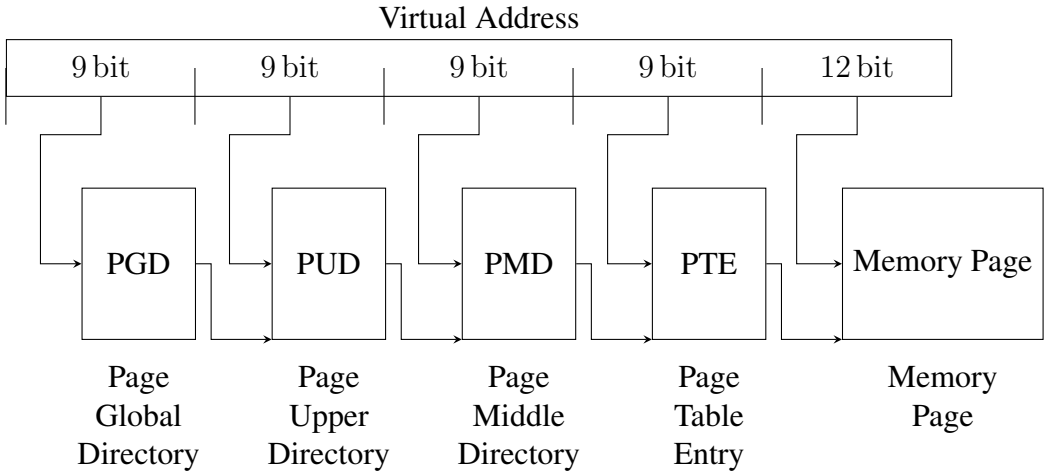


Figure 2.1: Representation of the x86_64 architecture's 4-level page table structure, showcasing the sequential mapping from a virtual address to physical memory. Each segment of the virtual address is mapped to its respective level in the page table.

In Linux, each process has a list of `vm_area_structs` which describe a contiguous memory area in the process address spaces [48]. This includes information like access permissions and whether the memory region is backed by a file and a pointer to the PGD of the process.

### Process Creation

To persist a running process, we need to access some of its data in addition to mapped memory pages. Also, to recreate a previously persisted process, we need to create a new process that we adjust according to the data we backed up.

The Linux kernel keeps track of processes using the `task_struct` data structure, which contains comprehensive information about each process, such as its state, memory usage, and identifiers [49].

Process creation in the kernel is primarily facilitated through two system calls: fork and clone [42]. The `fork()` system call creates a new process by duplicating the calling process, resulting in a child process that is an exact copy of the parent but with a unique process ID. However, `fork()` can be resource-intensive as it involves copying the entire memory space of the parent process.

To offer a more efficient and flexible alternative, the kernel provides the `clone()` system call. `clone()` is used for both thread and process creation, allowing specific control over shared resources between the parent and the child processes. This approach is particularly effective for creating lightweight threads as it reduces the overhead of process creation by sharing certain resources like the address space.

# Chapter 3

# Related Work

In this chapter, we highlight existing research that aligns with the themes of our thesis. This chapter reviews the Zhuque paper, which introduces a runtime for applications to achieve crash consistency, even if they are not inherently persistent memory-aware. While our thesis implements a similar system, our approach is not dependent on PMem.

We also examine eADR, a set of instructions in certain Intel CPUs that enable flushing of all modified data in caches to memory in the event of power loss.

Additionally, we highlight some emerging non-volatile memory systems to provide a comprehensive view of the current landscape in non-volatile memory systems.

## 3.1 Zhuque

Hodkins et al. [14] introduce Whole Process Persistence (WPP), a new programming model designed for systems with persistent caches, specifically addressing the challenges posed by PMem. In the paper, they focus on making it possible to profit from properties like byte-addressability while solving the challenges of having to rewrite software and losing performance.

WPP aims to simplify PMem programming by making the entire state of a process persistent. In the event of a power failure, this state can be reloaded, and execution resumes with an application-defined interrupt handler. The paper also discusses the Zhuque runtime, which provides WPP transparently by interposing on C system call bindings in userspace. This runtime requires minimal programmer effort and significantly outperforms existing PMem libraries by a factor of 5.24x for PMDK, 3.01x for Mnemosyne, 5.43x for Atlas, and 4.11x for Clobber-NVM.

The paper identifies limitations in previous PMem systems and proposes WPP as a solution that treats power failure as a recoverable exception. It showcases

the performance improvements of WPP over existing systems and its flexibility in supporting legacy programs and complex locking schemes.

## 3.2 Whole-System Persistence

Databases and key-value stores are predominantly maintaining all their data in main memory in server clusters potentially housing hundreds of terabytes of data. The recovery of the system from a backend storage becomes exceedingly time-consuming, ranging from minutes for a single server to several hours for an entire cluster.

The paper "Whole-System Persistence" proposes a technique of the same name as a solution for systems equipped with non-volatile memory, aiming to significantly reduce recovery times [31]. Whole-System Persistence (WSP) distinguishes itself by eliminating runtime overheads through a "flush on fail" strategy, where transient state data in processor registers and caches are flushed to non-volatile RAM (NVRAM) only upon detecting a power loss. This method leverages the residual energy from the system power supply to ensure the data flush can be finished even with power to the server lost.

Comparative analysis with existing solutions, such as Mnemosyne - which flushes caches upon each commit - reveals that WSP offers a substantial performance advantage. For instance, WSP outperforms Mnemosyne by more than 2.4 times in an OpenLDAP benchmark. Furthermore, WSP simplifies the failure recovery process, making a power outage appear akin to a system suspend/resume event, with complete restoration of the stack, heap, and thread context.

WSP addresses the issue of persistent objects' dependency on volatile ones by persisting the entire system state. Utilizing residual energy for data flushing, tests on their system demonstrated a capability to sustain power for at least 10ms after energy loss, with cache flushing requiring less than 5ms across all systems. The approach employs the `PWR_OK` signal from Advanced Technology Extended (ATX) specifications, monitored via a microcontroller, to detect power loss timely.

The exploration of "Process Persistence" is highlighted as a potential area for future work.

## 3.3 eADR

In the realm of persistent memory and its integration into system architectures, the concept of atomic durability is critical for ensuring crash consistency in applications. Traditional atomic durability techniques for PMem systems, which rely on

volatile cache, often lead to significant performance overhead. Intel's introduction of enhanced Asynchronous DRAM Refresh (eADR) for Optane PMem presents an opportunity to develop a more efficient system for atomic durability in PMem environments [20].

The paper "Efficient Atomic Durability on eADR-Enabled Persistent Memory" discusses a technique named LOAD, which is a low-overhead atomic durability method built upon eADR [55]. LOAD is designed to leverage the memory hierarchy effectively. It introduces two key components: Transaction-aware Cache (TaC) and Device-Friendly Logging (DFL). TaC utilizes the memory hierarchy to move older, yet valid data from higher-level caches (like L1) to lower levels (such as L2), thereby retaining old version data in caches for crash recovery. On the other hand, DFL is responsible for recording necessary old version data into logs. This ensures that transactional data can be atomically moved from the last-level cache to PMem.

In terms of data movement, the path typically follows from the core to L1, then to L2, LLC (or L3), WPQ (Write Pending Queue), and finally to the DIMM [19]. While ADR ensures that data in the WPQ is saved to the DIMM during power loss, eADR expands this to include the entire memory subsystem within the CPU, encompassing the whole cache hierarchy. This includes specific instructions to flush caches and implement fence instructions.

However, a challenge arises with the performance overhead incurred when flushing cache lines. LOAD addresses this by implementing DFL. DFL logs data written to PMem from the LLC. In the event of a reboot, DFL allows for partially updated data in PMem to be rolled back to a valid state using the logs. Notably, LOAD achieves this with only a 1% performance overhead [55].

## 3.4 Emerging Nonvolatile Memory

Focusing on emerging memory technologies, the discussion extends beyond 3D XPoint, highlighting that it is not the only persistent memory solution available. MRAM (Magnetoresistive RAM) is another notable technology in this space [54]. It is a type of RAM based on magnetic effects and is commonly used in microcontrollers where non-volatile properties are essential. Looking ahead, MRAM could potentially be used in CPU caches [23].

A recent development in this field is SOT-MRAM (Spin Orbit Transfer Magnetoresistive RAM) [12]. This technology is a successor to STT-MRAM (Spin-Transfer Torque MRAM) but is not yet commercially available. SOT-MRAM is recognized for being faster and more energy-efficient than its predecessor.

Despite the discontinuation of Optane, there continues to be active development and innovation in non-volatile memory (NVM) technologies, presenting alternatives for future research.

# Chapter 4

# Design

In this chapter, we present the system requirements, both functional and non-functional. The chapter discusses the design aspects at a high level, particularly focusing on how memory allocations are tracked and persisted. Additionally, we explore the design strategy behind the recreation of processes, providing insights into the methodologies and considerations involved in accurately restoring a process to its previous state post-recovery. The actual details on our implementation follow in Chapter 5.

## 4.1 Requirements

The design of the whole process persistence, integrated into coreboot, focuses on three key requirements: **accurately restoring the process to its original state**, ensuring **low performance impact during runtime**, and **minimizing energy usage after receiving the power loss signal**.

To restore a process as closely as possible to its original state, the system has to capture a snapshot of the process while it is running. This includes storing the memory region layout, encompassing file backings and other details crucial for the process's environment. Additionally, it involves capturing the entire state of the process stored in the PCB, including PID, CPU state with all registers, threads, open files, permissions, and all physical page frames.

The recreation phase involves recreating the process using the same binary as the persisted one reconstructing all memory regions, including shared libraries and open files. The process's pages are restored to their previous state, and the process information, such as PID and general settings, are set to mirror the persisted process. The newly created process's execution state, including the program counter and registers, is altered to match where the previously persisted process was at the moment of persistence.

The focus of this design is to minimize the impact on runtime performance. This is achieved by limiting the operations to storing or removing page mappings and memory regions in RAM managed by the firmware, avoiding any heavy computations or PCIe device access. This approach ensures that the regular operation of the system remains largely unaffected while the process persistence activities are carried out.

When a power loss signal is received, the firmware writes all memory pages and other captured data like memory mappings, memory regions, and information from the PCB to the FPGA introduced in Chapter 2. The final step of writing an `OK` bit to the FPGA marks the completion of the persistence process.

## 4.2 Process Tracking

The tracking of a process during its runtime begins with a specialized `starter` binary that receives an executable as an argument. This binary clones itself using specific arguments, adapting the standard process creation mechanism to tell the kernel that the newly started binary should be tracked during runtime and persisted on a given signal.

The Linux kernel is modified to include a `should_be_persisted` flag in the `task_struct` [48], allowing the kernel to identify the process that it should persist. When a process with this flag is started, the kernel communicates this event to firmware, which then resets its state in preparation for the new process.

The specialized `starter` binary executes the given executable using the `execve` system call [44]. The kernel tracks all memory region creations and deletions associated with the process, including details like virtual address, length, flags, backing file, and protection levels, and copies this information to the firmware.

When new page table entries are created during memory access, the kernel communicates the virtual and physical address combinations to the firmware. This happens when the program is accessing an address that does not already have a Page Table Entry (PTE) leading to a page fault. Similarly, the deletion of PTEs is communicated to the firmware.

Since we are tracking PTEs, we are not capturing memory pages that are swapped out by the operating system. It is possible to persist these pages, but this would increase the complexity of the implementation and add latency to the persisting of the process. Thus, it is possible that our solution will not correctly recreate a process on a system with enabled swap.

### Firmware

We install a special SMI handler in the coreboot firmware that reacts to certain events sent by the Linux kernel and writes the gathered data to the FPGA device on a given event. To make it possible for the kernel and the firmware to exchange data, a specific memory location is used as a mailbox for both the kernel and firmware to put data for the other party to read.

## 4.3 Process Recreation

The recreation process begins with the initiation of a specialized `recreator` binary, which is given the binary that needs to be restarted as a parameter. The system assumes the process was already persisted as described in the previous section. This `recreator` binary is designed to clone itself with special parameters that inform the kernel it should start the recreation procedure.

The recreation is happening completely in the kernel. It starts by reading general metadata, such as the previously persisted process's binary name, from the firmware and conducts a sanity check to ensure the correct binary is recreated. As part of the `clone()` system call, the kernel adds additional memory regions to the virtual memory of the process, which it loads from the firmware. The recreated memory regions also include file backings and permissions.

The kernel then iterates through the PTEs that it reads from the firmware, adding new PTEs to the new process within the boundaries of the already created memory regions. This process results in the same virtual memory layout but different physical addresses than those in the original process, due to the dynamic nature of memory allocation.

Next, the state of the process, including the PID, CPU state like registers and file descriptors, is set. This ensures that the recreated process mirrors the state of the originally persisted process.

An important aspect of this process recreation is that it occurs not during normal runtime but before the actual start of the process. While recreating a process, the firmware is used to access data stored on the FPGA.

# Chapter 5

# Implementation

We shift our focus from theoretical designs to their practical implementation. This implementation is specifically carried out on a 12th generation Intel x86 CPU and an MSI Z690-A PRO motherboard, which is selected for its compatibility with coreboot. A key component of this implementation is the use of the already introduced FPGA device, which is used as a storage device. This chapter details how the concepts and designs previously discussed are actualized in a real-world hardware and software environment.

## 5.1 Kernel

For implementing the firmware-based whole process persistence, a Linux kernel based on the upstream version 6.1 is used. The approaches used are not limited to this version and could be backported to older versions or implemented in the current mainline kernel from kernel.org.

#### Kernel Firmware Communication

Communication with the firmware is encapsulated within an extra kernel module, `smi-trigger`. This module provides methods for the rest of the kernel to either communicate events to the firmware or read data from it. To transport data packets between firmware and kernel, we reserved the first $16\,\mathrm{MiB}$ of the previously mentioned firmware memory for a mailbox shared between the kernel and the firmware. To use this mailbox from within the kernel, another kernel module `smi-mailbox` provides interfaces for mailbox read and write operations.

`smi-trigger` internally writes to the mailbox and performs several checks. These checks include whether the system is in the correct state to receive the given event and if the properties of the received data are within defined boundaries. To

send these events and invoke the firmware, the kernel triggers a SMI in software by writing to the I/O port `0xb2`, with the value written to the port corresponding to the specific event type. The subsequent sections will provide a detailed account of the various events that we implemented.

Our function to trigger an SMI from within the kernel is shown in the listing 5.1.

Listing 5.1: Triggering SMI in the kernel

```
1  void trigger_smi_interrupt(enum EventType eventType)
2  {
3      printk(
4          KERN_DEBUG "SMI␣triggered␣with␣EventType:␣%i\n",
5          (u32)eventType
6      );
7
8      asm volatile("mov␣%0,␣%%eax;"
9                   :
10                  : "r"((int)eventType)
11                  : "eax"
12          );
13
14     asm volatile("out␣%eax,␣$0xb2");
15 }
```

### 5.1.1 Process Tracking

Process tracking within the Linux kernel involves several modifications.

The first modification is in the e820 parser, which is adapted to reserve $256\,\mathrm{MiB}$ of memory for the firmware to store internal data at a specific address. This location of this memory is required to be within the lower 32 bits due to coreboot executing the SMM handler in x86 real mode. We chose `0x10000000` as the address, which is hardcoded into the kernel and the coreboot firmware.

To be able to persist a specific process, we have to add a property the kernel's abstraction of processes, the `task_struct`. We did this by adding a boolean `should_be_persisted`. To tell the kernel that a newly started process should be persisted, we modified the clone system call. In our modified kernel, the `clone()` system call includes a check for a flag. We chose the currently unused `CLONE_DETACHED` flag [49]. If it is set, the `should_be_persisted` flag in `task_struct` is set to true. This flag in the `task_struct` is used by the memory subsystem to identify the marked process. We expose this flag to userspace by adding it to the `/proc` filesystem as shown in the figure 5.2 Additionally, an *init* event is communicated to the firmware which resets its state.

Listing 5.2: Struct for requesting memory page

```
1 $ cat /proc/42/process_persistence
2 Process Persistence flag: true
```

The kernel's page table entry management is also involved in process tracking. During `do_anonymous_page`, `do_file_page`, `do_no_page` and `zap_pte_range` operations, when PTEs are added to or removed from the current process's virtual memory area, these events are communicated to the firmware [50]. However, no additional logic is added to the kernel side for these operations. The PTE communicated consists of both the virtual and physical addresses.

Finally, in `do_mmap` and `unmap_region` functions, the information passed to these functions is communicated to the firmware and stored there. This includes details such as the file, virtual address, permissions, flags, and size of the memory regions [50].

### 5.1.2 Power Loss Event Handling

To separate the concerns of being able to detect power losses in a system and being able to recreate a process, we choose to simulate the power loss events. It is possible to detect an imminent power loss by checking the voltage level of the ATX `PWR_GOOD` line [31], but this was out of scope for this thesis.

Our kernel includes a module that creates a special file in the `/dev` directory, `/dev/wpp-trigger`. This file acts as an interface to communicate with the WPP system. When a predefined magic number is written to this file, the kernel module initiates the power loss handling procedure.

| Magic number | Description |
|---|---|
| 10 | Trigger power loss procedure |
| 20 | Reset state of firmware |
| 30 | Send test PTE |
| 40 | Send test memory region |
| 50 | Toggle debug flag 1 |
| 60 | Toggle debug flag 2 |

Figure 5.1: Magic numbers for `/dev/wpp-trigger` device

This procedure involves identifying and managing the process marked for persistence. The kernel module searches for a process with the `should_be_persisted` flag set. If such a process is found, the module removes it from the scheduler, effectively halting its execution on any CPU core. Halting the process needs to be done before any data can be persisted because the

state of the process is not captured in an atomic operation. A process that changes content in memory while they get persisted would be captured inconsistently.

To make sure all data in the cache hierarchy of the CPU is written back into system memory, we flush all caches into memory. To this, the kernel issues a `WBINVD` instruction on all cores using the `wbinvd_on_all_cpus` function provided by the kernel [16, 45].

Once the process is halted, the kernel module proceeds to communicate the PCB to the firmware. The PCB includes information about the process, such as open files, the CPU state, and other relevant data. This transmission is a crucial step in ensuring that all necessary details of the process state are available for persistence.

Following the halting of the process and the communication of its state, the kernel module then signals the power loss event to the firmware. In response, the firmware initiates the process of persisting the halted process to the FPGA device. This step completes the power loss handling sequence within the kernel, ensuring that the process state is preserved in the event of an actual power loss.

After the firmware completes the persisting, the process will be killed by the kernel to prevent it from changing the state of the machine. If the process would alter open files this could corrupt the recreated process that was recreated from the snapshot saved on the FPGA.

### 5.1.3 Process Recreation

The implementation of the process recreation is implemented based on the `clone()` system call. To recreate a previously persisted process, a special `recreation` binary is being executed. When the `clone()` system call is invoked with the `CLONE_DETACHED` parameter by the `recreation` binary, the system undertakes additional steps [49]. Both the need to track a process and to recreate a previously persisted process are expressed by adding the `CLONE_DETACHED` flag. To distinguish both cases, the kernel also takes into account the name of the binary. If it is equal to `recreator`, it executes the *process recreation* codepath. Initially, it retrieves the metadata of the last persisted process from the firmware and conducts a sanity check on the binary name. This step ensures that the correct process is being recreated.

If the sanity check is successful, the kernel retrieves memory region information from the firmware. These memory regions are then added to the newly created process. At this stage, the virtual memory area of the process includes all necessary components such as mapped open files, the process binary, and dynamically linked libraries. These components are set to be loaded by the kernel upon access.

However, to restore the contents of private memory regions or modified parts of regions, like open files, the actual pages must be loaded into memory. The

firmware, having tracked all added PTEs and saved the contents of the pages to the FPGA, provides the necessary information for this restoration.

The process involves filling the memory inside the already created memory regions with PTEs stored in the FPGA. This step restores the actual data that the process was using. However, it also leads to changes in the existing physical addresses.

### 5.1.4 PCIe Base Address Remapping

In the SMM, a challenge arises from the limitations imposed by the x86_64 real mode where only the lower $4\,\mathrm{GiB}$ of RAM are usable. This constraint becomes significant when dealing with the FPGA that exposes 511 PCIe virtual functions [53]. These virtual functions cannot fit within memory regions accessible from real mode.

The Linux kernel assigns the first BAR, BAR0 of the FPGA to a memory region that lies outside the lower $4\,\mathrm{GiB}$ of RAM accessible from SMM. Coreboot is not able to access these memory regions, necessitating a modification in the kernel to prevent any remapping outside the lower $4\,\mathrm{GiB}$ of memory.

The kernel typically utilizes the `pci_assign_resource` function to adjust memory regions [46]. In this case, the function is modified to avoid remapping the memory regions of any PCIe device that matches the FPGA's vendor and device IDs.

## 5.2 Firmware

Similar to the Linux kernel itself, coreboot is not distributed as a pre-compiled binary but in a source-code format [40]. There is a very limited choice of modern mainboards that are supported by open-source firmware. Dasharo is a distribution of coreboot and provides ports of the coreboot firmware to several mainboards. Thus, we can modify the firmware of a modern Alderlake-based mainboard, the MSI Z690-A PRO [9].

The main goal for the firmware implementation is to handle incoming data from the kernel and keep track of the memory regions and PTEs of the process. On a given signal, it is able to dump all this data and the content of the pages to the FPGA device. After system restart, it restores its internal state from the FPGA and waits for the kernel to request information about the previously persisted process while the kernel recreates it.

### 5.2.1 Coreboot Implementation

To implement a generic handler for SMIs in coreboot, a function `int mainboard_smi_apmc(u8 data)` has to be implemented [46]. This function will be called by the coreboot SMI handler with `u8 data` being the value written to the SMI port by the kernel, as shown in 5.3. The process persistence logic implemented in firmware has $256\,\mathrm{MiB}$ of system memory reserved. The base memory address is `0x10000000` and is sectioned as displayed in 5.2.

| Memory start address | Description | Section size |
|---|---|---|
| 0x10000000 | Mailbox | 0x01000000 |
| 0x11000000 | Metadata | 0x02000000 |
| 0x13000000 | Memory Regions | 0x03000000 |
| 0x16000000 | Internal Buffers | 0x02000000 |
| 0x18000000 | Page Mappings | 0x08000000 |

Figure 5.2: Layout for the memory that is reserved for the firmware to save temporary data about the process to persist.

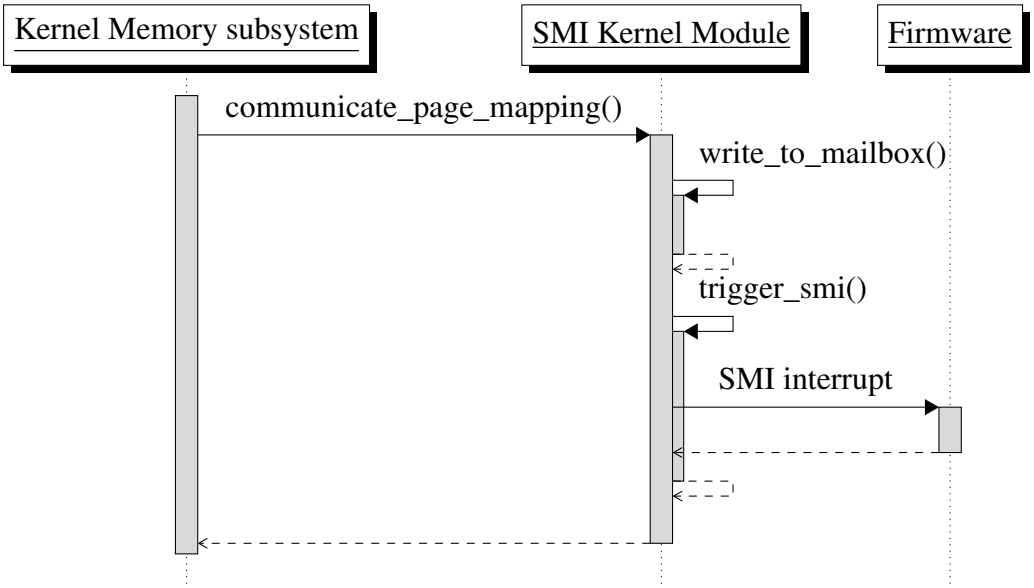


Figure 5.3: Example for the flow of information through kernel and firmware: A newly created PTE from the kernel memory subsystem is forwarded to the SMI kernel module developed in this thesis, which writes information about the PTE into the shared mailbox and triggers an SMI. This will invoke the firmware, which will perform actions depending on the received event.

### 5.2.2 Event Handlers

#### Init

The *Init* event handler clears the memory reserved for firmware operations and sets up the environment to be able to process incoming memory regions and page table entries. To accomplish this, the memory region for PTEs, kernel memory regions, and metadata is reset.

#### Add Memory Region

Calls to `mmap()` or files that are opened by a process will be creating new memory regions in the process's virtual memory area. To make it possible to persist these regions, we added a pair of events communicating addition or removal of such a region. To fully recreate the region, we need the properties listed in the listing 5.3. These include the start of the region at `virtual_address`, its size `length`, an optional path of the file backing the region `file_path`, and the protection and flags which are analogous to the parameters given to the `mmap()` system call. The *Memory Region* section is sectioned into slots of size `sizeof(struct memory_region_information)`. The struct in the listing 5.3 will be placed in the *Memory Regions* section of firmware memory. To find a slot for the incoming `memory_region_information`, the *Add Memory Region* handler iterates over all slots in the *Memory Regions* section until it finds an empty one.

Listing 5.3: Add memory region information

```
1 struct memory_region_information {
2     char file_path[100];
3     u64 virtual_address;
4     u64 length;
5     u64 protection;
6     u64 flags;
7 };
```

#### Remove Memory Region

When a memory region is unmapped or a file closed, the corresponding removal of this memory region will be forwarded to the firmware by passing the virtual address of the region to the firmware which is removing the corresponding `struct memory_region_information` from its memory.

### Add Page Table Entry

This handler receives a PTE in the form of the struct in listing 5.4 from the kernel and saves it in the *Page Mappings* section.

Listing 5.4: Memory page mapping struct

```
1 struct memory_page_mapping {
2     u64 virtual_address;
3     u64 physical_address;
4 };
```

The *Page Mappings* memory region is sectioned into slots of size `sizeof(struct memory_page_mapping)`. When receiving a new mapping, the handler searches for the first free slot and copies the incoming mapping into the slot. To prevent the handler from having to search through the whole *Page Mappings* region to find a new slot, the region is further divided into indices. There are 16 sections, and the index of the section is determined by the following formula: `(virtual_address & 0xF000) » 12`. This reduces the worst time effort to find an empty slot by a factor of 16.

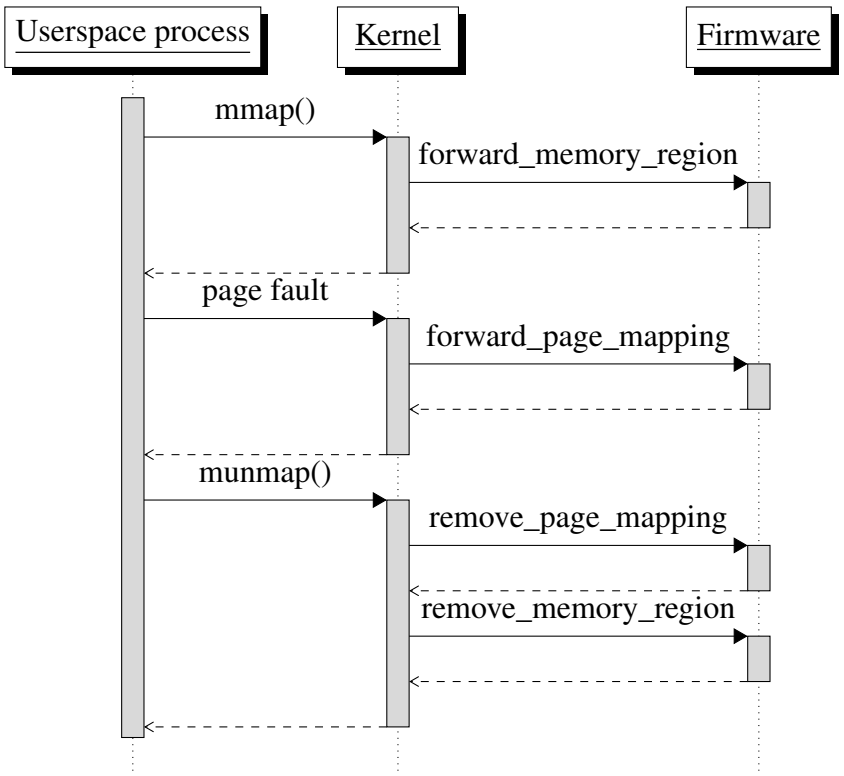


Figure 5.4: The creation of new memory regions and the creation of new PTEs through page faults is forwarded to the firmware transparent to the process.

### Remove Page Table Entry

This handler also receives a `struct memory_page_mapping`. It searches the slots in the *Page Mappings* memory region for the pair of virtual and physical addresses and zeroes out the section when found. Similar to the addition of new page mappings, the removal handler also only has to scan through one of the 16 indices to find the matching entry.

### Write Data

The *Write Data* handler is used to write data to the *Metadata* section of the firmware memory. Since all of the memory sections except the *Metadata* section are managed by the firmware itself, this event handler can only be used to write the *Metadata* section. Data received from mailbox is shown in listing 5.5.

Listing 5.5: Struct to write data

```
1 struct data_command {
2     enum memory_sections section;
3     u32 offset;
4     u32 size;
5 };
```

The only valid section for writing data is the *Metadata* section, but the interface is used for reading data as well. Through the `offset` and `size` parameters, the kernel tells the firmware which exact memory section should be overwritten. The firmware will copy the number of bytes given in the `size` parameter from the mailbox into the specified section of memory at the given offset.

### Read Data

The *Read Data* handler is built analogous to the *Write Data* handler. It receives a `struct data_command` and copies the number of bytes given in size from the given section and the given offset to the mailbox. After the handler runs, the kernel can read the mailbox.

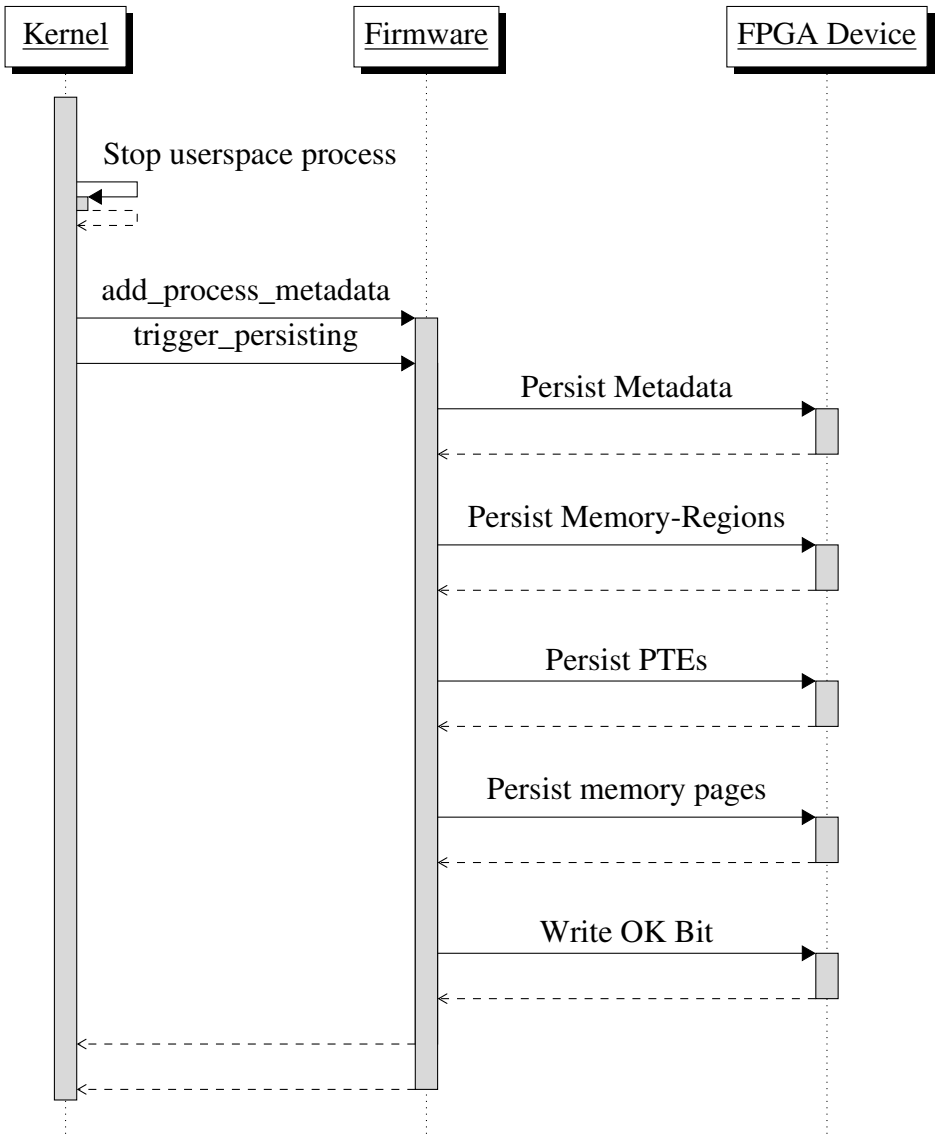


Figure 5.5: Sequence of persisting the previously tracked process onto the FPGA. After the process is stopped and process metadata is sent to the firmware, it writes each memory region to the FPGA and copies all used pages from memory to the FPGA.

### Trigger Persisting

The *Trigger Persisting* handler is the core of the persistence operation in the firmware. It gets triggered when the process that should be persisted is running and a power loss signal is received. After the kernel stops the process and copies metadata like the PCB into the *Metadata* section, this handler copies all data to

the FPGA. First, the handler copies all of the memory sections of the firmware to the FPGA. This includes the sections *Metadata*, *Memory Regions*, and *Memory Pages*. The whole $256\,\mathrm{MiB}$ of internal memory is getting copied to the FPGA. After copying the internal state of the firmware, the memory pages have to be copied. All memory pages in the *Page Mappings* section are copied from system memory to the FPGA in the same order they appear in the *Page Mappings* section. We copy the pages by issuing a command to copy $4\,\mathrm{KiB}$ of memory from the base address of the page to the FPGA. This process is detailed in 5.5 After all pages are saved to the FPGA, an OK bit is set to indicate a finished persistence process. The layout of the FPGA memory is shown in 5.6.

### Restore from FPGA

The *Restore from FPGA* handler is called as part of the recreation process. After the process was performed and the system experienced a shutdown, this event handler is executed. It takes the firmware sections that were copied to the FPGA in the Trigger Persisting handler and copies them from the FPGA back to the reserved firmware memory. This makes it easier for the kernel to copy data from the sections using the *Read Data* command.

### Get Pages from FPGA

After Restore from FPGA, the kernel can use Read Data to access all data except the actual content of the memory pages. Access to the pages saved on the FPGA is given through this command handler.

Listing 5.6: Struct for requesting memory page

```
1 struct get_page {
2     u32 index;
3     u64 address;
4 };
```

The firmware reads the $4\,\mathrm{KiB}$ page at the given index from the FPGA and places them in the main memory at the give `address`.

| FPGA start address | Description | Section size |
|---|---|---|
| 0x40000000 | Firmware Metadata | 0x002000000 |
| 0x42000000 | Firmware Memory Regions | 0x003000000 |
| 0x45000000 | Firmware Internal Buffers | 0x002000000 |
| 0x47000000 | Firmware Page Mappings | 0x008000000 |
| 0x50000000 | System Memory Pages | (up to) 0x800000000 |

Figure 5.6: FPGA storage layout of a persisted process

#### Debug

The debug handler is only used for development purposes and performs several actions. These include:

- Discover the FPGA device on the PCIe Bus
- Activate the FPGA device and setup its queues
- Check whether the FPGA device functions correctly by writing to it and reading from the same address
- Log information about the internal state of the firmware memory sections
- Log contents of the mailbox

### 5.2.3 FPGA Device Communication

The FPGA device is connected via PCIe and employs Persistent Memory (PMem) to access the entire RAM of the system while exclusively using MMIO for its operations [24]. The initialization of the FPGA involves mapping its registers into memory and resetting the device. Coreboot provides the necessary primitives for communicating with PCIe devices and setting them up.

#### FPGA Initialization

For the FPGA initialization, the first task is to locate the FPGA by enumerating all PCIe devices and filtering them based on the vendor and device ID. Once identified, the PCIe device is enabled, setting the `bus master` and `memory` flags. This enables the FPGA to access the entire RAM of the host system using DMA [32]. The FPGA is capable of performing operations such as copying from any Optane address to a host memory address and vice versa.

These operations are managed through individual ring buffers of requests, and the memory for these buffers must be allocated from the firmware, not the FPGA. The memory locations of these buffers are then written to the memory-mapped registers of the FPGA.

#### FPGA Functions

The read and write functions of the FPGA utilize their own queues but are analogous to each other. The ring buffer is updated once the FPGA completes a command. When issuing a command, it is queued into the ring buffer, and the FPGA registers, which contain the index of the most recent object in the buffer, are incremented.

However, every 128th element of the ring buffer is unusable due to the FPGA implementation, requiring these entries to be skipped.

It is possible to issue commands at a rate faster than the FPGA can process them. To prevent overwriting commands in the ring buffer that have not yet been completed, it is necessary to track the number of in-flight commands. If this number reaches the buffer capacity, the system must wait until a command is finished before issuing new ones. This process ensures all commands are handled.

### 5.2.4 Development and Debugging of Firmware

In this section, we show some of the problems and subtleties you encounter when developing and debugging system firmware. In contrast to the kernel, it is much harder to update firmware and broken builds are harder to recover from, since a broken firmware means your mainboard does not boot anymore. Also, logs generated by system firmware are harder to read than userland or kernel logs.

#### Coreboot Build and Deployment

Building the coreboot image for a board involves different configuration steps [39].

We use the Dasharo distribution of coreboot, which has already established a configuration for booting EDK2 on the mainboard in use [8]. Dasharo offers a Docker image that comes pre-equipped with all necessary dependencies for compiling their coreboot distribution, simplifying the build process.

Additionally, Dasharo provides scripts for building coreboot and deploying it onto the MSI board. Our configuration process involves several critical steps: providing the serial number and UUID of the board, enabling an entry point for SMIs, activating logging within the SMM, and deactivating lockdown and security features. The latter is particularly important during the development phase to ease the re-flashing of the board.

To flash the firmware onto the board's onboard flash memory, we employ flashrom [41], a tool for interacting with flash chips. We are proceeding according to Dasharo's resources when adding the previously extracted serial number and system uuid into the binary and resigning the binary before flashing it.

#### Firmware Debugging

There are several challenges when developing and debugging system firmware. Changes do not only require a lengthy compilation, but the firmware also has to be written to flash memory and the whole system has to be restarted. To test multiple changes without having to recompile and restart the system, we use feature flags in

the firmware that can be triggered via the interface the kernel uses to communicate with the firmware.

Since a system with malfunctioning firmware is probably not going to boot we added a source of non-determinism for changes to firmware code that is used in the boot sequence. Changes that can break the boot process are placed behind a `should_use_new_feature` switch that chooses to run the new feature 90% of the time based on the `timestamp_get` function in coreboot, which on x86_64 systems is based on the timestamp counter `rdtsc` instruction [37] [16].

If a change nevertheless is preventing the system from booting, there are several ways to get the system out of this bricked state. Since version 1.1.2 from September 2023, the Dasharo coreboot version for the MSI Z690-A PRO is supporting MSI FLASHBIOS. This feature helps with recovery by allowing the board to flash the original firmware from a USB drive. To enable this, in the last $64\,\mathrm{KiB}$ of the firmware image, some special symbols contain information about the board identification and currently installed firmware [38].

Similar to the kernel's `printk` infrastructure, coreboot also provides print functions. Logs that are emitted through these functions can be piped to various outputs. Since we are not interested in logs from the early boot stages, access to the coreboot logs from the running operating system is sufficient [35]. All logs emitted by our software are prefixed with `wpp:` to make it easy to omit all other output produced by the firmware.

Several challenges can arise when writing to and reading from the Optane memory using the FPGA. During the development phase, we encountered an unexpected configuration change in the host machine, where the system was initiated with the Input/Output Memory Management Unit (IOMMU) enabled. The FPGA's inability to access all memory protected by the IOMMU led to a situation where the system froze while awaiting the completion of write commands.

To be able to debug this issue, we implemented a timeout mechanism. If the system is waiting for the completion of all commands for more than 5 seconds, it will continue even if not all commands succeeded but logging the timeout.

# Chapter 6

# Evaluation

Now, we focus on assessing the practicality and effectiveness of our implementation. This chapter showcases our test system and outlines the performance impact on runtime and the duration of the persistence process across several applications with varying patterns of memory access. Additionally, we discuss the inherent limitations of the concept as well as those arising specifically from our implementation approach.

## 6.1 Test Environment & Methodology

To be able to make meaningful statements about the performance of our solution, we have to present the setup of our testing environment, detailing the hardware and software configurations utilized to evaluate the implementation's performance.

#### Environment

The test machine is based on an MSI Z690-A PRO motherboard, equipped with 32 GB of DDR5-5600 DRAM (4x 8GB). The CPU is an Intel Core 12700K, including 8 performance and 4 efficiency cores. Since all performance cores support 2x hyperthreading, the CPU supports up to 20 threads. This MSI board is one of the only modern mainboards with coreboot support from Dasharo.

The kernel and custom `starter` and `recreation` binary were compiled with gcc 11.2. The operating system used for the tests is Ubuntu 22.04, running a modified version of the Linux kernel based on version 6.1.55. The firmware used for testing was based on the Dasharo coreboot firmware, version 1.1.2. Kernel and firmware were modified as described in the previous chapters. The test system was equipped with the FPGA device developed by Werling et. al. [53].

For controlling the test machine without physical access, we utilized PiKVM, a Keyboard Video Mouse (KVM) solution [11]. PiKVM helped with tasks such as resetting and rebooting the system and settings in the boot process.

#### Test Methodology

We conduct benchmarks to assess the runtime performance of our WPP solution, focusing on single-threaded applications. The data collection process faced challenges, particularly in identifying applications that could be effectively restored with our current implementation and pinpointing real-world applications where performance is primarily constrained by the creation and removal of PTEs rather than CPU limitations.

To maintain consistency across evaluations, all tests were conducted using the same revision of the kernel and firmware. All debug logs related to adding or removing PTEs in the kernel and firmware were removed for the testing. Additionally, the firmware was reset to its default state between tests to eliminate any potential carryover effects. We did not experience any problems when not resetting the firmware but added the resets as a precaution. To ensure the reproducibility of our findings, tests were executed using an automated script. Time measurements were conducted using the `date` command to get the passed time in milliseconds [43].

## 6.2 Runtime Performance Impact

Now, we delve into the performance implications of our whole process persistence solution on process runtime, as initially outlined in the design chapter. We aim to minimize the performance impact, particularly noting that the main overhead arises during the creation of numerous memory mappings. The effect on workloads varies based on the frequency of creation and deletion of PTEs. We explore both the theoretical performance implications by comparing against a default state under a synthetic load scenario and the actual performance impact through testing with a real in-memory database, Redis [33]. This approach provides a comprehensive view of how our implementation affects system performance in both controlled and practical environments.

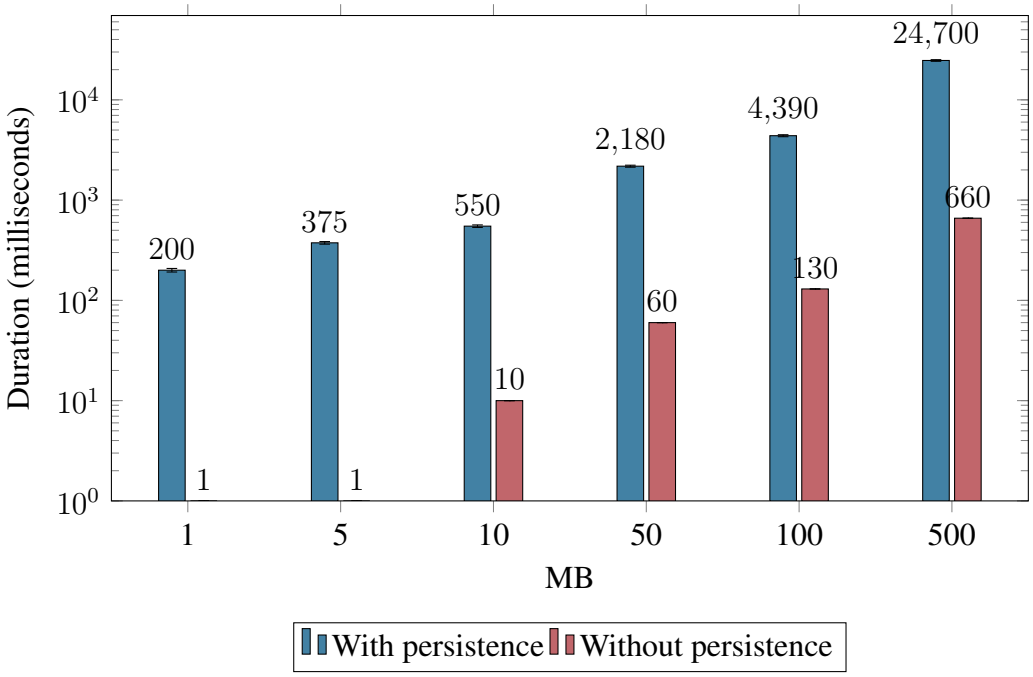


Figure 6.1: Duration of a pass of our synthetic benchmark which is allocating and freeing memory regions of the given size, thus highlighting the bottleneck of our implementation, the creation and removal of PTEs.

### 6.2.1 Synthetic Load

In this section, we examine the runtime performance impact of our whole process persistence solution under a controlled environment. For these tests, programs are not recompiled for persistence but are initiated using a `starter` binary. The test program is designed to allocate large, contiguous chunks of memory ranging from 1 MiB to 500 MiB using the `mmap()` system call, deliberately avoiding malloc to bypass the standard library's memory allocation strategies. With this benchmark, we test the overhead of creating and removing a large amount of PTEs.

The program executes a cycle of calling `mmap()`, populating the allocated memory with random content, and then releasing it with `munmap()`, repeating this process 10 times. Our evaluation focuses on measuring two primary aspects: the one-time overhead at the start due to the use of the starter binary instead of launching the real binary directly, and the overhead associated with memory mapping. The latter is considered negligible as it occurs only 10 times. However, the overhead for each newly assigned PTE is significant due to the requirement for a context switch into SMM for every PTE assignment, which is resource-intensive.

Our findings indicate a substantial initial overhead, with program runtime in-

creasing by $150\,\mathrm{ms}$. This overhead originates from the need to start the `starter` binary which needs to clone itself to start the real application. When a process that has to be persisted is started, the firmware is resetting its state, contributing to the initial performance overhead. However, as the size of the data sets increases, this overhead stabilizes around a 35x increase for workloads involving frequent mapping and freeing of memory regions. We have to keep in mind that this workload is not a representation of real processes and that most real world workloads are not experiencing a slowdown of this magnitude.

The tests, conducted 500 times to ensure reliability, present the median result for consistency.

### 6.2.2 Redis

We extend our performance testing beyond synthetic loads to include a real world application. We selected Redis version 5.6, a widely used in-memory key-value store, for this purpose [33]. Redis facilitates a single-threaded architecture and offers an included benchmarking tool that we used to simulate 50 clients accessing the server with payloads of different sizes.

Our tests encompassed `SET`, `LRANGE`, and `LPUSH` commands, varying the sizes of the data involved. We use one Redis instance started with our `starter` binary and thus being in tracked mode and another one being normal Redis process. To benchmark these, we use the included `redis-benchmark` tool with 4 different payload sizes, doing 50 requests in parallel and performing 100000 requests. The results are laid out in 6.2. Initial findings indicate that for smaller data sizes, where the server does not require additional memory allocation, the performance of Redis remains consistent, showing no statistically significant deviation from the not persisted version. However, as the data size increases, necessitating the allocation of larger memory chunks and consequently the creation of more PTEs, we observed a notable impact on performance. This effect was particularly pronounced in the `LRANGE_600` test, where a significant drop in performance was attributed to frequent reallocations of memory, highlighting the relationship between memory management operations and application performance.

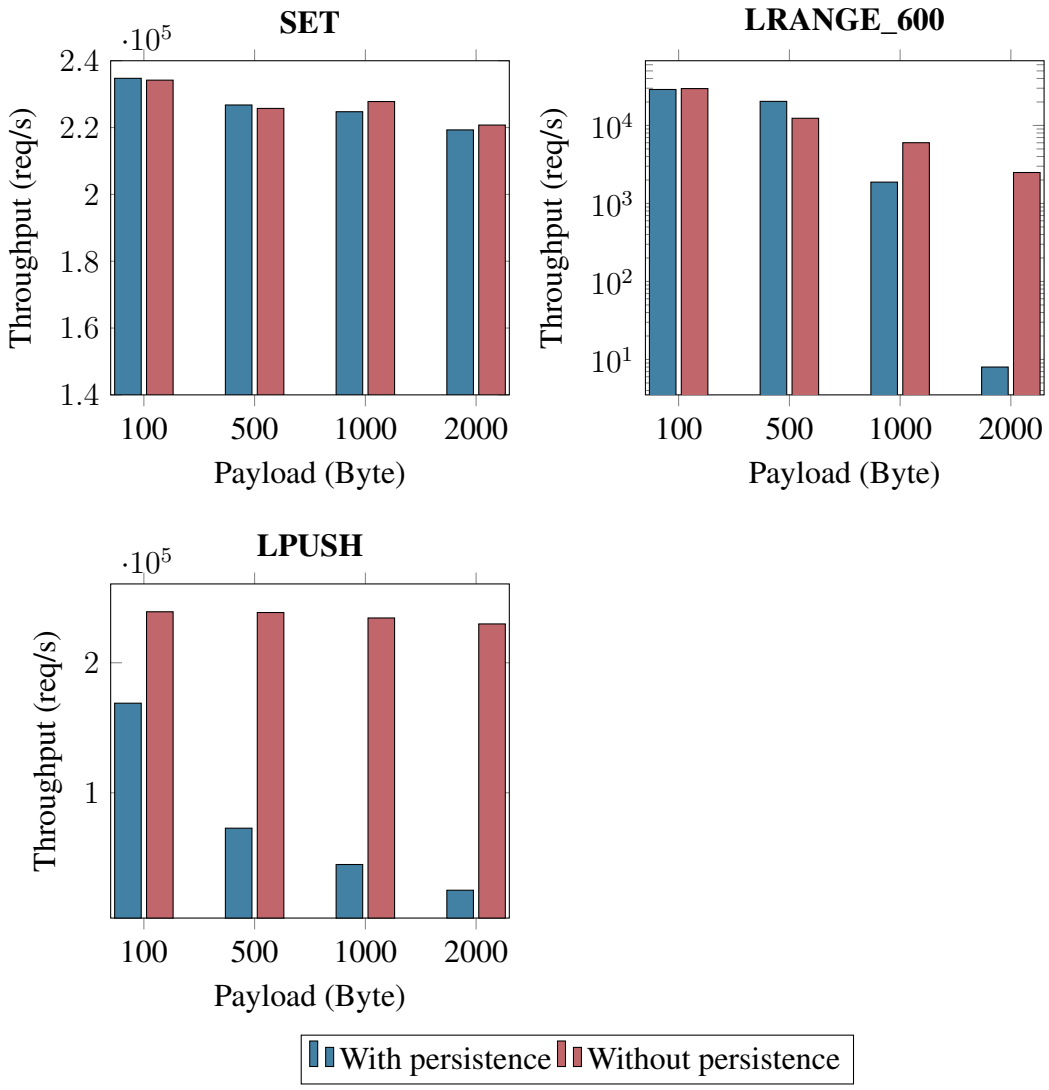


Figure 6.2: Throughput of common redis operations. Operations with a small payload size see a small or no performance impact, while larger payloads force the application to allocate more memory which is slowing it down noticeably when the redis process is tracked by our solution.

### 6.2.3 Impact on other Workloads

We also have to test the impact of a process being tracked on other workloads on the same machine. We use the $7z$ LZMA benchmark that runs in parallel to one process allocating $10\,\mathrm{GiB}$ of memory while being tracked by our solution [29]. We used one thread and standard dictionary sizes of $4\,\mathrm{MiB}$ - $32\,\mathrm{MiB}$. In our benchmark 6.3

we see that the throughput of the runs that are executed in parallel to the process marked for persistence suffers is only about 10 % of the control group. Although our system can execute 20 threads in parallel, the constant invoking of the SMM by the process to persist is causing all other cores to enter SMM [10]. This degrades the performance of each running userland process and kernel operations.

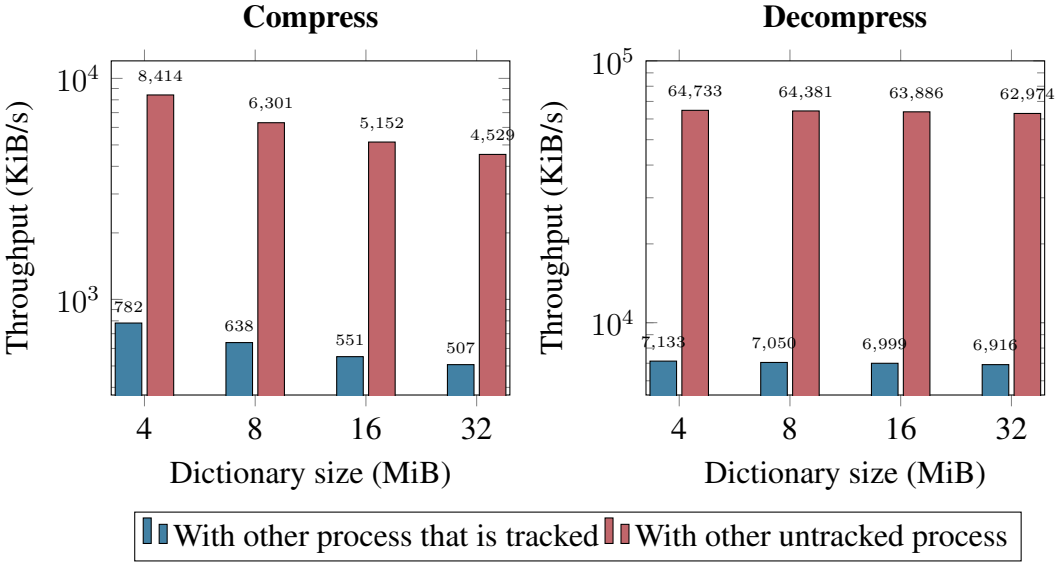


Figure 6.3: Throughput of the *7z* LZMA benchmark for various Dictionary sizes for both compression and decompression. We compare the differences in throughput that is caused by another process being tracked by our solution. Since the tracked process forces all other cores into SMM frequently, the performance of the benchmark drops to 10% of it's original throughput.

## 6.3 Persisting Performance

Now, we focus on the performance aspects of persisting a process using our solution. One inherent challenge in this process is the overhead incurred from the continuous transfer of metadata. The firmware's data sections, amounting to $256\,\mathrm{MiB}$, are transferred in their whole, resulting in approximately $160\,\mathrm{ms}$ of overhead. This transfer time increases with the amount of memory allocated by a process.

The time to persist a process is heavily dependent on the amount of allocated memory. The time to persist a process for various amounts of allocated memory is shown in 6.4. Our system is achieving a transfer speed of up to $1.3\,\mathrm{GiB/s}$ of allocated memory per second to the FPGA. The FPGA faces a constraint due to the necessity of conducting $4\,\mathrm{KiB}$ move operations, rather than more efficient $1\,\mathrm{MiB}$

transfers.

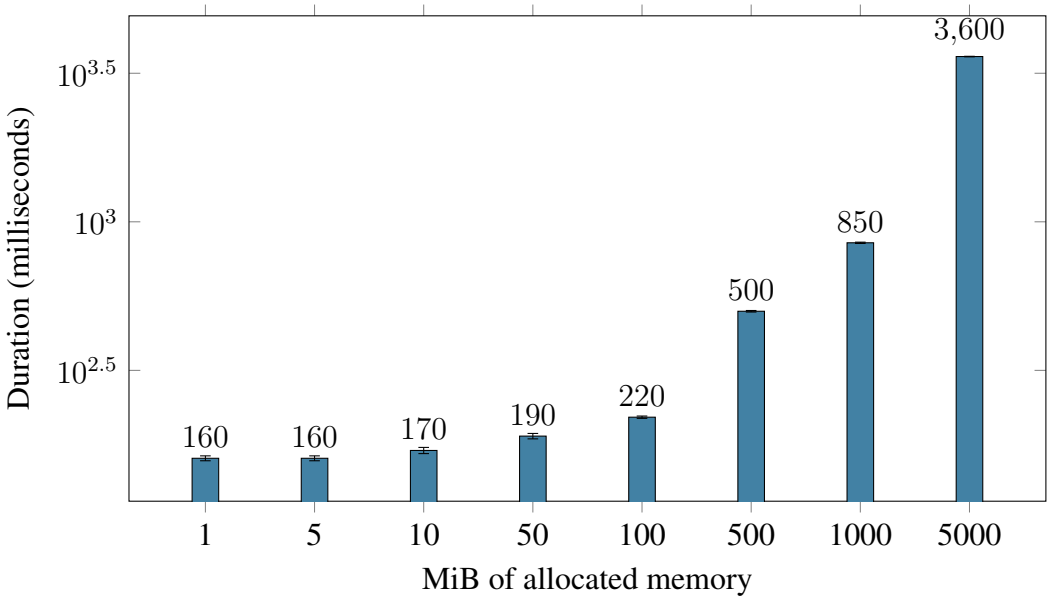


Figure 6.4: Duration of the complete persisting process after the power-loss event was received for different sizes of allocated memory. The initial $160\,\mathrm{ms}$ are a static overhead that arises from the firmware copying a fixed amount of metadata.

To evaluate the performance of our persistence approach, we conducted benchmarks using a synthetic load. This load involves allocating a specified amount of RAM, populating it with data, thus creating a specific number of PTEs, and then triggering a power loss (simulated by writing to `/dev/wpp-trigger`).

## 6.4 Discussion

In the following, we address the constraints and challenges encountered both in the conceptual framework and the specific implementation. One key limitation at the conceptual level is the inability to recreate any process accurately due to significant changes in the environment. From the implementation perspective, our approach is heavily reliant on specific technologies and architectures, including the use of a custom FPGA, x86 processors, SMM, and the handling of normal page sizes.

### Concept

The concept of whole process persistence as described faces several limitations. These limitations stem from the inherent complexity of accurately recreating a

process's state and the environment it operates in.

One key limitation is the expectation that processes to be persisted should remain unaware of this persistence mechanism and should not require any changes to their source code to help with the persistence. This requirement poses significant challenges in maintaining the exact state of the environment during process recreation.

Time-sensitive aspects present another challenge. The passage of time can lead to discrepancies, such as changes in the system time or timeout issues in communication protocols like Bluetooth and TCP/IP. These protocols might have moved past the state they were in when the process was originally persisted, leading to potential communication breakdowns or synchronization issues.

Applications like the Chrome Browser or the Media Subsystem in Android [4, 28] are sectioned into several running processes to isolate different parts of the application from one another. Processes that are part of a larger set of interdependent processes pose additional complexity. For the system to function correctly, all these related processes would need to be recreated simultaneously, maintaining their inter-process communication and synchronization.

Resource allocation presents further challenges. Upon recreation, previously used resources like files or network ports might be unavailable or blocked, which could hinder the process's ability to resume its normal operation.

### Implementation

There are other limitations inherent to the specific choices made in the implementation strategy and the technologies employed.

One of the primary limitations arises from the use of SMM, which is not portable to other architectures like ARM or RISC-V. This reliance on SMM restricts the implementation to platforms that support this specific mode, primarily x86 architectures. However, there is a potential workaround that involves implementing the required functionality solely within the kernel, which could offer greater portability across different architectures.

Another challenge is related to the kernel setup of the process's virtual memory layout, such as the stack address. Modern operating systems are employing strategies like Address Space Layout Randomization (ASLR) to randomize the address space of userspace processes [2]. This complicates the process of accurately recreating the process state, necessitating the disabling of ASLR.

The implementation is also only implemented for standard $4\,\mathrm{KiB}$ x86 pages and does not support huge pages. While any storage device that can be accessed over the PCIe Bus can be used from a concept point of view, the implementation relies on an FPGA device that is not publicly accessible.

The frequent usage of SMM is a severe performance problem both for the

process that is being tracked and other processes running on the same system. A SMI forces all of the cores to enter the SMM. So each clock cycle our solution resides in SMM essentially halts all other processes on the system until we leave SMM.

### Residual energy

The current solution's portability is limited due to the absence of a standardized guideline on the amount of energy a Power Supply Unit (PSU) should be able to supply after a power loss on the grid before voltage begins to drop. Moreover, external variables such as connected devices or the CPU's workload can further accelerate the depletion of residual energy. Consequently, any solution predicated solely on a PSU's residual energy lacks universality and necessitates testing across each used hardware configuration to ensure that the time to persist a process is actually covered by the residual energy of the PSU.

# Chapter 7

# Conclusion

In this chapter, we revisit the concept of Whole Process Persistence, initially explored in the Zhuque paper [14], and address the question of whether it is feasible to achieve process persistence without relying on system-inherent persistent memory. Our investigation confirms this possibility through a firmware-aided approach that leverages an FPGA as a persistent storage device.

Our solution introduces a low-complexity firmware role, primarily acting as a bridge to the FPGA while the process is running. We decided to reduce the complexity of operations executed by the firmware, due to the challenges in debugging it. We have integrated PCIe primitives, FPGA specific logic, and mechanisms for kernel communication into coreboot-based firmware, alongside adjustments to the process and memory subsystems of the Linux kernel to communicate information about a currently running process to the firmware. Additionally, we introduce modifications to the subsystem creating new processes in the kernel, enabling the recreation of a previously persisted process.

Notably, our approach is not exclusively dependent on FPGA technology, as it can be extended to other PCIe devices. We demonstrated our solution on a modern consumer motherboard equipped with an Alder Lake CPU. The observed performance penalty is highly dependent on the memory allocation strategy of the software being persisted.

While we have a worst-case performance decrease of about 35 times, real world benchmarks show that applications that are not allocating much memory during runtime are not slowed down at all. This is indicating that our solution is competitive with existing models. However, performance significantly declines with frequent allocations and deallocations of PTEs and a large amount of PTEs, highlighting a limitation in our approach. Moreover, the feasibility of persistence is constrained to processes of low complexity due to the challenge of accurately recreating them in a changed environment.

When looking back to the requirements formulated in Chapter 4, our solution makes it possible to accurately **track**, **persist** and **recreate** low complex processes with a small memory footprint. As long as the allocated memory of an application is not above $100\,\mathrm{MiB}$, the duration of persisting the process is consistently below $220\,\mathrm{ms}$, so it is possible to accurately restore a process while only bringing small performance regressions during runtime when the memory footprint of the application is small enough.

## 7.1 Future Work

Following the presentation and evaluation of our approach to whole process persistence, we identified several key areas ripe for further improvement and research. These areas encompass the exploration of the novel Compute Express Link (CXL) protocol, the development of a real power loss detection mechanism, cross-platform portability of our solution and the exploration of performance enhancements.

### CXL

CXL is a cache-coherent interconnect that is electrically compatible with PCIe and connects processors and accelerators similar to PCIe [5]. It supports several protocols, including CXL.io, which is based on existing PCIe mechanisms, as well as CXL.cache and CXL.mem for cache and memory protocols, respectively.

CXL is particularly interesting for use cases that involve the memory system. CXL.mem is integrated into the cache and memory subsystem of a processor and can be used to address memory on an accelerator card at the byte level. This capability could be leveraged to offload data to an external device, such as the FPGA device used in this thesis. It presents the possibility of sharing a single device for data storage between multiple servers, all running a version of process persistence.

CXL.mem and CXL.cache could simplify the implementation of functionalities like crash-consistent transactions, memory snapshots, or compression of memory by offering mechanisms to share memory and introduce cache coherence into the bus. These capabilities, named „Enhanced memory functions“ by Boles et. al. could replace the need for custom implementations in the kernel or userland [3].

The approach taken in this thesis with PCIe could be redone using CXL. This shift might even allow for bypassing the coreboot stage entirely and relying solely on kernel-based changes with CXL, offering a streamlined process for the process persistence mechanism.

### Power Loss Detection

In this thesis, the mechanism for detecting power loss is not an actual hardware-based detection but rather an event simulated by the user and sent to the kernel. This approach is due to the lack of a standardized method for the system to notify about a power loss.

While the ATX specification does include a `PWR_OK` signal line, it does not generate an interrupt [18, 31]. To effectively use this signal for power loss detection, it would be necessary to continuously read the line and forward this information to the kernel. One possible method for achieving this could be through a micro-controller that monitors the `PWR_OK` signal and triggers an interrupt upon changes [31].

However, this approach is not ideal as it lacks portability and general applicability across different hardware setups. It requires additional hardware and integration efforts, making it a less elegant solution. Therefore, more research is needed to develop a more universally applicable and efficient method for detecting power loss in systems designed for WPP.

An improvement could be achieved through the integration of an Uninterrupted Power Supply (UPS), which can supply significantly more energy than the residual energy typically available in the PSU. This would enable the system to remain powered for minutes, rather than just milliseconds, during a power loss.

The implementation of such a system would require the UPS to communicate a shutdown signal to the operating system. For Linux systems, tools such as Network UPS Tools (NUT) and Apcupsd offer robust solutions for managing communications with UPS devices across various vendors [25, 26]. NUT is a cross-platform tool with support for multiple UPS communication protocols from different vendors and configuration flexibility via config files. It also allows the execution of custom scripts in response to specific events, such as power failures.

For our purposes, a custom script could be devised to initiate the power loss persistence process as previously outlined, followed by a system shutdown. This approach would not only extend the available time window for data persistence during power outages but also enhance the overall reliability because the chances of the system shutting down before the persistence process is finished are reduced.

### Performance Optimization

An important factor influencing performance is the tradeoff between runtime performance and persistence performance. This becomes particularly evident in scenarios where there is a significant amount of memory allocated that is not backed by a file, adversely affecting persistence performance and reducing the likelihood of timely process persistence.

The frequency of memory allocations and deallocations also has an impact. Regularly allocating and deallocating memory pages can incur a performance penalty during runtime. The optimal strategy for managing this tradeoff is highly dependent on the memory allocation behaviour of the process. For instance, optimizations aimed at reducing the time required for persisting data might lead to reduced performance of the application while it is running.

One potential solution for improving performance involves writing memory pages to the FPGA while the process is running. This approach could be particularly beneficial for processes with a large memory footprint that is infrequently written to, on the other hand not helping processes that allocate little memory and changing it frequently.

In addition to these optimizations based on the memory allocation profile, there are possible general optimizations that can be applied regardless of allocation patterns. For example, it may be advantageous to avoid saving any pages that belong to backed files marked as clean or pages that belong to read-only files. By implementing these optimizations, it is possible to enhance the performance of the persistence process.

### Portability

Since our current approach is bound to x86 features like the SMM and needs customized firmware to work, it is not portable and hard to deploy. There is no need to use system firmware to collect the data of a process during its runtime and dumping it onto a storage device like the FPGA. A solution implemented as an out-of-tree kernel module would be more performant since it would not slow down other processes due to the frequent triggering of the SMM. Also, being not reliant on system firmware would make the solution portable and thus compatible across a wide range of mainboards and even other CPU architectures like *ARM* or *RISC-V*.

### Event Batching

A problem of our solution is the loss of performance both for the process to persist and other processes running on the same system. Entering SMM is causing significant overhead for all other processes since all cores enter SMM, and we should minimize time spent in this mode. One solution to this could be to not invoke the firmware for every PTE but to collect created or removed PTEss in the kernel and forward them to firmware in one batch. The Linux kernel already implements a similar performance optimization. When a page fault occurs, it is not only mapping the page related to the faulting memory address but a few pages around the faulting one. This is done in the `do_fault_around` function [50]. While this is a performance optimization in the kernel, at the moment it could

decrease the performance of a process tracked by our solution. Batching PTE creations and removals would decrease the bottleneck introduced by frequently switching into SMM.

# Acronyms

**ACPI** Advanced Configuration and Power Interface. 5–7, *Glossary:* ACPI

**ASLR** Address Space Layout Randomization. 42, *Glossary:* ASLR

**ATX** Advanced Technology Extended. 14, 23, 47, *Glossary:* ATX

**BIOS** Basic Input/Ouput System. 5, 7, 54, *Glossary:* BIOS

**CXL** Compute Express Link. 46, *Glossary:* CXL

**DMA** Direct Memory Access. 32, *Glossary:* DMA

**FPGA** Field Programmable Gate Array. v, 4, 8, 9, 18, 19, 24, 25, 30–35, 40–42, 45, 46, 48, *Glossary:* FPGA

**IOMMU** Input/Output Memory Management Unit. 34, *Glossary:* IOMMU

**PCB** Process Control Block. v, 4, 17, 18, 24, 30, *Glossary:* PCB

**PCIe** Peripheral Component Interconnect Express. v, 4, 7–9, 18, 25, 32, 45, 46, *Glossary:* PCIe

**PMem** Persistent Memory. 4, 9, 13–15, *Glossary:* PMem

**PSU** Power Supply Unit. 43, *Glossary:* PSU

**PTE** Page Table Entry. v, 18, 19, 23, 25–28, 30, 36–38, 41, 45, 48, 49, *Glossary:* PTE

**SMI** System Management Interrupt. 6, 7, 19, 22, 26, 33, 43, *Glossary:* SMI

**SMM** System Management Mode. 4–6, 8, 22, 25, 33, 37, 40–43, 48, 49, *Glossary:* SMM

**UEFI** Unified Extensible Firmware Interface. 5, 7, *Glossary:* UEFI

**UPS** Uninterrupted Power Supply. v, 47, *Glossary:* UPS

**WPP** Whole Process Persistence. v, 13, 14, 23, 36, 47, *Glossary:* WPP

# Glossary

**ACPI** A specification that enables operating systems to control the amount of power given to each device attached to the computer, facilitating power management and device configuration. 5

**ASLR** A security technique used in operating systems to randomly position the address space of process components. This makes it harder for attackers to predict the location of specific instructions or structures, thereby mitigating certain types of attacks. 42

**ATX** A motherboard and power supply configuration specification that improves on previous standards like AT by rearranging the layout to allow for better airflow and more efficient use of space. 14

**BIOS** Firmware used to perform hardware initialization during the booting process and to provide runtime services for operating systems and programs. It acts as an intermediary between the operating system and the computer hardware. 5

**CXL** A high-speed CPU-to-Device and CPU-to-Memory interconnect aimed at high-performance computing, offering efficient performance and scalability for future computing environments. 46

**DMA** A feature allowing connected peripherals to directly access the main system memory without the CPU being involved. 32

**FPGA** An integrated circuit designed to be configured by a customer or a designer after manufacturing, allowing for customizable hardware solutions in various applications. In this thesis, FPGA always refers to the FPGA device introduced in "Analyzing and Improving CPU and Energy Efficiency of PM File Systems" [53]. 4

**IOMMU** A memory management unit that connects a DMA-capable I/O bus to the main memory, providing address translation and memory protection from I/O devices, enhancing system security. 34

**PCB** A data structure used by computer operating systems to store all the information about a process, including process state, process number, registers, and memory management information. 4

**PCIe** A high-speed serial computer expansion bus standard designed to replace older bus standards like PCI, allowing for faster data transfer between the motherboard and attached devices. 4

**PMem** A type of non-volatile storage technology that retains data even when power is turned off, combining the speed of RAM with the data persistence of traditional storage. 4

**PSU** A component that converts electrical power from a source to the correct voltage and current to run a computer system. 43

**PTE** An entry in a page table, which contains information about how virtual memory addresses map to physical addresses, including details like the physical address itself, protection attributes, and presence bit. 18

**SMI** A special interrupt type that is used for system management, allowing the operating system to be temporarily suspended for hardware or firmware to perform low-level tasks. 6

**SMM** A special operating mode in Intel and compatible CPUs intended for handling system-wide functions like power management, system hardware control, and proprietary OEM designed code. 4

**UEFI** A specification that defines a software interface between an operating system and platform firmware, replacing the legacy BIOS with a more modern, feature-rich interface that includes boot and runtime service calls available to the operating system. 5

**UPS** A device that offers emergency power to servers and computers to prevent data loss or hardware damage during power failures. 47

**WPP** A programming model introduced by Hodkins et. al ensuring all process states remain persistent. Upon system restart after a power failure, the process state is fully reloaded, allowing execution to resume seamlessly with minimal programmer intervention [14]. v, 13

# Bibliography


[1] Uchenna P Daniel Ani, Hongmei He, and Ashutosh Tiwari. Review of cybersecurity issues in industrial critical infrastructure: manufacturing in perspective. *Journal of Cyber Security Technology*, 1(1):32–74, 2017.

[2] David Herrera Aristizabal, David Mora Rodriguez, and Ricardo Yepes Guevara. Measuring aslr implementations on modern operating systems. In *2013 47th International Carnahan Conference on Security Technology (ICCST)*, pages 1–6. IEEE, 2013.

[3] David Boles, Daniel Waddington, and David A Roberts. Cxl-enabled enhanced memory functions. *IEEE Micro*, 43(2):58–65, 2023.

[4] Jeongdong Choe. Memory technology 2021: Trends & challenges. In *2021 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, pages 111–115. IEEE, 2021.

[5] Compute Express Link Consortium. Compute express link specification, 2024.

[6] coreboot authors. Coreboot docs: Grub, 2024. `https://www.coreboot.org/GRUB2`.

[7] coreboot authors. Coreboot docs: Seabios, 2024. `https://www.coreboot.org/SeaBIOS`.

[8] Dasharo. Msi desktops: Initial deployment, 2024. `https://docs.dasharo.com/unified/msi/initial-deployment/`.

[9] Dasharo. Releases: Msi pro z690-a (wifi) (ddr4) dasharo release notes, 2024. `https://docs.dasharo.com/variants/msi_z690/releases`. Accessed at 20.01.2024.

[10] Brian Delgado and Karen L Karavanic. Performance implications of system management mode. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 163–173. IEEE, 2013.

[11] Devaev, Maxim. Pikvm faq, 2024. https://docs.pikvm.org/first_steps/.

[12] Tetsuo Endoh, Hiroaki Honjo, Koichi Nishioka, and Shoji Ikeda. Recent progresses in stt-mram and sot-mram for next generation mram. In *2020 IEEE Symposium on VLSI Technology*, pages 1–2. IEEE, 2020.

[13] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[14] George Hodgkins, Yi Xu, Steven Swanson, and Joseph Izraelevitz. Zhuque: Failure is not an option, it's an exception. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 833–849, 2023.

[15] Intel. Intel optane persistent memory - start up guide, 2020.

[16] Intel. Intel 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4, 2023.

[17] Intel. Intel 100 series and intel c230 series chipset family platform controller hub (pch) datasheet volume 1 of 2, 2024. https://www.intel.de/content/www/de/de/content-details/332690/intel-100-series-chipset-family-platform\-controller-hub-pch-datasheet-volume-1.html.

[18] Intel Corporation. Atx specification - version 2.2, 2024. https://cdn.instructables.com/ORIG/FS8/5ILB/GU59Z1AT/FS85ILBGU59Z1AT.pdf.

[19] Intel Corporation. eadr: New opportunities for persistent memory applications, 2024. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory\-applications.html.

[20] Intel Corporation. Intel optane persistent memory 200 series brief, 2024. https://www.intel.de/content/www/de/de/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html.

[21] Maciej Jakubowski and Piotr Sypek. Electromagnetic simulations with 3d fem and intel optane persistent memory. In *2022 24th International Microwave and Radar Conference (MIKON)*, pages 1–5. IEEE, 2022.

[22] Jungi Jeong, Jianping Zeng, and Changhee Jung. Capri: Compiler and architecture support for whole-system persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 71–83, 2022.

[23] Thomas Jew. Mram in microcontroller and microprocessor product applications. In *2020 IEEE International Electron Devices Meeting (IEDM)*, pages 11.1.1–11.1.4, 2020.

[24] Yussuf Khalil. Fpga-accelerated non-volatile memory access, October 27 2022.

[25] Kroll, Russell and Quette, Arnaud and de Korte, Arjen. Network ups tools user manual, 2024. `https://networkupstools.org/docs/user-manual.pdf`.

[26] Kropelin, Adam and Sibbald, Kern. Apcupsd user manual, 2024. `https://networkupstools.org/docs/user-manual.pdf`.

[27] Tianxi Li, Yang Wang, and Xiaoyi Lu. On the discontinuation of persistent memory: Looking back to look forward. *Memory*, 8:10, 2023.

[28] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–35, 2021.

[29] Mohammed Adnene Trojette. 7z(1) - linux man page, 2024. `https://linux.die.net/man/1/7z`.

[30] R Muralidhar, H Seshadri, V Bhimarao, V Rudramuni, I Mansoor, S Thomas, B Veera, Y Singh, and S Ramachandra. Experiences with power management enabling on the intel medfield phone. In *Proc. of Linux Symposium*, pages 35–46, 2012.

[31] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[32] PCI-SIG. Pci express base specification revision 4.0, version 1.0, 2024.

[33] Redis Ltd. Introduction to redis, 2024. `https://redis.io/docs/about`.

[34] The coreboot authors. Coreboot: Architecture, 2024. https://doc.coreboot.org/getting_started/architecture.html. Accessed at 28.01.2024.

[35] The coreboot authors. Coreboot docs: Output and consoles, 2024. https://www.coreboot.org/Console_and_outputs. Accessed at 15.01.2024.

[36] The coreboot authors. Coreboot source code configs/config.msi_ms7d25_ddr5, 2024. https://github.com/Dasharo/coreboot/blob/dasharo/configs/config.msi_ms7d25_ddr5. Accessed at 15.01.2024.

[37] The coreboot authors. Coreboot source code src/include/cpu/x86/tsc.h, 2024. https://github.com/coreboot/coreboot/blob/5191623149e5f15b869cd103597eb7a26f399bd9/src/include/cpu/x86/tsc.h. Accessed at 15.01.2024.

[38] The coreboot authors. Coreboot source code src/mainboard/msi/ms7d25/msi_id.s, 2024. https://github.com/Dasharo/coreboot/blob/ad90f0fbdd27842cc8a371747fc589fee4b40cae/src/mainboard/msi/ms7d25/msi_id.S. Accessed at 15.01.2024.

[39] The coreboot authors. Coreboot: Starting from scratch, 2024. https://doc.coreboot.org/tutorial/part1.html. Accessed at 22.01.2024.

[40] The coreboot authors. Welcome to the coreboot documentation, 2024. https://doc.coreboot.org/index.html. Accessed at 10.01.2024.

[41] The flashrom authors . flashrom - manual page, 2024. https://flashrom.org/classic_cli_manpage.html. Accessed at 09.02.2024.

[42] The Linux authors. clone(2): create a child process, 2024. https://linux.die.net/man/2/clone.

[43] The linux authors. date(1) - linux manual page, 2024. https://man7.org/linux/man-pages/man1/date.1.html. Accessed at 30.01.2024.

[44] The linux authors. execve(2) - linux manual page, 2024. https://man7.org/linux/man-pages/man2/execve.2.html. Accessed at 15.01.2024.

[45] The Linux authors. Linux source code arch/x86/lib/cache-smp.c, 2024. https://github.com/torvalds/linux/blob/v6.1/arch/x86/lib/cache-smp.c. Accessed at 10.02.2024.

[46] The Linux authors. Linux source: drivers/pci/setup-res.c, 2024. https://github.com/torvalds/linux/blob/master/drivers/pci/setup-res.c.

[47] The Linux authors. Linux source: include/arch/x86/kernel/e820.c, 2024. https://github.com/torvalds/linux/blob/master/arch/x86/kernel/e820.c.

[48] The Linux authors. Linux source: include/linux/mm_types.h, 2024. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/mm_types.h. Accessed at 25.01.2024.

[49] The Linux authors. Linux source: include/linux/sched.h, 2024. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/sched.h. Accessed at 25.01.2024.

[50] The Linux authors. Linux source: mm/memory.c, 2024. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/mm/memory.c. Accessed at 22.01.2024.

[51] tianocore/edk2 authors. Build and integration instructions, 2024. https://github.com/tianocore/edk2/blob/master/UefiPayloadPkg/BuildAndIntegrationInstructions.txt.

[52] Unified EFI Forum. Advanced configuration and power interface specification, 2017. https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[53] Lukas Werling, Yussuf Khalil, Peter Maucher, Thorsten Gröninger, and Frank Bellosa. Analyzing and improving cpu and energy efficiency of pm file systems. In *Proceedings of the 1st Workshop on Disruptive Memory Systems*, pages 31–37, 2023.

[54] Shimeng Yu and Pai-Yu Chen. Emerging memory technologies: Recent trends and prospects. *IEEE Solid-State Circuits Magazine*, 8(2):43–56, 2016.

[55] Taiyu Zhou, Yajuan Du, Fan Yang, Xiaojian Liao, and Youyou Lu. Efficient atomic durability on eadr-enabled persistent memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 124–134, 2022.

[56] Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Beyond BIOS: developing with the unified extensible firmware interface*. Walter de Gruyter GmbH & Co KG, 2017.