Karlsruhe Institute of Technology

FPGA-Accelerated Non-Volatile Memory Access

Master's Thesis
submitted by

cand. inform. Yussuf Khalil

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Jun.-Prof. Dr. Christian Wressnegger
Advisors:	Dipl.-Inform. Thorsten Gröninger
	Lukas Werling, M.Sc.

May 04 – October 27, 2022

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

Karlsruhe, October 27, 2022

Abstract

Intel Optane Persistent Memory (PMem) is a recent non-volatile memory technology that provides higher integration density than current DRAM-based memories. PMem offers traditional load/store semantics with byte addressability and may be used as additional main memory complementary to DDR4 modules. Access performance is, however, severely worse than with DDR4 memory to the point where CPU cores may become stalled for prolonged times while waiting for memory operations to complete. Previous works have shown that this behavior can negatively impact the runtime performance of unrelated processes running concurrently on the same system.

Asynchronous copy offloading has been proposed as a mechanism to overcome the performance implications of parallel accesses to PMem. However, previous implementations based on Intel I/OAT have been unsuccessful as I/OAT hardware is incapable of saturating Optane bandwidth.

In this thesis, we present the design of a FPGA-based PCIe accelerator device for asynchronous copy offloading. To reduce the latency impact of data transfers via the PCIe bus, we connect the PMem modules directly to the FPGA instead of the CPU and design a custom MMU optimized specifically for typical PMem use cases. Based on SR-IOV, we implement a multiplexing scheme that enables lock-free parallel command submission while maintaining proper process isolation without requiring kernel mediation. We weave the pieces together to offer userspace processes an interface that provides `memcpy()` semantics with asynchronous completion.

We show that our design is capable of saturating PMem's bandwidth while significantly reducing the CPU time spent for memory operations by up to 98.8 %. Regarding latency, we observe a slowdown by up to 100×.

Note: This work makes use of proprietary technology that was provided by Intel Corporation under the terms of a non-disclosure agreement. The respective passages are therefore censored in the public version of this document.

Contents

Chapter 1

Introduction

The memory hierarchy of contemporary computers typically encompasses four major components: registers, caches, main memory, and mass storage [17]. Caches and main memory are usually built using volatile SRAM and DRAM cells, respectively, whereas hard disks (HDDs) and NAND flash-based solid state drives (SSDs) are typically used for non-volatile mass storage. In recent years, however, a lot of research has been done towards new memory cell technologies and modified memory hierarchy designs. This research is not only motivated by the every-increasing demands for performance and storage space, but also by an issue known as the *memory wall*. Originally coined as a term in 1995 by Wulf et al. [79], the *memory wall* describes the performance implications that are to be expected when DRAM latency, bandwidth, and size do not keep pace with the increase in processing power. Although the authors feared that their "prediction of the memory wall is probably wrong," they were right in the sense that between 1980 and 2015, CPU performance has improved by a factor of about 10,000, whereas DRAM latency has merely seen a 10× improvement [17]. Similarly, regarding capacity, while DRAM-based memories have achieved a 128× increase in the time from 1999 to 2017 [59], *big data* and other current-era computing trends continue to cause an exponential increase in the amount of data being processed [14]. Naturally, this creates a strong demand for fast and large main memories.

Research has brought several ideas forward. For example, *processing-in-memory* [15] and *near-memory computing* [74] have been proposed as novel computing paradigms where processing hardware is placed either within memory modules or near to them to avoid having to move all data back and forth to a central processing unit. Further, new non-volatile memory cell technologies are currently being researched, e.g., STT-RAM, ReRAM, and PCRAM [19]. This thesis revolves around a specific recent memory technology by Intel called *Optane*. Despite Intel being secretive about the details of the *3D XPoint* cell

technology that powers Optane, it is commonly believed to be based upon PCRAM [7]. In essence, PCRAM (Phase-Change Random Access Memory) memory cells work by inducing heat into a material in order to make it switch between amorphous and crystalline phases [69]. The electric resistance level of the cell then changes depending on the phase. Multiple different crystallization states are possible by adjusting the duration of heat exposure, and thereby, a single PCRAM cell may store multiple bits of data. This property enables one of the main benefits of Optane: compared to current DRAM-based DDR4 memories, 3D XPoint features a 3.3× higher density [6].

Optane comes in two flavors: Intel offers both SSDs based on Optane technology [35] as well as byte-addressable memory modules, known as DCPMMs (Data Center Persistent Memory Modules) or simply PMem (Persistent Memory) [36]. We focus on the latter.

Looking at the memory hierarchy again, Optane PMem adds an interesting twist: the mere existence of mass storage distinct from main memory is currently necessitated by the fact that DRAM is volatile, and in addition, by its comparably low integration density versus HDDs and SSDs. Novel byte-addressable and non-volatile memory technologies such as Optane PMem have the potential to bring fundamental change to the memory hierarchy that has been mostly the same for several decades now. In its current form and given the way it is used in practice, PMem may be looked at as an additional level in the hierarchy: there is now both a *volatile* and a *non-volatile* main memory. Naturally, a fundamental question arises from the perspective of operating system designers and software engineers: *how and when should we use this new non-volatile main memory?*

Currently, there is no simple answer. Intel's marketing documents propose using PMem for virtualization hosts or database systems together with DDR4 memory (to be used as a cache for PMem) and emphasize benefits such as potential cost savings and faster restart times [37]. Researchers have come up with further ideas, e.g., specialized file systems such as NOVA [81], which exploits the properties of hybrid main memory architectures to achieve both high performance and strong consistency guarantees. However, a severely limiting factor for the practical applicability of Optane PMem is its performance. Yang et al. [82] have conducted an empirical study of PMem's performance characteristics and found that several of the assumptions that were used to design non-volatile memory-based systems before PMem was commercially available do not hold true for the final product. First, severe write amplification may occur for smaller writes as 3D XPoint uses a 256 B access granularity. Second, write performance scales badly with multi-threading: bandwidth begins

to drop as soon as more than two CPU cores perform parallel writes, even when using only 256 B accesses to avoid write amplification.

Werling et al. [78] have shown the second issue to be of particular importance for file systems. If only a limited number of threads may concurrently access the storage media in order to maintain performance, a file system would need to make use of semaphores for write operations. However, as a large amount of heterogeneous applications may use a file system at the same time, high lock contention then arises as another problem that again hurts overall performance.

They propose *asynchronous copy offloading* as a mechanism to solve the performance drawbacks of parallel PMem access. The general idea is to store all data that is to be persisted in fast DRAM first, and to have another external unit of hardware perform copies to PMem asynchronously. This approach is suitable for file systems as they typically enforce consistency via other means such as journaling [4] and do not necessarily require synchronous writes.

In this thesis, we aim to co-design specialized hardware and software (i.e., the operating system driver and user application interface) to implement an FPGA-based PCI Express device for the task of asynchronous copy offloading for PMem. We propose using our design as a platform for further research regarding out-of-order processing, file system acceleration, and optimized cache design for PMem.

In Chapter 2 we begin by introducing related work on the topic as well as the technologies that form the foundation for our design. Afterwards, in Chapter 3, we explore the design space and discuss possible options. Then, we describe what choices we made and elaborate on our final design. In Chapter 4, we conduct an evaluation of our approach with extensive benchmarks and discuss the results. Finally, we conclude our work in Chapter 5 with an outlook towards further possibilities with future research on the topic.

Chapter 2

Background

In this chapter, we want to take a deeper look at Optane PMem, the concept of asynchronous copy offloading as well as previous implementations, and the technologies we used for our design. Based on these fundamentals, we then present our design of an FPGA-based accelerator for asynchronous copy offloading in Chapter 3.

2.1 Optane Persistent Memory

Intel and Micron introduced their novel *3D XPoint* memory cell technology in 2015 [24] and Intel launched the first SSDs based upon 3D XPoint in 2017 under the brand name *Optane* [27]. The *Optane Persistent Memory* (PMem) that this thesis revolves around saw its general availability on the market about two years later in 2019 [26]. So far, there are two generations of PMem and another one to come, known as PMem 100, PMem 200, and PMem 300 [49]. Apart from performance improvements and reduced power consumption, PMem 200 introduced a feature known as eADR, short for *extended Asynchronous DRAM Refresh* [20]. PMem 100 already guarantees that all data that reaches the CPU's integrated memory controller will be persisted even in the event of a power failure. With eADR, PMem 200 takes this a step further: it is guaranteed that all data in the CPU's caches will be persisted when an unexpected power failure occurs. Platform hardware is required to provide enough residual charge to the CPU to ensure that it can still flush data after the power source has failed. The obvious implication is that application developers do not need to perform explicit cache flushes anymore to ensure that data is persisted, which was proven to be beneficial for performance [70]. Other works have taken more creative approaches to the possibilities enabled by eADR. For example, Zhang et al. [84] have presented *NBTree*, a lock-free concurrent implementation of

B^+ trees that leverages eADR to ensure that all threads always read persistent data without explicit synchronization. In July 2022, Intel announced that their Optane business unit will be shut down due to high losses, however, they clarified that they still plan to release PMem 300 to the market [2]. We solely used PMem 100 for this thesis.

PMem currently supports two different operating modes known as *Memory Mode* and *App Direct Mode* [34]. In Memory Mode, the Optane memory is used as the sole main memory and any DRAM-based DDR memory in the system serves as an additional cache level. There are no strong persistence guarantees in this mode, as newly written data may be held in the DRAM caches for prolonged time and is then lost in case of a power failure. In App Direct Mode, in contrast, PMem is exposed to the operating system as a separate memory area distinct from all DRAM-based memory. Mixing both modes is also possible in the sense that the memory modules can be partitioned to assign different amounts of space to each mode.

Modern Linux versions feature the `libnvdimm` subsystem along with the corresponding `ndctl` userspace tool to configure non-volatile memory modules such as Optane PMem [64]. Via `ndctl`, it is possible to configure several namespaces (i.e., partitions) that are then exposed as different device files in Linux's device file system. Two modes are available that behave as block devices with traditional `read()`/`write()` semantics, known as `raw` and `sector`. For our purposes, the `DAX`-based interfaces are more interesting: `fsdax` and `devdax`. Linux's `DAX` (direct access) infrastructure allows to bypass page caches when accessing non-volatile memory [13]. The `fsdax` mode leverages this functionality to allow processes to directly `mmap()` files in compatible file systems (e.g., `ext4`). `devdax` provides a character device through which programs may directly read and write raw data on the underlying device without a file system or other intermediaries. The approach we implement in this thesis resembles `devdax` in the sense that it also provides raw access to Optane memory. Implementations of file systems and other use cases are out of scope for this thesis and remain as future work. We use Optane PMem with a `devdax` namespace as comparison baseline for the evaluation in Chapter 4.

Even before Optane PMem was commercially available, researchers have put thought into possible applications and designs based on persistent memory modules that were already looming on the horizon. They typically employed software emulation [9] or treated DRAM as if it were persistent [60] to evaluate their ideas. However, actual tests using the final product such as the benchmarks conducted by Yang et al. [82] have found the real performance characteristics of PMem "to be more complicated and nuanced" than what was previously assumed. Using latency measurements and the hardware's

performance monitoring infrastructure, they have brought several unexpected insights about Optane's behavior to light.

First of all, they found that write latencies are typically about as long as with DDR4 memory. Read accesses, however, are 2 to 3× slower than with DDR4. Repeated writes to a small region eventually cause a latency spike up to 50 µs which hints that there is likely an internal remapping mechanism at play. Regarding bandwidth, a maximum of 6.6 GB s^{-1} may be achieved in the case of reads with a single Dual Inline Memory Module (DIMM), whereas write performance tops out at 2.3 GB s^{-1}. However, there is a caveat to the attainable bandwidth in practical scenarios: accesses with a size below 256 B are slow and may achieve only a fraction of the maximum bandwidth. This is caused by PMem's internal structure: although the DIMMs exhibit a byte-addressable interface, the 3D XPoint cells are built in 256 B arrays and thus form a block structure. Smaller write operations therefore cause *write amplification*: the DIMM's internal controller first needs to read the rest of a 256 B block before it writes it back again together with the new data. Finally, the achievable bandwidth depends strongly on the number of threads that concurrently access the Optane modules – higher concurrency generally results in smaller total bandwidth. This very much contradicts the modern trend towards increased parallelism with multi-core and many-core CPUs.

As Werling et al. [78] have shown, parallel accesses to PMem are in fact hurtful to a system's overall performance. Unlike with SSDs or HDDs, the total CPU utilization rises about linearly with the number of concurrent writer threads. The root cause for this issue is that the CPU's pipelines begin to stall when they have to wait for outstanding write operations to complete. In turn, there is less CPU time available for other processes.

2.2 Asynchronous Copy Offloading

Based on the idea that not all memory operations are necessarily required to be synchronous in some applications, Werling et al. [78] have proposed the concept of *asynchronous copy offloading* to combat the CPU overhead induced by parallel accesses to Optane Persistent Memory. They argue that the memory interface based on load/store instructions "takes away control from the operating system" in the sense that it cannot govern I/O scheduling anymore. In turn, threads that are I/O-bound on PMem devices also become CPU-bound. Hence, I/O stalls always lead to CPU stalls and reduce the practically usable CPU time. Consequently, their approach fundamentally relies on taking back control over I/O to allow the system to make progress even while operations on PMem are pending. Their idea is to move the effort required to perform copies from

the CPU to periphery hardware that executes them asynchronously in the background in order to lift the burden of stalling on memory instructions from the cores. To this end, they propose to perform accounting to measure the overall bandwidth consumption and automatically switch from synchronous operations to asynchronous ones as soon as the available bandwidth is saturated to proactively prevent stalls. Through this concept, they aim to preserve minimal latencies as long as possible to combine the best of both worlds.

Several implementations of asynchronous copy offloading for PMem have been created so far. For one, Werling et al. created a prototype that simply isolates all copy operations on a single CPU core to show that the approach is generally capable of reducing the overall CPU load. A first implementation that completely moved the copy tasks towards external hardware was built using Intel's *I/O Acceleration Technology* (I/OAT) [78]. Originally designed to reduce the CPU overhead for processing network packets, I/OAT essentially provides a mechanism to asynchronously copy data between the system memory and a periphery device (e.g., a network interface controller) [21]. The implementation simply leveraged I/OAT hardware to asynchronously move data between DRAM and PMem. However, the approach was unsuccessful because I/OAT was unable to saturate PMem's bandwidth: write operations were capped at about 500 MiB s^{-1}. Given that more than half of the possible bandwidth is lost, I/OAT was deemed unsuited as a driver for asynchronous copy offloading.

Maucher [58] presented *GPU4FS*, an implementation of asynchronous copy offloading that employs graphics processors. They went a step further and built a file system that runs on the GPU via compute shaders with the goal of alleviating the overhead induced by file system operations in the operating system kernel. In addition to simple `memset()` and `memcpy()` operations, GPU4FS allows submitting file system commands through ring buffers that are then processed asynchronously on the GPU. Although they are able to saturate the Optane memory's bandwidth to about 80 % with an AMD Radeon RX 6600 XT graphics processor connected via a PCIe 3.0 x8 link, latency is a severe issue with their approach: they experience delays of >1 ms for operations.

As we will discuss in Chapter 3, our FPGA-based implementation solely supports asynchronous accesses. However, we also provide software-based pseudo-synchronous access modes that rely on busy waiting. We leave proper support for synchronous accesses that allows automatic switching between synchronicity modes as described above as future work.

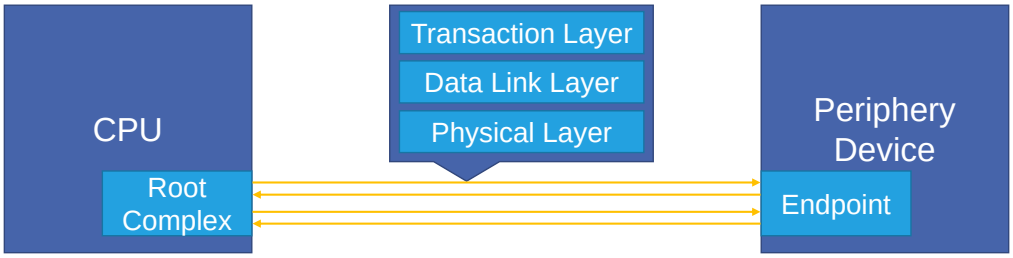


Figure 2.1: Exemplary PCI Express link with two lanes and no bridges between the root complex and the endpoint. Three layers make up the PCIe implementation stack: the *Transaction Layer*, the *Data Link Layer*, and the *Physical Layer*.

2.3 PCI Express

PCI Express (PCIe), short for *Peripheral Component Interconnect Express*, was introduced in 2003 as a novel bus for periphery hardware to succeed previously common interfaces such as PCI and AGP [63]. The latest version, PCIe 6.0, was finalized in January 2022 [83]. PCIe was designed with the goal to provide low cost, PCI compatibility on the software level, low latency, and high bandwidth all while covering different market segments from small embedded devices up to professional server systems [62, Section 1.1]. Today, it is commonly used for all kinds of periphery devices in a computer system, e.g., graphics processors, network adapters, SSDs, USB controllers, or FPGAs [17]. We introduce PCIe here as the FPGA-based design we implement in Chapter 3 interfaces with the base system via PCIe 3.0.

A single PCIe *link* between two PCIe components consists of one or more *lanes*, where each lane contains two pairs of differential signal wires for receiving and transmitting data [62, Chapter 1]. With the current PCIe 6.0 standard, a single lane can transmit 64 Gbit s^{-1} in each direction [83], whereas the older PCIe 3.0 that we use allows for 8 Gbit s^{-1} per lane and direction. In the simplest form, a link is established directly between the *root complex* and an *endpoint* device. The *root complex* interfaces with the CPU, and *endpoints* are the periphery devices that are to be connected with the system. Additionally, one or more bridges may exist on the path between the root complex and an endpoint. In modern x86 systems, the root complex is typically embedded within the CPU. An exemplary PCIe x2 link is depicted in Figure 2.1.

The PCIe interface is subdivided into three layers, namely the *Physical Layer* (PHY), the *Data Link Layer* (DLL), and the *Transaction Layer* (TL) [62, Section 1.5]. The PHY defines the electrical interface and performs conversion between the parallel device-side and the serial bus-side interfaces. The DLL is responsible for maintaining integrity and sequential ordering of data pack-

ets, and therefore provides error detection and correction as well as sequence numbers and performs packet retransmission, if necessary. Hence, it fulfills a similar purpose as the *Transmission Control Protocol* (TCP) in computer networks. Finally, the TL is where application data is transmitted in the form of *Transaction Layer Packets* (TLPs). TLPs can be divided into four categories [62, Chapter 2]:

- Memory transactions – e.g., `MWr` and `MRd` to perform memory-mapped writes and reads. These may either access system memory or an endpoint's *Base Address Register* (BAR) space. BARs are, in essence, memory-mapped areas that are provided by an endpoint.
- I/O transactions – e.g., `IOWr` and `IORd` to perform writes and reads in I/O space. These exist primarily for backwards compatibility with the old PCI standard.
- Configuration transactions – e.g., `CfgWr0` and `CfgRd0` to write and read *Type 0 Configuration Space*.
- Message transactions – i.e., `MsgD` and `Msg` for messages with or without data, respectively. In addition to vendor-specific messages, PCIe defines a set of standard messages, e.g., for power management, error signaling, or time measurement.

Some transactions, e.g., `MWr`, are *posted*, i.e., they do not require a subsequent response or completion signaling. For non-posted transactions, the submitting PCIe device expects a response via `Cpl` (completion without data) or `CplD` (completion with data) TLPs.

Among PCIe's features is *Single Root I/O Virtualization* (SR-IOV) [62, Chapter 9], which enables multiplexing PCIe endpoint devices through multiple *functions*. The device is then represented as a hierarchy with a *physical function* at the top and one or more *virtual functions* (VFs). Each VF may provide its own set of BARs and is distinctly addressable on the bus. SR-IOV therefore provides a bus-level isolation mechanism that allows to efficiently virtualize PCIe devices. A hypervisor may then allocate virtual functions to virtual machines, thereby allowing them to make use of accelerator hardware without requiring mediation from the hypervisor at runtime. As we show in Chapter 3, we make use of SR-IOV to multiplex our design for process-level address space isolation.

Compute Express Link (CXL) has evolved in the recent years as a new interconnect standard specifically for professional use cases [10]. It reuses PCIe's PHY, but defines its own protocols in the upper layers, namely `CXL.io` for PCIe-style device management and register access, `CXL.cache` for system

memory access, and `CXL.mem` for device memory access. In contrast to PCIe, CXL aims to provide lower latencies and full cache coherency specifically for memory operations in both directions. The first CXL-capable x86 CPUs are expected to arrive towards the end of 2022 with AMD's *Genoa* generation [1]. Regarding endpoint devices, Samsung has announced a CXL-based memory expander with DDR5 memory [73]. Intel *Agilex* FPGAs with CXL support are already available [30]. Early research works by Jung [51] have further demonstrated the design of an SSD with a CXL interface. We consider it an interesting endeavour for the future to explore a CXL-based implementation of our design.

2.4 Intel Avalon

The *Avalon* standard is a set of interface specifications designed by Intel for transporting data with single-ended digital signaling in Intel FPGAs [22]. We introduce it here since all data buses in our design are based on Avalon as we will show in Chapter 3.

Avalon defines a total of seven interfaces, most of which are rather trivial, e.g., for clock signals. For data buses, there are two important ones: *Avalon Streaming* (Avalon-ST) and *Avalon Memory Mapped* (Avalon-MM). Avalon-ST is solely designed for unidirectional data flow, whereas Avalon-MM provides memory-style semantics with bidirectional communication between a *host* and one or more *agents* and addressing for read and write operations. As we only used Avalon-MM, we give an overview of its most important signals here:

- `address` is set by a host to communicate an address. Avalon addresses may either represent symbols (e.g., bytes) or words. The size of a *word* equals the number of symbols transmitted in a single clock cycle. In case multiple agents are connected to a single host, an address map is used to assign address ranges to the different agents. The mapping is then resolved by a router on the bus.

- `read` is asserted to request data from an agent.

- `write` is asserted to transmit data to an agent. Note that, as there is only one address channel, only either a read or a write operation may be signaled in a single clock cycle.

- `writedata` transmits the data for a write operation.

- `readdata` transmits the data returned by a read operation.

- `readdatavalid` signals that the response of a read operation is ready and signaled via `readdata`.
- `byteenable` contains a bit for each symbol in a word and is used to activate the respective symbol lanes on the `writedata` channel or to request the respective symbols in a read operation.
- `waitrequest` is asserted by an agent to signal that it is not ready to receive requests. Wait requests generally affect both read and write operations – Avalon-MM does not allow using different signals for these.
- `burstcount` is used to signal burst transfers. Avalon-MM primarily supports incremental bursts: the starting address is transmitted in the first cycle and the agent is expected to calculate the following addresses itself in the next cycles. The `burstcount` signal defines the length of a burst in clock cycles. Line-wrapped bursts are also available, albeit we did not use them. These bursts start with an unaligned access within a line and wrap back around to the line's start address as soon as they reach the next line's border.
- `response` may be used by an agent to signal success or failure for an operation.

Avalon-MM is a rather simple bus interface – it forces in-order responses, does not feature bus-level power management capabilities, and only has a single address channel. Other on-die buses, e.g., AXI4 [3], are much more capable, however, Avalon-MM is enough for many purposes in the FPGA space.

Chapter 3

Design

In this chapter, we get to the heart of this thesis: the FPGA-based design we implemented for asynchronous copy offloading. We begin by discussing the requirements that need to be fulfilled for the approach to be successful. Afterwards, we explore the design space and take a look at the various options we have for our implementation. Based on these findings, we then present the implementation in full detail.

3.1 Design Space Considerations

We identify three major goals that we consider mandatory for a successful implementation of asynchronous copy offloading:

Quick Submission The idea behind asynchronous copy offloading arises from the overhead and wasted CPU time induced by wait cycles spent in the cores while write operations are pending. If submitting asynchronous copy commands takes equally long or even longer, nothing is gained. It is therefore imperative to make the submission process as quick as possible to be able to free up previously wasted CPU time for other purposes.

Low Latency Optane PMem is a non-volatile *main memory*. Given that main memory distinguishes itself from mass memory by the fact that it is actively used by threads running on the CPU as runtime storage, it is necessary to minimize access latencies as much as possible. This holds true nonetheless when an asynchronous interface is offered: even if the requirement of data becoming available immediately within the program flow is lifted, processes are still likely to require asynchronously requested data *soon* (from their perspective). Further, as described in Section 2.2, it may be worthwhile to allow mixing synchronous

and asynchronous operations in the future. In this case, minimizing latencies is an even more crucial goal in order to maintain CPU throughput. We therefore aim to create an implementation that does not rely on the kernel at runtime, but rather allows applications to directly interact with our hardware without an intermediary.

Bandwidth Saturation Saturating PMem in terms of its write bandwidth is difficult as we previously described in Section 2.1. On the one hand, a single thread running on a single CPU core is not enough to achieve the maximum possible bandwidth. On the other hand, parallel workloads with more than three threads quickly cause the total bandwidth to drop below the maximum. This leaves a very narrow window of two or three cores that may concurrently access PMem without hurting bandwidth in traditional setups with synchronous access. In contrast, we aim to design our hardware with the goal of being able to fully saturate PMem's bandwidth in as many scenarios as possible, including both single-threaded and highly multi-threaded workloads.

In the following we want to discuss certain aspects of the design that play a central role for achieving the three goals listed above as well as a a few side objectives.

3.1.1 Operations

First of all, it is important to decide on which operations, i.e., memory primitives, our implementation shall provide to user applications. As our primary goal is to implement asynchronous *copy* offloading, an obvious requirement is to have a copy operation, e.g., `memcpy()`. Notably, an implementation of `memcpy()` would be semantically different in our case from common `memcpy()` implementations in the sense that it is asynchronous. Unlike traditional `memcpy()`, it would not perform an actual copy operation, but submit a copy command to the FPGA and return immediately without waiting for the copy to be completed. For the scope of this thesis, an asynchronous `memcpy()` that implements copies between system memory and PMem is sufficient. However, there are a few more operations that may be desirable to have for real-world use cases:

`memmove()` `memmove()` only subtly differs from `memcpy()` in the sense that it allows copies within overlapping memory regions, which would otherwise result in undefined behavior when done with `memcpy()`. However, if our `memcpy()` implementation only allows copies between PMem and system memory and

not within PMem, there is no difference: the address spaces are disjoint and may not overlap in the first place.

If we are to implement copy operations within PMem (i.e., both source and destination are located in PMem), a correct, albeit inefficient implementation of `memmove()` could use two subsequent `memcpy()` operations that use system memory as scratch space.[1] A more efficient design would, of course, require an FPGA-side implementation of `memmove()`.

`memset()` `memset()` overrides a memory area with a specified pattern. Similar to `memmove()`, a software-side implementation is possible atop the `memcpy()` primitive by performing the `memset()` using scratch space in system memory. However, this similarly introduces significant latency and bandwidth overheads compared to an implementation with full hardware acceleration.

Synchronous Operations Despite asynchronous copy offloading being the high-level idea behind this thesis, we cannot ignore the potential need for synchronous variants of the operations we provide. After all, main memory is generally used in a synchronous manner in software applications. A simple implementation could be to wait until completion is signaled for an asynchronous operation. This can easily be done via busy waiting (i.e., a loop that only terminates when the completion signal is set by hardware).

Several optimizations are possible: first, we may reduce the implications that busy waiting brings along in terms of energy consumption by leveraging the new `UMWAIT` instruction [29] that is currently implemented in Intel *Tremont*, *Alder Lake*, and *Sapphire Rapids* CPUs [31]. `UMWAIT` is similar to the existing `MWAIT` instruction with the main difference that `UMWAIT` can be used by userspace applications (i.e., program code executed in x86 ring 3). In combination with `UMONITOR`, it is possible to configure an address that is subsequently monitored for store operations. `UMWAIT` exits when the previously configured address is written to or when a timeout deadline is reached. The processor core is put into an energy-efficient sleep state until then. However, unlike its kernel counterpart `MWAIT`, `UMWAIT` only allows shallow sleep states and therefore may still cause comparably high energy consumption when used for prolonged waits.

Depending on the latency of a single operation, it may make sense to schedule another thread during the wait time. In the absence of a hybrid threading model such as Go's *goroutines* [76], context switches to other threads are rather costly as they require switching to kernel mode first. We can leverage

[1] Note that the very same design could be used for a software-based `memcpy()` within PMem without support from the hardware side: simply use system memory as scratch space.

established techniques from the power management area [66] to make an online decision for how long we should wait in userspace before moving to another thread via the kernel: simply wait for half the time that a full context switch (i.e., the time until execution is fully transferred to another thread) is estimated to take, then perform the context switch. This approach has a competitive ratio ≤ 2 versus an optimal (offline) algorithm that knows the completion latency beforehand.[2] Note that a second context switch originates from the fact that it would be necessary to switch back to the thread that has initiated the synchronous operation. However, such an approach is only sensible if the latency lies in a range where it is sufficiently likely to stay below the time for a context switch.

Of course, instead of implementing synchronous operations atop asynchronous ones, another possibility is to provide them directly via traditional load and store instructions, e.g., `mov`. This, however, creates a requirement that highly depends on the employed bus interface and other design aspects: the PMem address space must be directly embedded into a process's virtual address space.

Finally, if synchronous operations can be executed with a reasonable latency, it may be feasible to conduct a hybrid approach that automatically switches between synchronous and asynchronous data movements depending on the current overall bandwidth utilization. This follows Werling et al.'s [78] original proposal for asynchronous copy offloading as we described in Section 2.2.

Atomics Atomic operations are a crucial tool to establish consistency and avoid race conditions in parallel applications. While it is generally possible to implement them in an asynchronous fashion as well, there is a strong case to have synchronous atomics as program behavior is often immediately dependent on the result of an atomic operation (e.g., in the case of a *compare-and-swap* commonly used to implement spinlocks).

Further, unlike the operations in the previous paragraphs, it is not possible to build atomics based on asynchronous copy primitives as that would not provide the required atomicity. Atomics therefore absolutely require a hardware-based implementation.

3.1.2 PMem Location

Implementing asynchronous copy offloading using FPGAs enables a new degree of freedom regarding the placement of PMem in the system's topology: we may attach the memory modules directly to the FPGA and instantiate the

[2] A more generalized form of this problem is also known as the *Ski Rental Problem* [57].

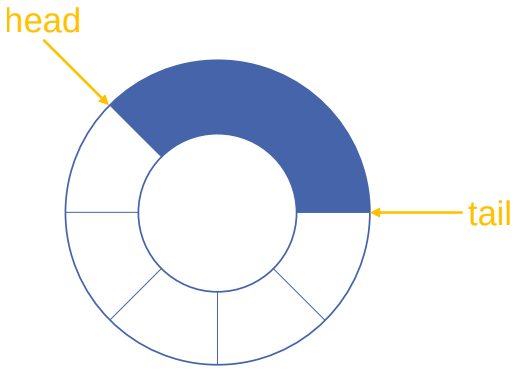


Figure 3.1: A ring buffer with enough space for eight elements. The three elements in the space between the head and tail pointers are currently occupied. A new element would be inserted at the position denoted by the tail pointer.

corresponding DDR-T controller there. We consider this a major advantage over previous approaches based on GPUs (as presented in Section 2.2). With a GPU-based implementation, the PMem modules are necessarily connected to the CPU's memory controllers, thereby incurring a roundtrip for each copy operation – data needs to be moved to the GPU after being read from DRAM or PMem, and then again from the GPU to the destination memory. This is obviously bad for latency as each byte of data needs to travel over the bus twice. In our quest to minimize latency, we therefore strongly make the case for having the Optane memory connected to the FPGA instead of the CPU. In the following subsections we generally assume this as a foundational design choice.

3.1.3 Command Submission

Asynchronous operations naturally require a mechanism to submit and queue commands. There are several options when it comes to how this may be implemented.

First of all, it is important to decide on the data structure that shall be used to buffer pending commands. *Ring buffers* (also known as circular arrays) are a common choice here [71]. Figure 3.1 shows an illustration of a small ring buffer with space for eight elements. In essence, a ring buffer is an array with a fixed size that wraps around to the beginning when the last position is reached, hence it forms a *ring*. Producers insert elements (i.e., commands in our case) by writing to the position that is denoted by a *tail* pointer, i.e., the position of the last element in the buffer. Consumers, on the other side, can fetch elements starting at the current *head* pointer (that points to the first

element in the buffer) up until the tail pointer is reached. The head and tail pointers are increased when elements are removed and added, respectively. When the tail pointer reaches the head pointer, the buffer is full. Depending on the implementation, present data may be overwritten or the producers need to stall until space is available again. Due to its properties with $O(1)$ insertion and removal and a fixed size, we deem ring buffers an appropriate choice for our use case. For correctness, we cannot allow pending operations in the ring buffer to be overwritten and therefore require that command submission is stalled as soon as the buffer is full. In turn, it is imperative that the ring buffer is large enough that stalls occur infrequently. An appropriate size may be determined via benchmarks and remains as an implementation detail.

Regarding placement, there are essentially two options where command buffers may be located: either in main memory or in FPGA-side memory. In the latter case, this would also imply that the FPGA has to manage the buffers itself, whereas software only needs a mechanism to send commands to the FPGA (e.g., via memory-mapped I/O (MMIO)). However, this may prove advantageous regarding latency – if buffers are located in main memory, several steps are required until hardware can start executing a command: first, the command descriptor needs to be stored in main memory. Then, hardware would need to see the new command. Another design choice arises at this point: we may either send explicit notifications to a designated *bell register* (again, via MMIO), or have the hardware continuously poll the system memory for new commands. In any case, the command would finally need to be transmitted to the FPGA. It seems obvious that storing command buffers on the FPGA side is the most efficient solution in terms of latency.

However, buffer storage capacities at the FPGA side are limited: while certain on-die memory in the form of SRAMs may exist, it may quickly run full in high load situations. Depending on the amount of buffers as well as their size, it may become necessary to involve additional memory based on DRAM or similar. This induces a certain amount of complexity for device-managed command buffers. Further, the overhead described above regarding latency for host-managed ring buffers can be hidden by having the FPGA prefetch commands while others are still executing. It therefore seems doubtful whether the latency advantages of managing command buffers on the FPGA itself is beneficial enough to outweigh the implementation complexity. We may explore a design with FPGA-side buffers in the future. For now, we have chosen to manager buffers in system memory as we will show in Section 3.2.1.

The next question that arises is the amount of buffers necessary. We may have one buffer per operation type, or alternatively, one buffer for all operations. This choice mainly depends on whether ordering is required with respect to

operations with different types. If order of execution is important, there is no point in having different buffers belonging to the same execution thread: a numbering mechanism or similar would be required to establish the order between the buffers. We can look at two real-world examples for inspiration here: first, the NVMe protocol that is commonly used for communication with modern PCIe-based SSDs uses multiple parallel command queues with no intertwined ordering [61]. Second, in contrast, programs designed for common processors are naturally sequential within a single execution thread. Although modern processors often execute instructions out-of-order, semantic consistency is still guaranteed: programs behave as if all instructions were executed in-order, including memory instructions [17]. This is a rather fundamental decision regarding the purpose of our design: we do *not* aim to build another SSD, but rather something that fulfills a similar use case as main memory, just with an asynchronous interface. For this reason, we propose having a single ordered buffer for all kinds of operations. As we will show later in Section 3.2, however, this does not reflect the current implementation. Unified command buffers therefore remain as future work.

Nonetheless, there is a different reason to have multiple buffers: concurrency among threads. There is inherently no order between memory operations executed by threads running in parallel, except where explicitly enforced via synchronization mechanisms. Similarly, no dependencies between threads should exist in our model and synchronization-free concurrent command submission from parallel threads is desirable. We therefore argue that there should be one command buffer per thread that makes use of PMem.

3.1.4 Completion Signaling

In the previous subsection we discussed how commands may be submitted. After an asynchronous command has completed, it is necessary to inform the submitting thread. Notably, there may be multiple points in the execution process where it makes sense to notify the software. In the case of a copy operation that reads from PMem (i.e., writes to system memory), it is simple: software may expect to be notified when data is available in system memory to be used. However, when we swap source and destination (i.e., copy from system memory to PMem), there are two points that may be interesting for a user application: first, when the data to copy has been fully read from system memory, and second, when the data was successfully written to the PMem device. Both are semantically relevant: while the former allows software to free the buffer in system memory, the latter is important to establish consistency guarantees. Only the latter is strictly necessary.

The options for how completion signaling may be implemented depend

strongly on where the buffers are placed (as discussed in Section 3.1.3). If buffers are placed in system memory, signaling may simply be done by setting a bit in the command descriptor or by advancing the ring buffer's head pointer (in the case of one single completion signal per operation). Things are more complicated if we let the FPGA manage all command buffers, as there is no predestined location in system memory belonging to each command. Interrupts seem unsuited due to their large overhead, especially in situations with many small operations. A simple solution would be to introduce a monotonic counter that provides a unique identifier for each operation. The FPGA may then signal command completion by writing the last completed command's ID to a preconfigured location in system memory.

3.1.5 Bus Interface

In Chapter 2, we introduced the PCI Express bus. PCIe is the traditional choice for periphery hardware of all kinds with high bandwidth demands, e.g., graphics cards, solid state drives, or as in our case, FPGA-based accelerators.

For a PCIe-based implementation, there are several relevant variables within the design space, which influence achievable results regarding bandwidth, latency, and others. In the following we will iterate over them and discuss sensible choices.

SR-IOV As described in Section 2.3, SR-IOV is a PCIe feature that allows multiplexing a single PCIe device (i.e., a single physical function) via multiple *virtual functions*. Although originally designed for system virtualization purposes, using SR-IOV is an interesting option for our use case, too. For multi-threaded workloads that require isolation on the system level (e.g., due to threads being located in different processes or virtual machines), it is necessary to reflect that isolation on the hardware side. While commercial PCIe hardware such as graphics cards often provides multiplexing via concurrent virtual address spaces, they usually do not employ SR-IOV to create an immediate binding of address spaces at the bus level [52]. This has a negative implication on the system side: as several distinct virtual address spaces share the very same physical or virtual function, user applications may not be given direct control over a function without breaking isolation. In turn, it becomes necessary to involve a privileged instance (typically, the operating system kernel) for runtime operations.

We therefore propose to go a step further by not only viewing SR-IOV as a tool for efficient system virtualization, but also as a mechanism for process isolation. By creating a 1:1 mapping between virtual functions and device-side

virtual address spaces we can allow the operating system to safely hand over direct control over a virtual function to the process that owns the corresponding virtual address space. Given that PCIe allows a single physical function to have up to 65535 virtual functions [62, Section 9.3.3], we believe that is enough to satisfy the amount of concurrent threads running in a typical system, especially given that only a minority of these threads is likely to make use of Optane PMem.

Such an SR-IOV-based isolation mechanism may, however, only provide the necessary security guarantees if an IOMMU is present in the system that is capable of isolating I/O address spaces with per-VF granularity. Otherwise, handing direct control over VFs to user processes leads to situations where untrusted applications gain the power to read the memory of other applications or even the entire physical memory.

BARs PCIe allows a single physical function that employs *Type 0 Configuration Space* to have up to six 32 bit or up to three 64 bit base address registers (BARs) [62, Section 7.5.1.2]. The same amounts of BARs can be separately configured for virtual functions in the *SR-IOV Extended Capability* [62, Section 9.3.3].[3]

Some BAR space needs to be reserved for configuration purposes. All configuration that requires elevated privileges (e.g., address space setup) may be handled via a BAR on the physical function. Depending on how the command submission process is implemented (i.e., whether command queues are stored in system memory or handled at the device side) it may be necessary to reserve a BAR in each virtual function in order to serve as bell register and possibly for command queue configuration.

However, if buffering commands in system memory is to be avoided (as described as an option in Section 3.1.3), it is possible to use virtual function BARs as command submission ports. The command descriptor may then be encoded via the BAR address (e.g., it may contain a virtual address in the device-side virtual address space), the data submitted when writing to the BAR, and possibly also via the BAR number. Notably, as the `MWr` TLPs that PCIe employs for writing to device BARs are posted [62, Section 2.4] (i.e., they do not require subsequent completion signaling), it can be expected that no greater delays or stalls are incurred on the CPU side than when writing to DRAM-based system memory.

[3] On a side note, the *Resizable BAR* feature in PCIe explicitly prohibits resizing a 32 bit BAR to a 64 bit one [62, Section 7.8.6]. Otherwise, it would be possible to setup six 64 bit BARs this way.

TLP Size As we described in Section 2.3, the *Transaction Layer* is the uppermost level in the PCIe protocol stack where application data is transmitted. Depending on the endpoint's as well as the root complex's capabilities, a single transaction layer packet (TLP) may transport a payload with a maximum size between 128 B and 4096 B [62, Section 7.5.3.4]. According to the PCI-SIG, common implementations support a maximum payload size of either 256 B or 512 B [12]. Due to the reduced overhead for header data, higher payload sizes are generally favorable in terms of bandwidth. However, there is a tradeoff: larger payloads also incur higher latency, and depending on the use case, it may be beneficial to have partial transfers arrive early instead of optimizing the overall turnaround time of a single transfer. This is especially true given that a single PMem DIMM may achieve a read bandwidth of less than 7 GB s^{-1} [82], whereas even the rather old PCIe 3.0 delivers a bandwidth of about 16 GB s^{-1} via a x16 link [11]. In turn, it is unnecessary for our goals to fully saturate the possible PCIe link bandwidth. For a PCIe-based implementation, it may therefore be worthwhile to conduct benchmarks using different TLP sizes.

Port Bifurcation A single PCI Express link consists of up to 32 lanes [62, Section 1.2]. In general, the link width equals the port width. However, it is possible to *bifurcate* a port into several smaller links, e.g., a x16 port may be split into four x4 links with distinct physical functions on each link. Using bifurcation, it may be possible to split the total available bandwidth across applications in a coarse-grained but simple manner in order to achieve better fairness (i.e., equal distribution of bandwidth across applications) or quality of service (QoS) guarantees for specific applications. The need for a mechanism that enables process isolation on a bandwidth level is underlined by the mere existence of Intel's *Memory Bandwidth Allocation* (MBA) [18] technology that is part of a larger set of resource distribution strategies in their current processors known as *Resource Director Technology* (RDT) [40]. Previous work by Xiang et al. [80] has shown that clever use of MBA can be beneficial for overall system performance. We therefore deem it worthwhile to explore potential performance isolation mechanisms such as bifurcation in the future.

3.1.6 Memory Management

Based on the design choice to connect the Optane memory directly to the FPGA instead of the CPU (as described in Section 3.1.2), another new freedom opens up in the design space: we are not bound to the CPU's address translation mechanisms anymore and may freely design a memory management unit (MMU) specifically tailored to our needs. Modern 64 bit x86 CPUs employ a page table hierarchy with four stages for virtual address translation with a

default page size of 4 KiB [29]. 2 MiB and 1 GiB *huge pages* with 3-level and 2-level translations, respectively, are available too.

Regarding Optane PMem, we make two observations that influence our proposed virtual memory architecture:

1. Optane DIMMs are sold in sizes between 128 GiB and 512 GiB [37]. In contrast, DDR4 specifies DIMMs with a maximum size of 64 GiB [67].

2. As explained in Section 2.1, one of the currently provided ways of interacting with PMem is via a device file, through which applications can explicitly request a share of the available Optane memory. With such an interface, only few special applications in a system, e.g., databases, typically make use of PMem.

We therefore argue that there is no strong benefit to having fine-granular pages as internal fragmentation is less of an issue with large memories and few user applications. Instead, it seems favorable to prioritize address translation latency over allocation granularity. For minimal latency it is desirable to have a translation mechanism that is as simple as possible.

In consequence, we propose using single-level page tables. The translation process is extremely simple in this case: the upper bits of a virtual address are simply exchanged with the bits in the page table entry to calculate the physical address. The corresponding entry can simply be determined by using the very same bits from the virtual address as an index into the page table. In contrast, even segmentation would involve more complexity as it requires an arithmetic addition. Replacing the bits in the address, however, is very simple to realize in digital circuitry via trivial wiring.

To further reduce latency, it is also desirable to have the page tables stored in on-die memory on the FPGA. In consequence, an implementation should use a page size that is large enough that a sensible amount of page tables (for multiple virtual address spaces) can be held in on-die memory, but also as small as possible.

3.1.7 Command Processing Order

Performing memory operations in an asynchronous fashion opens up the possibility to execute them out-of-order with a large leeway regarding latency. As we described in Section 2.1, previous works have shown that the achievable bandwidth of Optane PMem highly depends on the access patterns. Based on a successful asynchronous memory implementation, we may explore the design of out-of-order processing algorithms in future work. We hope that

such a design can improve the average throughput of Optane memory modules without compromising the proper functioning and performance of userspace software when memory is accessed asynchronously. Out-of-order processing further introduces an interesting challenge regarding the support of mixed synchronous and asynchronous operations that we described as a future extension in Section 3.1.1. Synchronous operations would certainly need to be prioritized as they have stricter latency requirements. Ideally, we find a way to perform operation reordering that includes both synchronous and asynchronous accesses and provides the necessary quality-of-service guarantees for synchronous ones. However, we do not dive deeper into this topic in this thesis, but rather leave the idea as an interesting option for future research.

3.2 Implementation

In the previous section we have discussed the possibilities that the design space for FPGA-based asynchronous copy offloading has to offer and have established several fundamental choices for an optimal design. We will now present our actual implementation in full detail in a bottom-up approach: we start at the hardware (i.e., FPGA) side and move further upwards via the kernel driver towards userspace.

3.2.1 Hardware: DDR4 Variant

We based our design on an FPGA from the Intel Stratix 10 DX series [43], more specifically the `1SD280PT2F55E1VG` variant that is found on Intel's Stratix 10 DX Development Kit [44]. The FPGA was primarily chosen for its Optane support. In addition, one or more *P-Tiles* provide PCIe 4.0 hard IP with 16 lanes each and support for SR-IOV.

The `1SD280PT2F55E1VG` specifically features four P-Tiles (i.e., a total of 64 PCIe lanes) and 2.8 M logic elements [33] – more than enough for our needs. It is further rated at the highest possible speed grade, allowing for a clock tree speed of up to 1 GHz [42]. The development kit is constructed as a PCIe add-in card (AIC) where 16 lanes of a single P-Tile are exposed via the AIC's gold fingers, making it suitable for easy integration into off-the-shelf PC and server systems. A secondary FPGA, namely an Intel MAX 10, is present on the board that serves as board controller and may be used to control periphery circuitry on the development kit such as the clock generators. Some of the Stratix 10 FPGA's memory I/O banks are wired to on-board DDR4 memory, whereas the others are connected to two DIMM slots that may hold either DDR4 or Optane PMem modules. This theoretically allows us to implement a

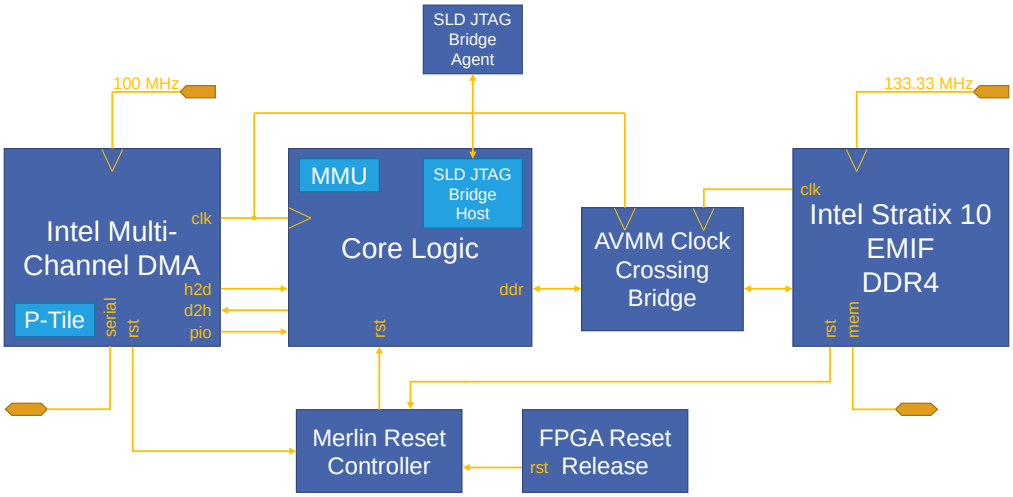


Figure 3.2: Hardware design overview with all instantiated IP blocks and the most important data signals, as well as clocks and reset signals. *Core Logic* denotes our own IP.

dual-channel design, however, as we will explain later in this subsection, we currently only support single-channel memory. Dual-channel operation with memory interleaving remains as future work.

Regarding embedded memory, all Stratix 10 DX devices contain *Memory Logic Array Blocks* (MLABs) as well as M20ks. MLABs repurpose the logic elements to provide 640 bit storage capacity each, whereas M20ks are 20 kbit SRAMs interleaved within the FPGA logic fabric. Some Stratix 10 DX variants additionally feature one or more large 47.25 Mbit eSRAM blocks or several gigabytes of on-package HBM2 memory. However, neither of those are present in the `1SD280PT2F55E1VG`.

3.2.1.1 Overview

Figure 3.2 depicts a coarse overview over the design we built on the Stratix 10 FPGA. The design was created and compiled using Intel Quartus Prime Pro 22.2 and all included intellectual property (IP) blocks (except, of course, our own logic) are provided by Intel as part of the Quartus tool suite. In the following we will iterate over all IP and describe their purpose in the design, and where important, their precise configuration. Note that we begin by describing a DDR4-only variant. A version with support for DDR-T (i.e., Optane) with the resulting differences is shown later in Section 3.2.2.

3.2.1.2 Multi-Channel DMA P-Tile Intel FPGA IP for PCI Express

The implementation is heavily influenced by the DMA engine we are using, namely *Multi-Channel DMA P-Tile Intel FPGA IP for PCI Express* (MCDMA) [48]. In short, MCDMA provides software-controlled, parallel data movement between system memory and the FPGA in both directions. It fully encapsulates the PCI Express P-Tile and handles all required bus configuration. In this regard, all PCIe signals are configured to be routed towards the P-Tile via the MCDMA IP. This includes the 100 MHz PCIe reference clock signal that the development kit board buffers via a Renesas `9ZML1252EKILF` clock buffering IC as well as the differential I/O signals for each lane that are wired directly from the board's gold fingers to the FPGA [50]. We configured the P-Tile to operate in PCIe 3.0 mode with 16 lanes. We may upgrade to PCIe 4.0 in the future if performance demands justify it. In PCIe 3.0 x16 mode, the IP uses a 512 bit-wide interface with a 250 MHz clock for all data signals. Notably, we use this clock signal to drive our entire core logic. For device identification, we configured the PCIe vendor ID to `0x3345` and the device IDs to `0x1` (PF) and `0x2` (VFs).

Regarding data transmission to user logic, MCDMA essentially provides three buses based on the Avalon-MM interface standard that we described in Section 2.4. The `H2D` (host-to-device) interface transmits data that was fetched from system memory, whereas `D2H` (device-to-host) requests data from the user logic for transfer into system memory. Both use 64 bit-wide addresses that are built from the address provided by software as well as the identifier of the VF that was used to submit the transfer request. It is noteworthy that MCDMA alternatively supports Avalon-ST interfaces for these buses, however, due to the lack of a proper addressing mechanism in the Avalon-ST variant, we opted for Avalon-MM. A third interface, `PIO`, is used to handle reads and writes to `BAR2`. The IP allows configuring this BAR freely according to the designer's demands. As we will show in Section 3.2.1.8, we use this BAR for virtual address space configuration.

Via `BAR0`, MCDMA exposes two registers: the *Queue Control and Status* Register (`QCSR`) and the *Global DMA Control and Status Register* (`GCSR`). A visual depiction of the registers is given in Figure 3.3.

The `QCSR` register space consists of a number of 256 B configuration objects corresponding to host-to-device as well as device-to-host copy command queues. Each object describes the configuration of a single ring buffer stored in system memory, i.e., its start address, size, and head and tail pointers. Notably, the tail pointer forms the *bell register* we proposed for fast command notification in Section 3.1.3. For quick access, the IP further allows writing the address of the last fetched command descriptor to a location in system memory (MCDMA names this feature *writeback*). This must be explicitly enabled both

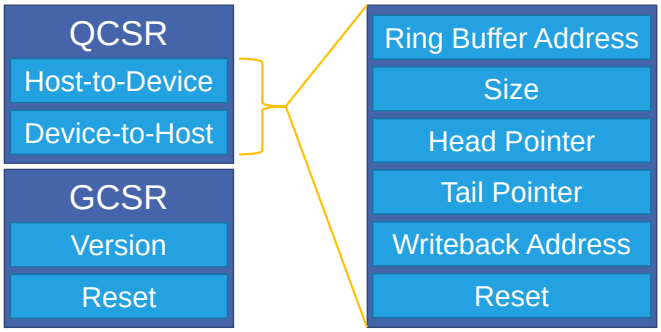


Figure 3.3: MCDMA's *Queue Control and Status* and *Global DMA Control and Status* registers. The `QCSR` is used to configure the command queues, whereas the `GCSR` merely exposes version information and provides a mechanism to reset the entire DMA engine.

for the entire ring buffer as well as for each command individually. As we will show in Section 3.2.4, we use this capability to stall command submission as soon as a ring buffer runs full. We would otherwise need to query the ring buffer's current head pointer via a costly MMIO read to the `QCSR`.

In the `GCSR` register, MCDMA merely exposes information such as its version number and a mechanism for a software-triggered reset.

The command descriptors in the ring buffers have a size of 32 B and contain all necessary information, i.e., source and destination addresses, transfer size, and a few control bits, e.g., to mark the descriptor as invalid or to enable the writeback feature described above. Figure 3.4 visualizes the contents of a command descriptor.

For our design, we enabled SR-IOV on the P-Tile with a total number of 511 virtual functions, i.e., the maximum that a P-Tile supports on a single physical function and without bifurcation. We then configured MCDMA to provide one pair of ring buffers per each virtual function. This forms the foundation for the SR-IOV-based isolation mechanism we described in Section 3.1.5. In this configuration, each virtual function has its own `QCSR` that allows configuring precisely the two ring buffers of that virtual function. This allows our kernel driver to let userspace configure and control the ring buffers by itself without breaking isolation as we will show later in Section 3.2.3.

For the sake of completeness, we would like to note that there are two additional, optional interfaces offered by the MCDMA IP: `BAM` (*Bursting Avalon Master*) and `BAS` (*Bursting Avalon Slave*). `BAM` allows software to perform MMIO reads and writes on a BAR and forwards all requests towards user logic. `BAS` enables user logic to perform arbitrary DMA requests to system memory. Although `BAM` would allow us to implement BAR-based synchronous operations

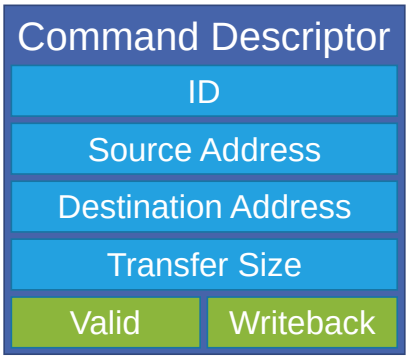


Figure 3.4: The contents of a command descriptor, as stored in the ring buffers that were previously configured via the `QCSR`. In the case of write (host-to-device) commands, *source address* references an address in the system memory, and *destination address* is handed to the FPGA-side user logic. The opposite is true for read (device-to-host) commands.

(as described as a potential goal in Section 3.1.1) it cannot be enabled together with the multi-channel DMA functionality. `BAS` does not have this limitation, however, we do not see how it could be useful for our purposes as long as asynchronous read and write commands are already entirely handled by the multi-channel DMA feature. In any case, using `BAM` seems problematic at the current time as Intel has documented several problems in their implementation regarding performance and potential PCIe packet corruption.

Although we have opted to use MCDMA for our initial implementation, we do not believe that it is the optimal choice for several reasons. First, we observed several issues with the MCDMA IP:

1. The last command descriptor in a 4 KiB page must be a special *link descriptor* that does not contain a command, but rather references the next descriptor's memory location, i.e., the position where the ring buffer continues. This is required even if the ring buffer is fully continuous across pages in the I/O virtual address space, and MCDMA halts operation on the respective command queue if software does not adhere to this rule. We consider this an unnecessary limitation as it merely introduces unnecessary latency overhead when the ring buffer crosses page boundaries.

2. A known bug that Intel mentions in their documentation is that MCDMA may lose data on other queues when a single queue is being reset by software. This may allow processes to disturb the correct execution of other processes that concurrently use our accelerator. As this is merely

an implementation issue that Intel may fix in the future and not a design flaw, we chose to ignore this problem for now.

3. MCDMA supports burst transfers for all Avalon-MM transactions with a burst length of up to eight clock cycles. However, it tends to create rather odd burst sequences. For example, on the `D2H` interface, we observe 1024 B transfers to be executed with three bursts: first, a single clock cycle, then eight cycles, and finally, seven cycles. Instead, it would be more efficient to emit two bursts with an equal length of eight cycles. MCDMA's behavior may cause decreased performance regarding Optane's 256 B access granularity (as described in Section 2.1) as long as we execute all commands in-order.

4. The `H2D` and `D2H` buses always idle for a single clock cycle between bursts, even in high load situations. This introduces an unnecessary latency overhead and reduces the achievable bandwidth.

Apart from these issues, there are several fundamental aspects that make MCDMA a subpar choice for our purposes. First, we argued that unified command buffers that contain commands of all types are desirable in Section 3.1.3. This is inherently not possible with MCDMA. Further, the Avalon-MM interfaces make a potential implementation of out-of-order processing (as suggested in Section 3.1.7) in the future difficult: Avalon, unlike other buses such as AXI [3], does not support out-of-order responses to transactions. Therefore, an implementation would need to perform complex and costly reordering of responses, which may reduce or even nullify any performance gains achieved by executing read and write operations out-of-order to optimize Optane access patterns. Finally, although implementing other operations than asynchronous copies may be possible via the `PIO`, `BAM`, and `BAS` interfaces, this seems like a rather cumbersome approach with difficult-to-predict performance. In conclusion, we therefore propose implementing a custom DMA engine specialized for our needs in the future. This would also allow us to make the most of the possibilities offered in the design space for a PCIe-based design as it alleviates the constraints imposed by MCDMA.

3.2.1.3 External Memory Interfaces Intel Stratix 10 FPGA IP

The *External Memory Interfaces Intel Stratix 10 FPGA IP* (EMIF) is essential for our needs as it provides the controller implementation for all memory protocols supported by FPGAs from the Intel Stratix 10 series, including the DDR4 interface that we use here [23]. As shown in Figure 3.5, the memory interface blocks in Stratix 10 FPGAs are organized as a hierarchy of *I/O columns*,

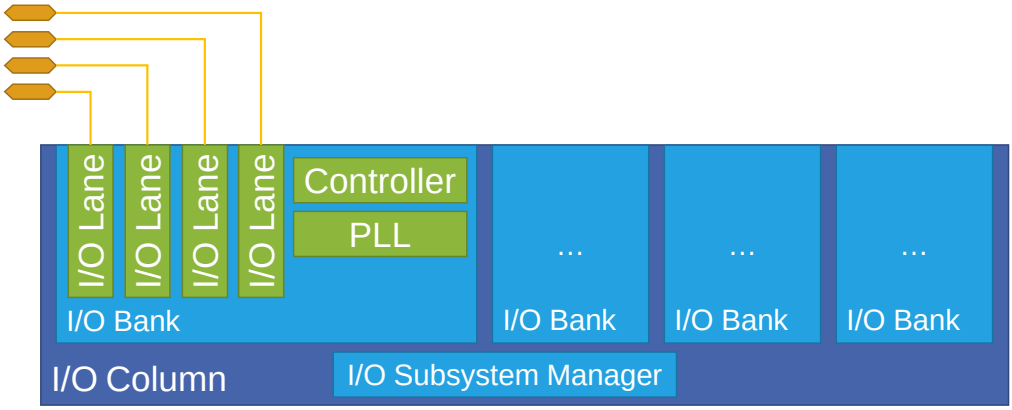


Figure 3.5: Coarse overview over the layout of an exemplary I/O column with four I/O banks in a Stratix 10 FPGA. Up to twelve signals may be wired to a single I/O lane, hence an I/O bank can drive up to 48 signals. Operating a DDR4 DIMM requires multiple I/O banks within a single I/O column.

I/O banks, and *I/O lanes*. The `1SD280PT2F55E1VG` features two I/O columns, one with ten I/O banks and the other one with nine I/O banks. Every I/O column further contains one *I/O subsystem manager* with an Intel Nios II processor core that is used to calibrate the data signals as well as important periphery signals. Each EMIF instance is assigned to one or more I/O banks within a single I/O column. Every I/O bank contains hard memory controllers for certain memory standards (including DDR4), phase-locked loops (PLLs) to generate I/O clock signals, and four I/O lanes. Each I/O lane, in turn, provides buffering and termination logic for twelve differential signals. These signals then lead to the external memory.

We configured our EMIF instance to support one Micron `MTA9ASF1G72PZ-2G9E1UG` DDR4 single-rank RDIMM with ECC. In this configuration, DDR4 uses one strobe signal (DQS) for eight data signals (DQ) with a total of 72 DQs, i.e., a total of nine DQS groups. One I/O lane is required per x8 DQS group, and a single I/O bank contains four I/O lanes, hence we need at least three I/O banks. Including periphery DDR4 signals (e.g., for addressing and commands), the setup requires a total of four I/O banks on the FPGA, leaving more than enough resources for additional memory channels in the future. A secondary EMIF instance would be required as a puzzle piece for a future dual-channel memory implementation in addition to the necessary memory interleaving logic that would need to be added to our core.

The development kit board features a Skyworks `SI5332A-C-GM2` clock generator that has several of its outputs wired to the FPGA's I/O banks [50]. We configured the IC to emit a clock signal with 133.33 MHz that drives the PLLs,

which are in turn setup to generate a 1066.66 MHz clock that is used to operate the DIMM. Note that even though we operate the memory with only 2133 MT s^{-1}, the DIMM supports a speed of up to 2933 MT s^{-1}. We configured the memory timings according to the values that DDR4 and Micron specify for the 2933 MT s^{-1} speed bin.

When used in DDR4 mode, EMIF derives a clock signal for user logic from the I/O PLLs that equals a quarter of the memory's clock, i.e., 266.66 MHz in our case. We do not use this signal to drive any of our own logic. To interface with user logic, EMIF exposes the `ctrl_amm` interface. `ctrl_amm` is an Avalon-MM interface with 64 B-wide data signals and a maximum burst length of 128 cycles. EMIF's controller provides buffering capacity for up to 64 pending read transactions.

3.2.1.4 Reset Release Intel FPGA IP

The *Reset Release Intel FPGA IP* is strictly required in all Stratix 10 designs [41]. Its purpose is to hold all logic in the FPGA fabric in reset state until the on-die *Secure Device Manager* has finished configuring the FPGA with a design bitstream provided by an external configuration host. To this end, it provides a signal named `nINIT_DONE` that asserts low as soon as configuration is done. We feed this signal to the *Merlin Reset Controller* that we describe in the next subsection.

3.2.1.5 Merlin Reset Controller Intel FPGA IP

The *Merlin Reset Controller Intel FPGA IP* combines and synchronizes a set of multiple reset inputs into a single reset output [39]. In our design, we instantiate it to create a reset signal that keeps our core logic in reset state until the FPGA and other essential IP blocks are initialized and the external memory is calibrated. We therefore feed it with three different reset inputs from the MCDMA IP, the EMIF IP, and the reset release IP.

3.2.1.6 SLD JTAG Bridge Agent Intel FPGA IP

While not strictly necessary for the functionality of our design, the *SLD JTAG Bridge Agent Intel FPGA IP* served an important purpose for debugging our implementation and we list it here for the sake of completeness. In combination with the *SLD JTAG Bridge Host Intel FPGA IP* that is instantiated within our core partition, it acts as a bridge on the development kit's JTAG chain, i.e., it allows debugging logic within our core to communicate with the host system. In turn, it enabled us to debug our core logic's signals and register states using Intel's *Signal Tap* logic analyzer [38].

3.2.1.7 Avalon Memory Mapped Clock Crossing Bridge Intel FPGA IP

In Section 3.2.1.2 we already explained that we use the 250 MHz clock signal provided by MCDMA to drive our core logic. As EMIF uses a different clock signal with 266.66 MHz, we cannot exchange data between our core logic and EMIF directly. Here the *Avalon Memory Mapped Clock Crossing Bridge Intel FPGA IP* comes to play: it implements asynchronous FIFO buffers and uses those to connect two Avalon-MM interfaces with different clocks [39]. Note that even if we operated the DDR4 DIMM with 1000 MHz instead of 1066.66 MHz, the resulting 250 MHz user interface clock would still have a different phase than the one provided by MCDMA. It may be possible to operate both MCDMA's and EMIF's PLLs with the same reference clocks to achieve phase-synchronous clocks on both sides in order to eliminate the clock-crossing bridge, which would potentially save us some latency. We have not verified though whether this is achievable with the development kit board's wiring. Such a design would likely require a custom PCB.

3.2.1.8 Core Logic

In the previous subsections we have discussed the various third-party IP blocks that we use. We will now present our own core logic. All digital logic for our core was written in the Verilog hardware description language. As previously noted in Section 3.2.1.2, we use the user clock signal provided by MCDMA to drive our core logic with 250 MHz.

One important aspect is the memory management unit (MMU) we implemented that can translate addresses from 1 TiB virtual address spaces to addresses in a 256 GiB physical address space with a page size of 16 GiB using single-level page tables. This implements the memory management architecture we proposed in Section 3.1.6. A total of 512 virtual address spaces may be handled concurrently, i.e., one virtual address space corresponding to each virtual function and one for the physical function. Each of these parameters is customizable to allow future extensions, e.g., for larger physical memory or other page sizes. Our MMU uses M20ks to store the page tables. While Quartus's compiler can automatically instantiate MLABs and M20ks from Verilog register descriptions, it cannot infer the configuration for complex use cases. Therefore, we created an explicit instance via the *RAM: 4-PORT Intel FPGA IP* [45]. As the name implies, the IP configures the M20ks in *quad-port* mode, i.e., they provide two write and two read interface ports. We configured the write inputs as well as the read address signals to be fully registered (this is strictly required in Stratix 10 FPGAs), whereas we left the output signals unregistered.

A single page table entry (PTE) consists of two values: the physical frame number (4 bit in the configuration described above) as well as a `valid` bit that denotes whether the virtual page is mapped to a physical frame. As we have $2^6 = 64$ virtual pages per address space and 5 bit per PTE, we need $64 \times 5 = 320$ bit of memory for a single page table, and in turn, $320 \times 512 = 163840$ bit in total for all 512 page tables. The page tables are stored consecutively in the M20ks, starting with the page table for address space 0. Therefore, we can can calculate a PTE's address as

$$pte\_address = address\_space \times 2^6 + page\_number.$$

The MMU can handle two address translations in parallel. To this end, it provides two sets of input interfaces, `virt_in1` and `virt_in2`. Both interfaces have three signals each: `address`, `address_space`, and `valid`. `address` transmits the virtual address that is to be translated, `address_space` denotes the number of the virtual address space in which the translation shall be performed, and `valid` should be asserted high to request a translation. `phys_out1` and `phys_out2` are the output interfaces corresponding to the two inputs. Again, they have three signals: `address`, which contains the translation result, i.e., the physical address, `valid`, which is asserted high as soon as the translation process is completed, and `error`, which denotes whether the provided virtual address is invalid. Finally, the `program` interface is provided to configure PTEs one at a time with four signals: `virt` provides the virtual page number that is to be configured, `entry` the PTE, `address_space` the address space's number, and `valid` that is to be asserted high when the other signals are ready and the new PTE shall be programmed.

Programming is implemented in a very simple manner: the `entry` signal is directly passed through to the memory and the address is calculated according to the formula above. For address translation, the MMU extracts the page number from the input address and calculates the PTE's address according to the same formula using asynchronous logic for both inputs and directly wires the results to the memory's read address signals (hence we use quad-port SRAMs). We then calculate the physical addresses from the retrieved PTEs and the page offsets extracted from the virtual addresses, again via asynchronous logic. The `valid` and `error` signals are then set accordingly.

The overall design with single-level page tables, local M20k SRAMs, partially asynchronous logic, and unregistered SRAM output ports allows us to perform address translations within a single clock cycle, i.e., in 4 ns at a clock rate of 250 MHz. This is one place where the strengths of a custom design, as the one we created here, show: our specialized memory management architecture allows address translations to be performed much faster than in a typical

x86 CPU. It is, however, noteworthy that our single-cycle MMU design is currently also the most limiting factor for the achievable clock speed in our implementation. Quartus's *Timing Analyzer* actually determines a maximum guaranteed clock rate of 224 MHz for our design under worst-case operating conditions regarding chip quality and temperature (i.e., 100 °C). We have not observed any issues in practice, though.

Several optimizations are possible in order to achieve higher clock rates without compromising on the MMU's performance. First, our current layout regarding the placement of our logic in the FPGA fabric is not ideal: according to the timing analysis, the clock signal incurs a delay of up to 1.186 ns before it arrives at our core logic via the clock tree. This puts a strain on the setup times and may be improved through optimized placement. Second, instead of using quad-port SRAMs, we may switch to using two dual-port memories instead. Intel's specification rates dual-port M20ks on the Stratix 10 at higher speeds than their quad-port counterparts [45]. However, this would come at the cost of increased resource consumption: we would need the double amount of M20ks. Our overall design currently uses 2051 out of the 11721 M20ks available in the `1SD280PT2F55E1VG`. Although there are plenty of M20ks left, we should not use them excessively as this would again increase the clock delay to M20ks that are far away on the die.

When our core retrieves a read or write request from MCDMA via the `D2H` or `H2D` interfaces, it takes the lower bits from the address provided by MCDMA (those effectively stem from the address provided by software in the command descriptor) and uses those as the virtual address to forward to the MMU. The upper bits that represent the virtual function's ID that emitted the request are extracted to serve as the address space number. Every request is first moved into a local ring buffer. Two buffers with enough space for 64 requests are implemented, one for read requests, the other one for write requests. When there are pending requests in one of the buffers at the start of a clock cycle, we forward them to EMIF via the corresponding Avalon-MM interface. As Avalon-MM only allows to send either a read or a write request in a single clock cycle, we have chosen to prefer read over write requests simply for the reason that a read request can be entirely transferred in a single clock cycle. However, further testing in the future is required whether more sophisticated strategies would be preferable to improve fairness. If the request was queued in the previous clock cycle, we directly forward the address from the MMU. Otherwise, the MMU's output addresses are saved in the ring buffers at their corresponding request's positions. As EMIF's data bus has a width of 64 B, we can transfer exactly 64 B in a clock cycle. At a clock rate of 250 MHz, this results in a maximum theoretical throughput of $\approx$14.9 GiB s^{-1}.

Our current implementation does not actually use the `valid` signals provided by the MMU as part of the `phys_out{1,2}` interfaces, but rather relies on the MMU to complete each translation within a single clock cycle as described above. In case the MMU signals that a translation has failed via the `error` signal, we currently simply drop the request. This may currently cause wrong behavior on the `D2H` interface as, in the case of read requests, MCDMA waits for a response that never arrives if we dropped the operation. While this opens up the possibility for denial-of-service (DoS) attacks from malicious software programs by deliberately providing invalid addresses, we do not consider this a fundamental design flaw as potential implementation fixes for this issue are conceivable. For example, we may send MCDMA empty data as a response to satisfy the request. Using a custom DMA engine instead of MCDMA may further alleviate the need to perform an actual DMA operation for every request. We therefore leave a fix as well as proper error handling with notifications to software, e.g., using interrupts, as future work.

To enable our kernel driver to send PTE configuration requests to the MMU, we employ MCDMA's `PIO` interface. As described in Section 3.2.1.2, the `PIO` interface provides an Avalon-MM bus that forwards read and write operations to PCIe `BAR2`. We use the provided address to identify the address space in which the PTE shall be written. The data provided in the `BAR2` access contains the PTE as well as the virtual page number that is to be configured. As MCDMA exposes the same `BAR2` setup both on the physical function and on every virtual function, we drop `PIO` writes that were performed on VFs. In turn, we can ensure that only the privileged host system's kernel is allowed to make alterations to the page tables.

Note that, when EMIF has asserted the `waitrequest` signal on the interface, we temporarily stop forwarding requests until `waitrequest` deasserts. Here the buffers come to play: if we were to stall MCDMA immediately when EMIF is temporarily unavailable, it would take some time for MCDMA to retrieve the data for further requests when EMIF is ready again. In this situation, we would have unnecessary idle cycles. In order to sustain the overall bandwidth, we have the buffers that are filled as soon as EMIF is not ready to accept further input. We only assert our own `waitrequest` signals towards MCDMA as soon as our buffers run full. This ensures smooth operation where we utilize as many clock cycles as possible to make progress with minimal latency.

3.2.2 Hardware: DDR-T Variant

[redacted]

3.2.3 Kernel Driver

In the previous subsections we elaborated on the digital design that we implemented on our FPGA. The hardware by itself would, however, be useless without the corresponding software. Nonetheless, before we can make proper use of the hardware's capabilities in userspace, we need to implement the required support from the operating system's side. We therefore continue our bottom-up approach in this subsection by presenting the kernel driver we built

for our design. Our driver is implemented in the C programming language as an out-of-tree module for Linux 5.17.5.

We composed the module of three different parts: the PCIe driver, the character device driver, and the MMU driver. The PCIe driver is primarily responsible for probing both the hardware device itself, i.e., the physical function, and all virtual functions provided by a device. Apart from the bus initialization sequence, it also provides a simple allocator for virtual functions. To this end, the PCIe driver maintains a freelist in the form of a simple circular doubly linked list (using the implementation from Linux's `list.h` [54]) that contains each currently unallocated virtual function. Two methods are provided to the other components: `dcpmm_fpga_pcie_alloc_vf()`, which removes the freelist's first element – if present – and returns it to the caller, and `dcpmm_fpga_pcie_free_vf()`, which adds the passed VF structure back to the freelist.

The MMU component manages page mappings and is responsible for configuring the device's MMU accordingly. For its allocator, it uses a reference counting mechanism based on Linux's atomic `refcount_t` infrastructure [56] that keeps track of how many address spaces currently reference each physical frame. We have not implemented support for persistent allocations, however, we believe that this can easily be done atop our current implementation. Five methods are provided by the MMU component:

`mmu_get_page_table_for_pid()` takes a process identifier (PID) and returns a pointer to a page table structure. The rationale here is that we want to have a single virtual address space (the *Optane VAS*) on the FPGA side per process, however, the hardware is of course unaware of the concept of a Linux process. All it knows are virtual functions, and therefore, address spaces are merely bound to VFs on the hardware side, whereas a single process may have multiple virtual functions allocated to it as we will describe later in this subsection. Hence, this method returns the Optane VAS page table corresponding to the given PID, and allocates a new one if necessary.

`mmu_assign_page_table_to_vf()` creates the connection between process-oriented page tables and the FPGA-provided virtual functions. It takes a pointer to a VF structure as well as a page table pointer and adds the VF to a list of VFs that share this page table. It then proceeds to configure all of the page table's entries on the VF by performing the required MMIO writes to the physical function's `BAR2`.

`mmu_disable_vf()` is the counterpart to `mmu_assign_page_table_to_vf()`. It

disconnects the passed VF from the page table it is currently assigned to and overrides all PTEs on the VF to mark them invalid. If the page table has no more associated VFs left, the table is deallocated. The reference counters for all mapped pages are decreased accordingly.

`mmu_alloc_page()` looks for a free physical frame (i.e., one whose reference count is zero), increments its reference count, and creates a mapping for that frame in the passed page table structure. The mapping is then communicated to the device by configuring the PTE on all VFs assigned to the page table. It returns the mapping's base address within the Optane VAS.

`mmu_free_page()` performs the opposite of `mmu_alloc_page()`: it removes the mapping for the given address in the passed page table. The respective physical frame's reference count is decreased and the PTE is invalidated on all VFs the page table belongs to.

Although not currently realized, the MMU component's implementation with full reference counting for physical frames can easily be extended to support shared memory in the future. Ideally we would create an implementation that integrates with the *System V Shared Memory* interface that is a part of the POSIX standard and is therefore commonly implemented in UNIX-like operating systems [77]. This is, however, most likely not possible without modifying the Linux kernel itself.

It is further noteworthy that we have not yet implemented a mechanism to scrub pages after they were freed. While this may pose a security issue, it is a rather trivial feature to add in the future that should not influence the runtime performance that we evaluate in Chapter 4.

The character device component is where the interaction with userspace happens: as the name implies, it provides a character device that application processes can use to interact with the hardware. The device is exposed – as it is commonly done in Linux – as a device file in the device file system (`devfs`), i.e., via the path `/dev/dcpmm_fpgaN`, where `N` is the device's number in the unlikely case that multiple devices are present in the system. Device files are created with mode `0666` to effectively allow all users to use the device. The following operations are implemented on the character device:

`open()` allocates one of the device's virtual functions to the process via `dcpmm_fpga_pcie_alloc_vf()` and associates it with the newly created file descriptor. The VF is also associated with the calling process's Optane page

table by calling `mmu_assign_page_table_to_vf()`. If no free VF is available on the device, `ENOMEM` is returned.

`ioctl(MAP_FPGA_PAGE)` allocates a page in the process's Optane VAS via `mmu_alloc_page()` and returns a pointer to the page's start address to userspace. `ENOMEM` is returned in case the MMU component fails to find a free physical frame on the device.

`ioctl(UNMAP_FPGA_PAGE)` frees a page from the process's Optane VAS by calling `mmu_free_page()`. `EINVAL` is returned if the passed address is not currently mapped.

`ioctl(DMA_PAGE)` takes a user-provided 4 KiB page in the CPU-side virtual address space and prepares it to be used in DMA operations conducted by the virtual function. This allows the calling process to use the page for copy operations to or from our hardware.

To implement its functionality, the `ioctl()` first calls Linux's `pin_user_pages()` [55] which prevents the kernel from swapping the page away into mass storage. Then, we call `dma_map_page_attrs()` [53] that effectively creates a mapping in the I/O virtual address space that belongs to the virtual function and configures the system's IOMMU accordingly. `EFAULT` is returned when the passed page is not mapped in the CPU-side VAS, and `ENOMEM` if the IOMMU mapping could not be established.

`ioctl(MMAP_SELECT)` controls `mmap()`'s behavior on the file descriptor. Two modes are available: `MMAP_SELECT_BAR` and `MMAP_SELECT_DMA_MEMORY`. Their purposes are described in the next paragraph about our `mmap()` implementation. If an invalid mode identifier is passed, the call returns `EINVAL`.

`mmap()` does two different things depending on the mode previously selected using `ioctl(MMAP_SELECT)`.

In the `MMAP_SELECT_BAR` mode, we remap the virtual function's `BAR0` to the calling process's virtual address space. As described in Section 3.2.1.2, `BAR0` contains the MCDMA IP's `QCSR` and `GCSR` registers that allow to configure the command submission queues. By mapping `BAR0` into userspace, we allow processes to take full control over their command buffers without needing to involve the kernel. We return `EINVAL` in case the requested mapping size is too small to fit `BAR0`, and further pass through any errors that occurred while trying to establish the mapping.

Through the `MMAP_SELECT_DMA_MEMORY` mode, the user process may ask to allocate a new contiguous memory area that can be used for DMA from the virtual function. We employ Linux's `dma_alloc_coherent()` [53] here to create the memory area with guaranteed cache coherency. Returning the allocation's start address in the I/O virtual address space is, however, not trivially possible as the `mmap()` system call merely returns a pointer in the process virtual address space. We therefore temporarily store the address in the I/O VAS internally and allow the process to retrieve it later using the `ioctl(LAST_DMA_ADDRESS)` call described in the next paragraph. `ENOMEM` is returned in case there is not enough free space in the physical memory or in the I/O VAS to create the allocation. Mapping errors are again passed through.

`ioctl(LAST_DMA_ADDRESS)` is meant to be used by user processes in conjunction with `mmap()` in `MMAP_SELECT_DMA_MEMORY` mode. It returns the last `mmap()` allocation's address in the virtual function's I/O virtual address space.

Note that there is no implementation of `read()` or `write()` or other methods that would allow direct access to data via the character device. It is further important that the entire security guarantees of our implementation depend on the presence of a capable IOMMU. If the system does not have an IOMMU, processes may pass arbitrary addresses in the CPU's physical address space to our hardware, which would thereby allow full read and write access to the entire system memory.

A downside to the current implementation is that userspace processes need to explicitly acquire DMA-capable (i.e., I/O VAS) addresses for each page they want to use as source or target for copy operations with our hardware. The entailing overhead for system calls may reduce the achievable performance. A simple, yet elegant solution emerges with future hardware. Version 3.0 of Intel's *Virtualization Technology for Directed I/O* (VT-d) [46, 65] introduces a novel feature named *Scalable I/O Virtualization* (Scalable IOV) [28]. The *Sapphire Rapids* CPU generation is going to be the first to support VT-d 3.0 including Scalable IOV [5]. Among the new capabilities offered by Scalable IOV is the support for *Shared Virtual Memory* (SVM). Via SVM, it becomes possible to share a process's virtual address space with I/O hardware, i.e., they can use the very same addresses to access system memory. To this end, PCI Express TLPs are prefixed with a *Process Address Space ID* (PASID) [62, Section 6.20]. The operating system may configure PASIDs in VT-d 3.0-capable IOMMUs to be associated with the virtual address spaces (i.e., the page tables) of specific processes. The IOMMU then performs address translations accordingly.

With such a design, it is however not guaranteed that the operating system

does not swap out pages to mass memory. While pinning all pages of a process would be a technically correct solution, the incurred downsides such as potentially reduced overall system performance through increased thrashing of other processes make this approach seem infeasible. Further, this would not work anymore if the allocated memory of all processes that use our accelerator is larger than the available space. Thankfully, modern PCI Express versions have introduced the *Page Request Interface* (PRI) as a part of the *Address Translation Services* (ATS) [62, Chapter 10]. ATS and PRI allow a PCI Express device to request the operating system to make a page available in the system memory. In turn, it is possible to leverage SVM without needing to pin the pages of userspace processes. Overall, a SVM-based implementation would allow us to let processes use their entire virtual address space for copy operations without any system call overhead, without needing to translate between CPU virtual addresses and I/O virtual addresses, and without pinning pages.

Although not implemented in this thesis, we would like to sketch how our design could easily be extended to be used efficiently in system virtualization scenarios. The most important buildings blocks are already there: SR-IOV and the ability to directly hand control over the virtual functions to userspace processes without breaching security. We may similarly hand out virtual functions to guest operating systems in virtual machines. This could be implemented via corresponding hypercalls in the drivers within the hypervisor and the guest OS. The guest driver may then, in turn, manage how virtual functions are given out to processes.

3.2.4 Userspace Library

We have now iterated over our hardware design as well as our kernel driver. This subsection addresses the last puzzle piece: our userspace library that abstracts away the complexity of interacting with the hardware and the kernel. In the end, programs are provided with a simple asynchronous `memcpy()`-style interface. Like our kernel driver, the userspace library is implemented in the C programming language.

The library is designed to provide one pair of read/write command buffers per thread. To this end, each thread is supposed to call `dcpmm_fpga_init()` once before it may use the provided functionality. The initialization method first tries to `open()` the device file at `/dev/dcpmm_fpga0` and thereby allocates a virtual function that is then associated with the calling thread. Using the `MMAP_SELECT_BAR` mode that we described in the previous subsection, we then map the VF's `BAR0` into the process's virtual memory. Finally, we proceed to setup the two ring buffers for read and write commands and configure the `QCSR` in `BAR0` accordingly (as described in Section 3.2.1.2). To achieve a

contiguous mapping, we use the `MMAP_SELECT_DMA_MEMORY` mode of our kernel driver's `mmap()` implementation for the queues. Our implementation currently hardcodes a queue size of 1 MiB. With a command descriptor size of 32 B, this provides enough space for 32768 descriptors. As MCDMA requires every 4 KiB page to have one *link descriptor*, 256 descriptors are wasted, and in turn, a total of 32512 commands may be stored. We deem this enough for practical applications.

After initialization has completed, the calling thread may map and access pages in the Optane virtual address space. To this end, we provide the `mmap_dcpmm()` and `munmap_dcpmm()` calls that are essentially wrappers around the `MAP_FPGA_PAGE` and `UNMAP_FPGA_PAGE` `ioctl()`s provided by the kernel driver. Unlike the traditional POSIX `mmap()` [77] that allows mapping arbitrary amounts of pages, our `mmap_dcpmm()` unconditionally retrieves a single 16 GiB page. Note that, as our kernel driver shares the device-side page tables between virtual functions of a single process, threads may of course also access pages that were mapped by other threads within the same process.

To access the device-side memory using *asynchronous copies*, our library exposes the `memcpy_from_dcpmm()` and `memcpy_to_dcpmm()` methods. Their signature is the same as C's `memcpy()`, however, the `src` argument in the `from` variant and the `dst` argument in the `to` variant must be addresses from the Optane VAS, respectively. The other arguments must be addresses in the system virtual address space. Note that our hardware does not currently support copies within the device-side memory. Our `memcpy_{from,to}_dcpmm()` implementations first translate all addresses in the process's CPU-side VAS to addresses in the virtual function's I/O VAS via the driver's `ioctl(DMA_PAGE)` system call. However, as system calls cause a non-trivial overhead, it would be vastly detrimental for the overall performance of our implementation if every copy operation required kernel assistance. For this reason, we cache the translations in userspace via a hash table using the *uthash* library [16]. To perform a translation, we first lookup the page in the hash table, and only if no entry is found we execute the system call to have the kernel driver establish a mapping in the I/O VAS. New mappings are, of course, added to the hash table.

After the I/O VAS address is known, the rest is easy: we fill the next free command descriptor in the respective ring buffer with the command data, i.e., source and destination addresses and copy size, and update the tail pointer in the `QCSR` to notify our hardware. We then return immediately without waiting for the command to complete, hence the operation is completed *asynchronously*. Note that there is currently a 4 KiB limit for the data size that may be copied in a single operation. This limit is not incurred by our hardware, but rather by a driver limitation: to copy larger areas, we would need to ensure that

consecutive pages in the CPU-side VAS are also contiguously mapped in the I/O VAS. Our kernel driver does not currently offer such functionality, therefore copy operations may not cross 4 KiB page boundaries. Again, this is another issue that can very trivially be solved with an SVM-based implementation as outlined in Section 3.2.3.

As mentioned in Section 3.2.1.2, we use MCDMA's writeback feature to determine whether the ring buffer that we want to submit an operation to is currently full. If that is the case, we busy-wait until there is space in the buffer again. This should not be harmful for performance in practice as long as the buffers are large enough.

Based on what we sketched in Section 3.1.1, we further offer simple pseudo-synchronous variants named `memcpy_{from,to}_dcpmm_wait()`. These methods employ busy waiting to emulate synchronous behavior, i.e., they loop until the hardware has signaled completion for the submitted command. More sophisticated implementations of pseudo-synchronous or truly synchronous operations according to the options we described in Section 3.1.1 remain as future work.

Chapter 4

Evaluation

We have described our implementation in the previous chapter in full detail. Now, we want to evaluate the success of our approach. We begin by introducing the tool we have written to conduct benchmarks and a description of the system we used for testing. Then, we present and discuss the results obtained with our DDR4 and DDR-T designs, respectively.

4.1 Benchmark Tool

Our benchmark tool is, just like our kernel driver and our userspace library, implemented in C. It aims to allow to compare several performance metrics of our design versus having the memory attached to the CPU directly. To this end, we implemented the individual benchmarks separately from several different backends. The backends implement the actual memory accesses and offer a backend-agnostic interface so that the very same benchmark code may be executed on each backend without modifications. We have implemented three different backends: `system`, `dax`, and `fpga`. As their names imply, they access system memory (i.e., DDR4) directly, via Linux's DAX interface for Optane (as we described in Section 2.1), and via our FPGA-based implementation, respectively. They provide the following set of methods to the benchmarks: `mmap()`, `munmap()`, `memcpy_from()`, `memcpy_to()`, `memcpy_from_wait()`, and `memcpy_to_wait()`. In the case of the `fpga` backend, they are implemented as simple wrappers around the respective methods from our userspace library as described in Section 3.2.4. When using the `fpga` backend, we ensure that all CPU-side pages that are used in copy operations have corresponding mappings in the I/O VAS before starting a benchmark run. Thereby, we avoid potential runtime overhead for system calls as described in Section 3.2.4. The `system` and `dax` backends have the calls described above implemented via the corresponding

methods from the C standard library and the POSIX interface. `_wait()` variants are realized using atomic memory fences and the x86 `clflush` instruction that forces the CPU to writeback specific cachelines [29]. `dax` differs subtly as we use AVX2 256 bit instructions for copying data from DRAM to PMem instead of the C library's `memcpy()`, i.e., `vmovdqa` for loads and `vmovntdq` for non-temporal stores. Yang et al. [82] have found that Intel CPUs achieve higher bandwidths when using non-temporal stores in PMem setups.

We implemented a total of five benchmarks: two that measure read and write-flush-read latencies, two that measure read and write bandwidths using sequential accesses, and another one that gauges the time it takes to copy a specific amount of data. Note that we do not benchmark the write latency on its own as our hardware does not currently signal the point where the write operation is completed on the attached memory, but rather only when the data has been received by the MCDMA IP. We therefore measure the combined latency for writing data and then reading it again. Together with the read latency benchmark we can then generate an estimate of the actual write latency. The latency benchmarks are entirely single-threaded and work by measuring the time between the submission of the first command or instruction and the time the data becomes ready 100 times. Then, the average time taken per operation over a total of 1000 iterations is calculated. The bandwidth benchmarks run for precisely 16 s and submit as many read or write commands of a specific size as possible during that time on one or multiple threads. After the 16 s have passed we submit one last *synchronous* command that completes after all the other commands are done. We then measure the total time taken as well as the amount of transfers that were completed and thereby calculate the bandwidth that each thread has achieved. We use Linux's `CLOCK_MONOTONIC` clock for all timing measurements [8].

As we show in the next section, we ran our benchmarks on a dual-socket system with two CPUs. To avoid the possible influence the NUMA layout has on our numbers, we made our benchmark tool NUMA-aware via the `libnuma` library [75]. More specifically, we configure the NUMA policies so that all threads are forced to run on a single, user-defined NUMA node and all memory is allocated on the local node. Further, to alleviate scheduling effects, we employ Linux's `SCHED_RR` policy with maximum scheduling priority for all our threads [72].

4.2 System Setup

We ran all benchmarks on the very same base system equipped with two Intel Xeon Silver 4215 CPUs installed on a Supermicro `X11DPi-N` mainboard.

The Xeon Silver 4215 is an SMT-capable octa-core 64 bit x86 processor from Intel's *Cascade Lake* generation [47]. We disabled SMT to avoid influences from differing core resource utilization due to scheduling variations. Both CPUs each have 16 GiB of single-channel DDR4 memory with a speed of 2133 MT s^{-1} and timings from the 2933 MT s^{-1} speed bin attached to them. This matches the configuration of the DIMM we used for our DDR4 hardware variant as shown in Section 3.2.1.3. Only one of the CPUs has a 128 GiB Optane DIMM from the PMem 100 generation connected to it. Our accelerator is attached via a PCIe 3.0 x16 link.

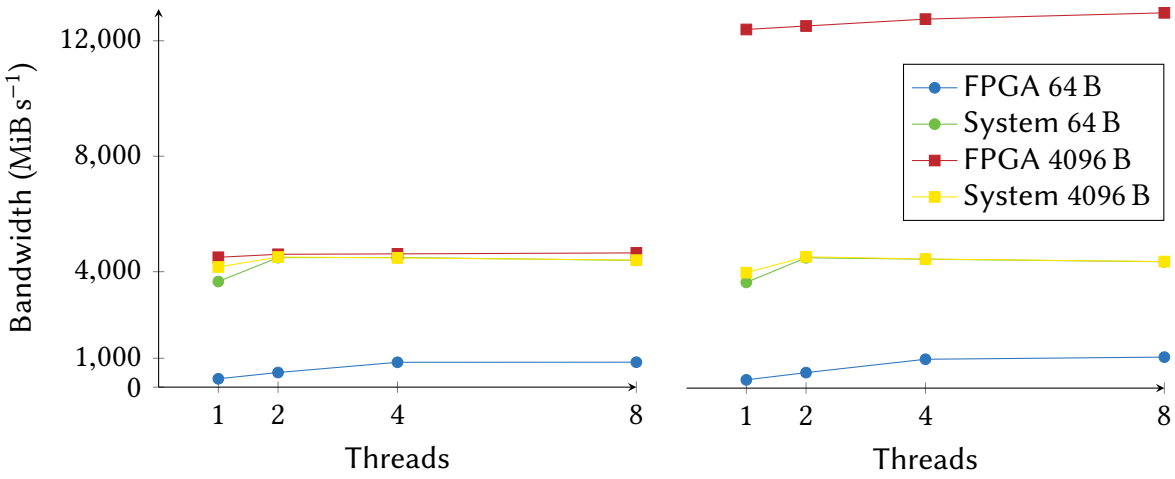
On the software side, we used Ubuntu 20.04 with a custom-built Linux 5.17.5 kernel. The kernel and all of our own code were compiled with gcc 9.4.0 using the default `-O2` optimization level.

4.3 Results: DDR4

In this section, we want to compare the performance of our design when used with DDR4 memory versus when accessing DDR4 system memory directly from the CPU via traditional load/store operations.

We begin by taking a look at the achievable read and write bandwidths. We ran all benchmarks with two different access sizes: 64 B and 4096 B. Here, *access size* denotes the size of the data transferred in a single copy operation. The sizes were not chosen arbitrarily: 64 B is the size of a single cacheline in the CPU and further equals the word length of DDR memories. 4096 B, i.e., 4 KiB, on the other side is the maximum amount of data we can transfer to the FPGA in a single operation as described in Section 3.2.4. Further, we executed all tests with varying parallelism between one and eight threads running concurrently.

Figure 4.1 shows the results. First of all, it should be noted that the read and write benchmarks are effectively the same in the system case: here, we simply copy from one place in the system memory to another. Hence, the numbers are nearly the same. We further find that the access size does not really matter when using system memory, however, small accesses gravely affect the achievable bandwidth when reading and writing to the memory attached to our hardware: with 64 B accesses, we achieve a read performance of up to 864 MiB s^{-1} with eight threads, whereas the bandwidth tops out at 4651 MiB s^{-1} when 4 KiB copies are used. This hints that further optimization of our design is needed to improve performance for small accesses. Further, increased parallelism is generally favorable with our design – single-threaded 64 B reads merely allow for a bandwidth of 290 MiB s^{-1}. The opposite is true for accesses to system memory: although two threads reach a higher copy performance than a single one (4508 MiB s^{-1} versus 4163 MiB s^{-1}), the total bandwidth drops again with



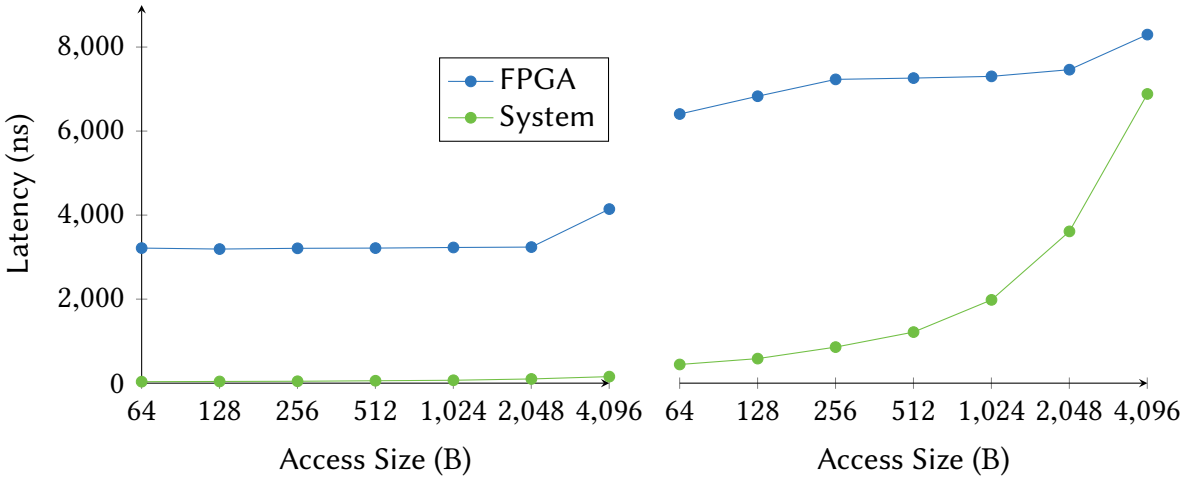
(a) Read bandwidth

(b) Write bandwidth

Figure 4.1: DDR4 bandwidth using our design (*FPGA*) versus system memory directly accessed from the CPU (*System*) with different access sizes, namely 64 B and 4096 B.

more threads down to 4400 MiB s^{-1}. The most glaring result is, however, the fact that our design achieves much lower bandwidth with read operations (up to 4651 MiB s^{-1}) than it does for writes (up to 13 002 MiB s^{-1}). We are not sure about the reasons that cause this deviation. A possible explanation could be that the DMA engine (i.e., Intel MCDMA that we introduced in Section 3.2.1.2) does not fetch commands fast enough to cope with the device-side latency of read operations, thereby leaving time gaps where no commands are processed. We have yet to verify this hypothesis, though. Improving read bandwidth is therefore left as future work.

Next, we compare the access latencies. Here, we only used a single thread to avoid variations induced by parallel loads. Instead, we took measurements for different access sizes. The results are graphed in Figure 4.2. Naturally, latencies are higher with our design than with direct accesses from the CPU. A 64 B read takes about 34 ns with the CPU and 3215 ns with our accelerator, a slowdown by about 100×. We can further observe a plateau regarding the latency with the FPGA up until 2048 B accesses. Latency only starts to rise with 4 KiB accesses. This effect can likely be attributed to the time overhead induced by the PCIe bus. The processing overhead from our hardware's side only starts to carry weight for large transfers. We can further estimate the write latency by subtracting the read latency from the write-flush-read latency. For 512 B accesses, we can thereby deduce an estimated write latency of 7261 ns − 3214 ns = 4047 ns for our hardware. Note that this value still includes a certain time overhead induced



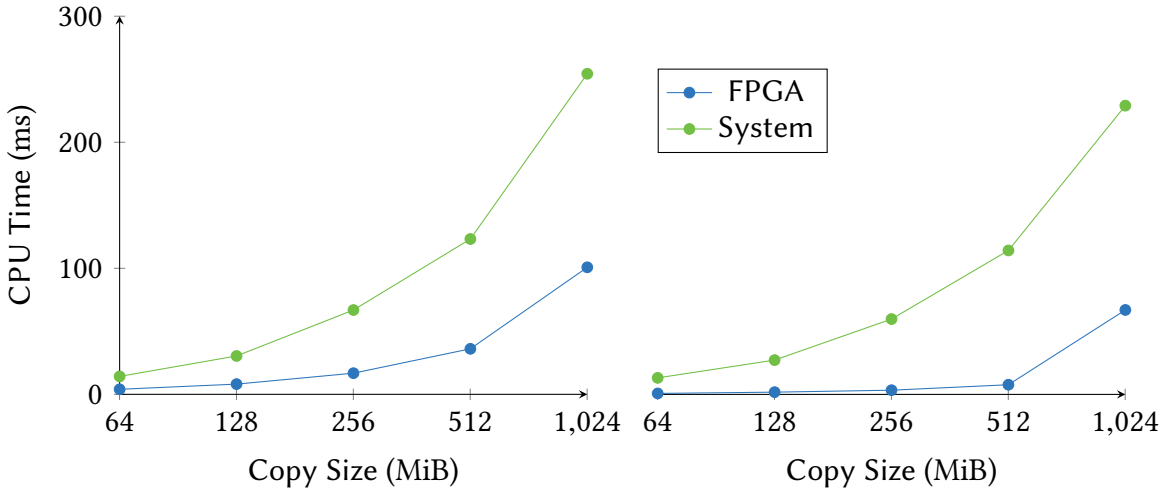
(a) Read latency

(b) Write-flush-read latency

Figure 4.2: DDR4 access latency using our design (*FPGA*) versus system memory directly referenced from the CPU (*System*) with different access sizes ranging from 64 B to 4096 B on a single thread.

by the fact that we wait until completion for the write command is signaled before we submit the following read command.

The idea behind the approach in this thesis is to reduce CPU overhead for copy operations. To measure the success of this goal, we compare the CPU time it takes to perform copy operations using consecutive 4 KiB copies. Figure 4.3 shows the results for different access sizes with one and eight parallel threads. In the parallel scenarios, the total workload is split evenly between the threads. We can see from the plots that the overall CPU time taken is always lower with our accelerator, thereby proving that our approach is capable of saving CPU time even with fast DDR4 memory. For example, in the single-threaded 64 MiB case, we achieve a 72 % reduction in the CPU time (4 ms versus 14.22 ms). The largest difference can be observed in the scenario with eight threads and a transfer size of 512 MiB: here, our design requires 93 % less CPU time (7.666 ms versus 114.222 ms). The plots show a superlinear uptick for our FPGA-based design between 512 MiB and 1024 MiB. The explanation is simple: as discussed in Section 3.2.4, our command buffers currently have enough space for 32512 command descriptors. With 4 KiB copies, we can therefore buffer copy commands for slightly less than 128 MiB. As our implementation busy-waits if the buffer is full until space becomes free again, a larger fraction of time is spent waiting when the transfer size becomes too large within a certain timeframe.



(a) One thread

(b) Eight threads

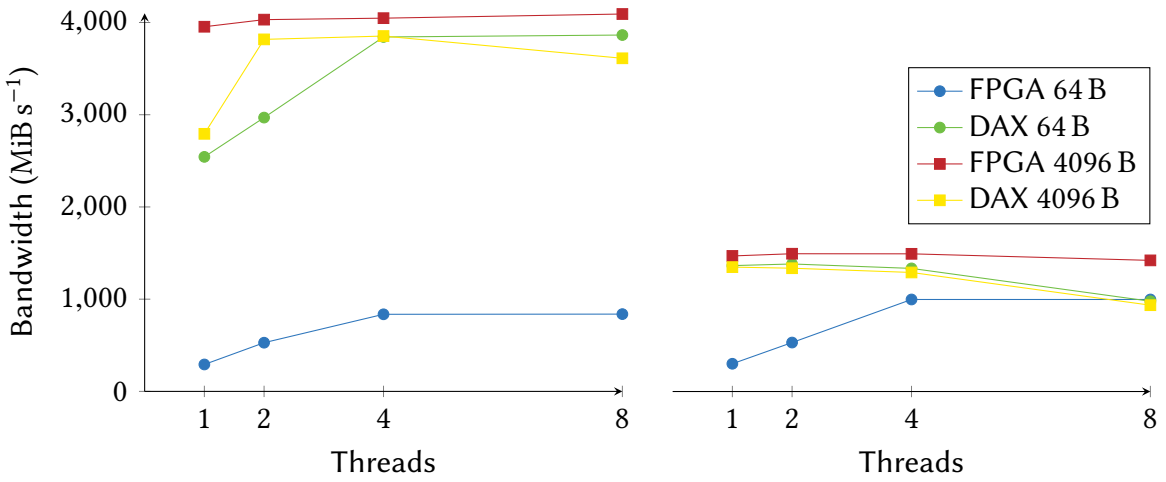
Figure 4.3: Visualization of the CPU time it takes to copy different amounts of data using synchronous accesses to system memory (*System*) versus using asynchronously submitted commands to our accelerator (*FPGA*).

4.4 Results: DDR-T

In the previous section, we evaluated the performance metrics of our design when used with DDR4 memory. Now, we want to do the same with the memory technology that this thesis is centered around: Optane PMem.

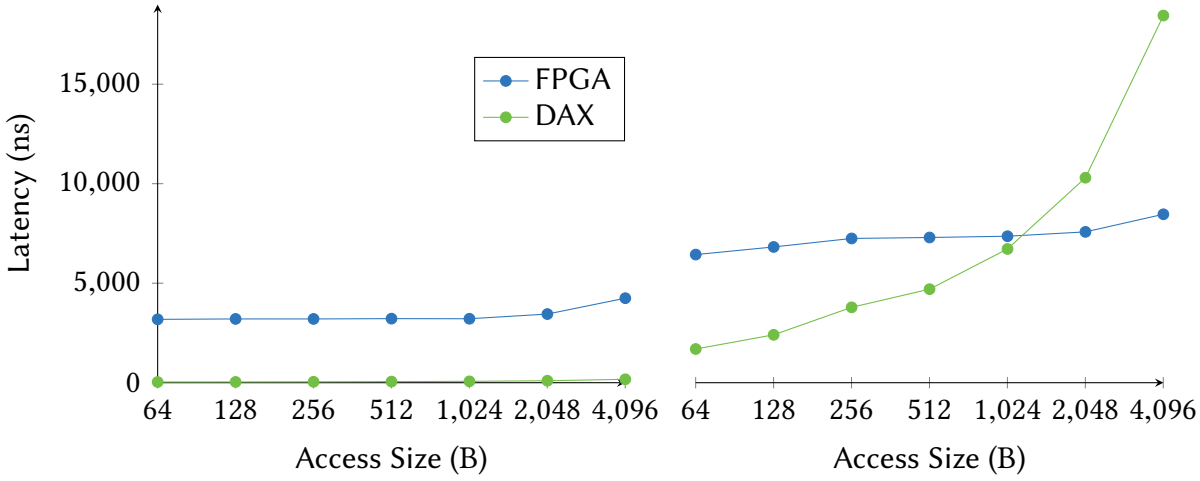
Again, we begin with the bandwidth results shown in Figure 4.4. Similar to the DDR4-based results, we see that small copy operations using 64 B accesses do not perform well. However, with 4096 B accesses, our implementation shines: the plots show that we reach higher bandwidths than the CPU in all cases and are less sensitive to parallelism. Where the CPU only achieves a write bandwidth of 936 MiB s^{-1} with eight parallel threads, our accelerator still reaches 1422 MiB s^{-1}, an improvement by about 52 %. Similarly, regarding the read bandwidth, the CPU fails to saturate the memory's theoretically achievable bandwidth with a single thread and achieves a mere 3612 MiB s^{-1}, where our accelerator allows for 4092 MiB s^{-1}.

Regarding latency, the numbers shown in Figure 4.5 are roughly the same for our accelerator. Again, this underlines that the latency is primarily dominated by the overhead induced by the data transfer via the PCI Express bus. Similarly, there is no significant deviation in the read latencies compared to what we previously saw with DDR4. However, the write-flush-read latencies show a surprising result: for large accesses with $\geq$ 2048 B, our implementation



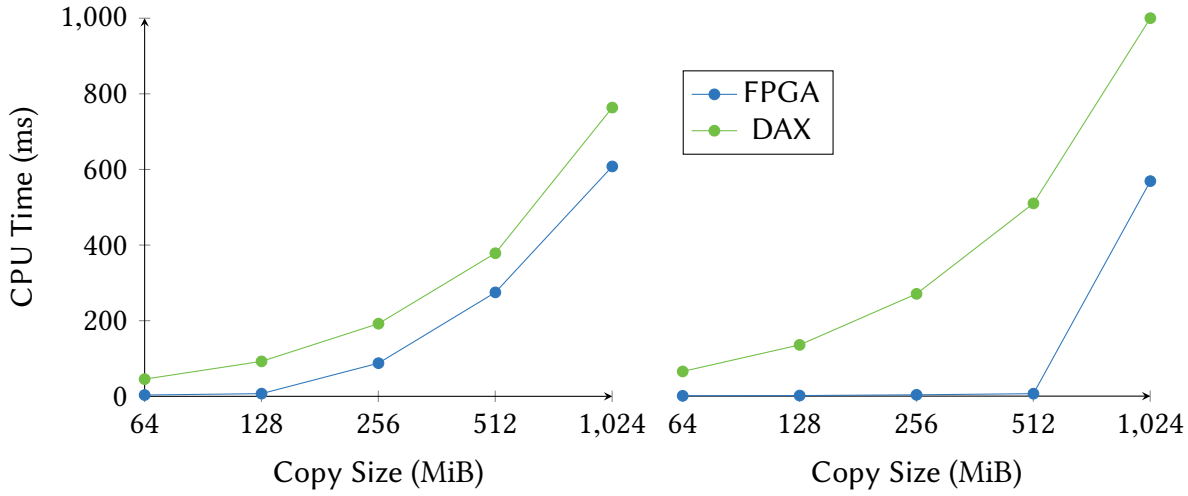
(a) Read bandwidth (b) Write bandwidth

Figure 4.4: Optane bandwidth using our design (*FPGA*) versus when directly accessed from the CPU (*DAX*) with different access sizes, namely 64 B and 4096 B.



(a) Read latency (b) Write-flush-read latency

Figure 4.5: Optane access latency using our design (*FPGA*) versus when directly referenced from the CPU (*DAX*) with different access sizes ranging from 64 B to 4096 B on a single thread.



(a) One thread

(b) Eight threads

Figure 4.6: Visualization of the CPU time it takes to copy different amounts of data using synchronous accesses to Optane (*DAX*) versus using asynchronously submitted commands to our accelerator (*FPGA*).

is actually faster than when the same sequence of accesses is performed synchronously using the CPU. With 4096 B copies, the CPU takes about 18 449 ns on average, whereas our implementation gets the job done in a mere 8463 ns. This equals a latency reduction by about 54 %.

Finally, we evaluate the CPU time used for copy operations in Figure 4.6. In single-threaded scenarios, our accelerator's curve looks similar to the CPU's, but stays up to 155 ms lower. The same effect that we discussed in the previous section with DDR4 plays a role here: as soon as the command buffers run full, our implementation waits until there is space again. In turn, the CPU time is offset roughly by the time it takes to actually copy 128 MiB. The largest relative reduction can be observed with 512 MiB copies with eight parallel threads: using synchronous accesses, the CPU spends about 510 ms. However, with our design, only about 6 ms are needed, resulting in a 98.8 % improvement. This proves that our implementation is able to achieve the primary goal of asynchronous copy offloading for PMem: reduced CPU overhead. Similar to what we saw with DDR4, there is a sharp uptick with 1024 MiB copies and eight parallel threads. Again, this is explained by the command buffers running full.

4.5 Discussion

The aim of this thesis was to implement asynchronous copy offloading for Intel Optane Persistent Memory based on the goal that we can significantly reduce the CPU time spent for memory accesses, and in turn, increase the overall system performance. Given the results in the previous sections, we believe that we can safely consider our approach a success in this regard. We were able to show that our implementation takes less CPU time compared to synchronous copies in all evaluated scenarios with a reduction of up to 98.8 %. Although we have not conducted benchmarks with other processes running in parallel, we consider it safe to assume that the overall system performance in such a scenario is also improved as the CPU time that we save is now available to be used by other programs.

In Section 3.1, we defined a set of three goals that we consider important for the practical viability of our approach: *bandwidth saturation*, *low latency*, and *quick submission*.

Regarding bandwidth, Figure 4.4 has clearly shown that we are able to saturate the bandwidth of a single PMem DIMM and even outperform the CPU's memory controller as long as the access granularity is large enough. DDR4 benchmarks, however, have shown that the read bandwidth that our design achieves is not quite on par yet for memories that are faster than PMem. We consider this a partial success nonetheless as the bandwidth is enough so far for Optane. However, as we plan to implement a dual-channel design in the future, it likely becomes necessary to work towards better read performance. As previously described in Section 4.3, we suspect that the DMA engine we employed in our hardware design is at fault here. We have already argued that Intel MCDMA is not an ideal choice for our use case in previous chapters. For example, the strictly separated command buffers are not ideal – in Section 3.1.3 we established that unified buffers are preferable to maintain sequential ordering between memory operations regardless of their direction. If we are able to verify that the lacking read performance is an inherent issue with MCDMA, we have an even stronger case for building a custom DMA engine.

Another reason for a custom DMA engine stems from the results we observed for small access sizes, e.g., 64 B. Here, we are generally unable to keep pace with the CPU in terms of bandwidth. This is, however, not surprising: as described in Section 3.2.1.2, each command descriptor requires 32 B of space in the command buffer, resulting in a memory overhead of 50 % just for submitting a command. In turn, the time spent for fetching commands plays a much larger role. We therefore argue that we may improve the performance of small accesses by optimizing both the size of a command descriptor as well as the fetching mechanism by speculatively reading ahead in the command buffer.

Regarding the command descriptors, we may pack the information we need into less than 32 B. A minimal command descriptor design would include a *valid* bit, two addresses of up to 64 bit each, a command type identifier with a few bits, and the payload size (e.g., in 32 bit). Hence, it should be possible to create command descriptors that are roughly 20 B in size, and in turn, we may fetch more descriptors in a single memory access. Further, smaller descriptors likely result in an even higher reduction in terms of CPU time and we could fit more commands in a buffer without making it larger.

The latency of memory operations with our design is, unsurprisingly, generally worse than with synchronous accesses from the CPU as we have seen in the previous sections. However, we suspect there is not a lot of room for improvement given that large parts of the overhead most likely stem from the PCIe bus. We want to conduct a deeper breakdown of the components that contribute to the latency in the future to analyze possible optimizations. In this matter, we further plan to switch to the CXL bus that we briefly described in Section 2.3. Using CXL may result in considerably better latencies for a design such as ours compared to PCIe.

The successful achievement of the third of our goals, quick submission, is already proven by the low CPU time required by our implementation. However, command submission quickly causes rising CPU overhead when command buffers are full as we have seen in Figure 4.3. Apart from larger buffers or smaller command descriptors, we may consider alternative ways of dealing with full buffers. For example, we may return immediately with an error code such as `EAGAIN` to signal that command submission is currently not possible to the calling thread. Such behavior is already commonly established practice with system calls such as `read()` when used with file descriptors that are configured as non-blocking [68]. Another possibility would be to schedule other processes or threads when a command cannot be submitted due to the buffer being full in order to have the system progress in the meantime until there is more space. These options ultimately stem from the fact that we have solved the issue of PMem accesses being I/O-bound and CPU-bound at the same time as Werling et al. [78] have argued and as we have discussed in Section 2.2. Thereby, the entire range of I/O handling and scheduling mechanisms found in operating systems becomes viable again with Optane PMem thanks to asynchronous copy offloading implementations such as the accelerator that we have presented in this work.

Chapter 5

Conclusion

Novel memory cell technologies promise interesting new possibilities, but also pose new challenges for operating system designers and software engineers. This thesis revolved around Optane Persistent Memory, a recent main memory technology by Intel based upon their 3D XPoint cell technology. Unlike traditional DRAM-based main memories, Optane offers persistence for the data stored on it. However, the bandwidth depends strongly on the access patterns as well as the degree of parallelism and is not nearly on par with what can be achieved with DRAM. The low bandwidth further causes non-trivial overhead regarding CPU time, which in turn can be significantly detrimental to the overall system performance.

Asynchronous copy offloading is an approach previously proposed by Werling et al. to combat the implications of Optane's performance by moving the load of performing memory copies to external hardware. As previous implementations have been of limited success, we have aimed to design custom hardware in the form of an FPGA-based PCI Express device in this work. Our design offers asynchronous access to Optane memory without runtime overhead for system calls and lock-free parallelism. With an initial implementation using DDR4 memory, we were able to show that our accelerator generally achieves a high enough bandwidth to potentially saturate Optane. Further, the CPU time spent for memory operations is reduced by up to 93 %. In the next step, we evaluated the performance achieved with actual Optane memory. As expected, the bandwidth is high enough to saturate a single DIMM and the required CPU time is far below what is needed with synchronous accesses. However, the latency is about 100× higher than when accesses are performed synchronously.

Overall, we consider our implementation a success nonetheless. The primary goal of reducing CPU time is greatly achieved without drawbacks in terms of bandwidth. Although the possibilities seem limited, we want to explore various options to reduce latency in the future.

5.1 Future Work

We have already discussed several improvements and extensions regarding the implementation in the previous chapters, e.g., a custom DMA engine or [redacted]. In this section, we want to talk about a few ideas that may lead to new scientifically relevant insights.

5.1.1 Additional Operations

We already elaborated on what operations may be viable in addition to asynchronous `memcpy()`s in Section 3.1.1. As we described in Section 2.2, the original proposal for asynchronous copy offloading by Werling et al. [78] included having an accounting-based switching mechanism that employs synchronous accesses as long as bandwidth is not saturated to keep latencies low. We aim to provide proper synchronous read and write operations in the future with traditional load/store semantics in the future. More broadly speaking, the possibilities of mixed-synchronicity interfaces may open up new use cases for persistent memory technologies such as Optane. Currently, in the memory hierarchy, each level is either synchronous (e.g., main memory) or asynchronous (e.g., mass storage), however Optane PMem both exhibits characteristics that are known from main memory as well as from SSDs. We believe that mixed-synchronicity interfaces for memory are an underexplored area in previous research.

5.1.2 Compute Express Link

In Section 2.3, we briefly introduced the novel *Compute Express Link* (CXL) bus standard that was designed to achieve high performance and low latency with full cache coherency specifically for memory and storage devices. Due to the lack of CXL-capable hardware at the start of this work, we built our design upon PCI Express. However, CXL seems naturally well-suited for our endeavour given its protocols that are tailored specifically for memory peripherals. CXL's full cache coherency further allows for interesting enhancements, e.g., the L4 cache design that we describe in Section 5.1.4. Hence, we plan to reimplement our approach with CXL-capable FPGAs and CPUs in the future.

5.1.3 Out-of-Order Processing

As discussed in Section 3.1.7, asynchronously executed memory operations enable the possibility to process them in an out-of-order fashion as it is not

as crucial to keep latency to an absolute minimum as it is with synchronous accesses. Given that PMem's performance depends strongly on the access patterns as shown by Yang et al. [82], we may leverage our asynchronous design to reorder accesses to improve the achieved performance. We would therefore like to explore suitable algorithm designs for out-of-order processing in the future.

5.1.4 Optane L4 Cache

In Section 2.1, we described that the 3D XPoint cells in Optane memories are constructed in a 256 B block structure despite Optane PMem offering a byte-addressable interface. Consequently, small writes are prone to cause write amplification. Given that modern CPUs typically use 64 B cachelines [17], there is a high likelihood that writebacks end up as 64 B accesses. In order to reduce write amplification, we suggest leveraging cache-coherent interconnects such as CXL to create a device with an inclusive L4 cache specifically for Optane PMem with a cacheline size of 256 B. We believe that such a design has the potential to significantly evade write amplification in many typical read-modify-write scenarios.

Bibliography

[1] P. Alcorn. *AMD's Data Center Roadmap: EPYC Genoa-X, Siena Announced, Turin in 2024*. June 15, 2022. URL: https://www.tomshardware.com/news/amds-data-center-roadmap-eypc-genoa-x-siena-announced-turin-in-2024.

[2] P. Alcorn. *Intel Kills Optane Memory Business, Pays $559 Million Inventory Write-Off*. Aug. 2, 2022. URL: https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good.

[3] ARM. *AMBA AXI and ACE Protocol Specification. AXI3 , AXI4 , and AXI4-Lite ACE and ACE-Lite*. 2013. URL: https://documentation-service.arm.com/static/5f915b62f86e16515cdc3b1c.

[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC Boston, 2018.

[5] A. Biswas and S. Kottapalli. *Sapphire Rapids. Next-Gen Intel Xeon Scalable Processor*. Aug. 23, 2021. URL: https://hc33.hotchips.org/assets/program/conference/day1/HC2021.C1.4%20Intel%20Arijit.pdf.

[6] J. Choe. *Intel 3D XPoint Memory Die Removed from Intel Optane™ PCM (Phase Change Memory)*. 2017. URL: https://www.techinsights.com/blog/intel-3d-xpoint-memory-die-removed-intel-optanetm-pcm-phase-change-memory.

[7] P. Clarke. "Patent search supports view 3D XPoint based on phase-change." In: *EE Times,[online], Jul* 31 (2015).

[8] *clock_getres(2) — Linux manual page*. Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/clock_gettime.2.html.

[9] J. Coburn et al. "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories." In: *ACM SIGARCH Computer Architecture News* 39.1 (2011), pp. 105–118.

[10] Compute Express Link Consortium, Inc. *Compute Express Link (CXL) Specification Revision 3.0, Version 1.0*. Aug. 1, 2022.

[11] D. Das Sharma. *PCIe® 6.0 Specification: The Interconnect for I/O Needs of the Future*. June 4, 2020. URL: https://pcisig.com/sites/default/files/files/PCIe%206.0%20Webinar_Final_.pdf.

[12] D. Das Sharma. *The PCIe® 6.0 Specification Webinar Q&A: Supported Features in PCIe 6.0 Specification*. Mar. 17, 2021. URL: https://pcisig.com/blog/pcie%C2%AE-60-specification-webinar-qa-supported-features-pcie-60-specification.

[13] *Direct Access for files*. Apr. 27, 2022. URL: https://www.kernel.org/doc/html/v5.17/filesystems/dax.html.

[14] J. Gantz and D. Reinsel. "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east." In: *IDC iView: IDC Analyze the future* 2007.2012 (2012), pp. 1–16.

[15] S. Ghose et al. "Processing-in-memory: A workload-driven perspective." In: *IBM Journal of Research and Development* 63.6 (2019), pp. 3–1.

[16] T. D. Hanson and A. O'Dwyer. *uthash: a hash table for C structures*. 2022. URL: https://troydhanson.github.io/uthash/.

[17] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2019.

[18] A. J. Herdrich, M. D. Cornu, and K. M. Abbasi. *Introduction to Memory Bandwidth Allocation*. Mar. 12, 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html.

[19] S. Hong, O. Auciello, and D. Wouters. *Emerging non-volatile memories*. Springer, 2014.

[20] A. Ilkbahar. *The Future of Intel® Optane™ Persistent Memory*. Dec. 16, 2020. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2020/12/The-Future-of-Intel%C2%AE-Optane%E2%84%A2-Persistent-Memory-Alper-Ilkbahar-.pdf.

[21] Intel Corporation. *Accelerating High-Speed Networking with Intel® I/O Acceleration Technology. White Paper*. Feb. 22, 2012. URL: https://www.intel.com/content/dam/doc/white-paper/i-o-acceleration-technology-paper.pdf.

[22] Intel Corporation. *Avalon® Interface Specifications*. Sept. 26, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/743655?fileName=mnl_avalon_spec-683091-743655.pdf.

[23] Intel Corporation. *External Memory Interfaces Intel® Stratix® 10 FPGA IP User Guide*. Mar. 11, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/666334?fileName=ug-s10-emi-683741-666334.pdf.

[24] Intel Corporation. *Intel and Micron Produce Breakthrough Memory Technology*. July 28, 2015. URL: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[25] Intel Corporation. [redacted] May 9, 2022.

[26] Intel Corporation. *Intel Optane DC Persistent Memory Readies for Widespread Deployment*. Oct. 30, 2018. URL: https://newsroom.intel.de/news/intel-optane-dc-persistent-memory-readies-for-widespread-deployment/#gs.gyg13n.

[27] Intel Corporation. *Intel Optane Memory Now Available – Boosts Speed for Gaming, Web Browsing and More*. Apr. 25, 2017. URL: https://newsroom.intel.de/news-releases/intel-optane-memory-now-available-boosts-speed-gaming-web-browsing/#gs.gyg1dy.

[28] Intel Corporation. *Intel Scalable I/O Virtualization. Technical Specification*. Sept. 2020. URL: https://www.intel.com/content/www/us/en/develop/download/intel-scalable-io-virtualization-technical-specification.html.

[29] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Apr. 2022.

[30] Intel Corporation. *Intel® Agilex™ FPGA Architecture. White Paper*. Oct. 13, 2022. URL: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/agilex-fpgas-game-changing-white-paper.pdf.

[31] Intel Corporation. *Intel® Architecture Instruction Set Extensions and Future Features*. Sept. 2022.

[32] Intel Corporation. [redacted] June 10, 2021.

[33] Intel Corporation. *Intel® FPGA Product Catalog Version 22.2*. July 22, 2022. URL: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/product-catalog.pdf.

[34] Intel Corporation. *Intel® Optane™ DC Persistent Memory. Quick Start Guide*. Sept. 2020. URL: https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[35] Intel Corporation. *Intel® Optane™ Memory - Revolutionary Memory: What is Optane Memory?...* 2022. URL: https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[36] Intel Corporation. *Intel® Optane™ Persistent Memory*. 2022. URL: https://www.intel.de/content/www/de/de/products/docs/memory-storage/optane-persistent-memory/overview.html.

[37] Intel Corporation. *Intel® Optane™ Persistent Memory Product Brief*. 2022. URL: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[38] Intel Corporation. *Intel® Quartus® Prime Pro Edition User Guide: Debug Tools*. July 8, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/721598?fileName=ug-qpp-debug-683819-721598.pdf.

[39] Intel Corporation. *Intel® Quartus® Prime Pro Edition User Guide: Platform Designer*. June 20, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/733065?fileName=ug-683609-733065.pdf.

[40] Intel Corporation. *Intel® Resource Director Technology (Intel® RDT)*. Oct. 3, 2022. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[41] Intel Corporation. *Intel® Stratix® 10 Configuration User Guide*. May 30, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/724219?fileName=ug-s10-config-683762-724219.pdf.

[42] Intel Corporation. *Intel® Stratix® 10 Device Datasheet*. Jan. 12, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/666450?fileName=s10_datasheet-683181-666450.pdf.

[43] Intel Corporation. *Intel® Stratix® 10 DX Device Overview*. Sept. 28, 2020. URL: https://cdrdv2.intel.com/v1/dl/getContent/670850?fileName=s10-dx-overview-683225-670850.pdf.

[44] Intel Corporation. *Intel® Stratix® 10 DX FPGA Development Kit User Guide*. Nov. 16, 2020. URL: https://www.intel.com/programmable/technical-pdfs/683561.pdf.

[45] Intel Corporation. *Intel® Stratix® 10 Embedded Memory User Guide*. Apr. 25, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/710838?fileName=ug-s10-memory-683423-710838.pdf.

[46] Intel Corporation. *Intel® Virtualization Technology for Directed I/O. Architecture Specification*. June 2022. URL: http://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf.

[47] Intel Corporation. *Intel® Xeon® Silver 4215 Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/193389/intel-xeon-silver-4215-processor-11m-cache-2-50-ghz.html.

[48] Intel Corporation. *Multi Channel DMA Intel® FPGA IP for PCI Express User Guide*. Aug. 19, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/740469?fileName=ug-683821-740469.pdf.

[49] Intel Corporation. *Product Fact Sheet: Intel's New Memory and Storage Products*. Dec. 16, 2020. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2020/12/MSM-Product-Fact-Sheet.pdf.

[50] Intel Corporation. *Stratix 10 DX FPGA DEVKIT Board - ENP*. Oct. 4, 2019. URL: https://www.intel.com/content/dam/altera-www/global/en_US/support/boards-kits/stratix10/dx_fpga/dx-dev-kit-enpirion-051219-100719.pdf.

[51] M. Jung. "Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD)." In: *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 2022, pp. 45–51.

[52] D. B. Kirk and W. H. Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[53] *Linux kernel source code: include/linux/dma-mapping.h*. Apr. 27, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/dma-mapping.h?h=v5.17.5.

[54] *Linux kernel source code: include/linux/list.h*. Apr. 27, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/list.h?h=v5.17.5.

[55] *Linux kernel source code: include/linux/mm.h*. Apr. 27, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/mm.h?h=v5.17.5.

[56] *Linux kernel source code: include/linux/refcount.h*. Apr. 27, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/refcount.h?h=v5.17.5.

[57] M. S. Manasse. "Ski Rental Problem." In: *Encyclopedia of Algorithms*. Ed. by M.-Y. Kao. Boston, MA: Springer US, 2008, pp. 849–851. ISBN: 978-0-387-30162-4.

[58] P. Maucher. *GPU4FS: A Graphics Processor-Accelerated File System*. Master Thesis. Aug. 2022.

[59] O. Mutlu et al. "A modern primer on processing in memory." In: *arXiv preprint arXiv:2012.03112* (2020).

[60] F. Nawab et al. “Dalí: A periodically persistent hash map.” In: *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[61] NVM Express Workgroup. *NVM Express Base Specification 2.0b*. Jan. 6, 2022. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0b-2021.12.18-Ratified.pdf.

[62] PCI-SIG. *PCI Express® Base Specification Revision 4.0 Version 1.0*. Sept. 27, 2017.

[63] PCI-SIG. *PCI Express™ Base Specification Revision 1.0a*. Apr. 15, 2003.

[64] *Persistent Memory Wiki*. June 12, 2020. URL: https://nvdimm.wiki.kernel.org/start.

[65] A. Raj. *Recent Enhancements in Intel® Virtualization Technology for Directed I/O (Intel® VT-d)*. Sept. 26, 2018. URL: https://01.org/blogs/ashokraj/2018/recent-enhancements-intel-virtualization-technology-directed-i/o-intel-vt-d.

[66] D. Ramanathan and R. Gupta. “System level online power management algorithms.” In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*. IEEE. 2000, pp. 606–611.

[67] Rambus Press. *DDR5 vs DDR4 DRAM – All the Advantages & Design Challenges*. Sept. 7, 2022. URL: https://www.rambus.com/blogs/get-ready-for-ddr5-dimm-chipsets/.

[68] *read(2) — Linux manual page*. Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/read.2.html.

[69] A. Redaelli, Redaelli, and Lekhwani. *Phase Change Memory*. Springer, 2017.

[70] A. Rudoff. “Persistent memory programming without all that cache flushing.” In: *SDC* (2020).

[71] P. Sanders et al. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019.

[72] *sched_setscheduler(2) — Linux manual page*. Mar. 22, 2021. URL: https://man7.org/linux/man-pages/man2/sched_setscheduler.2.html.

[73] A. Shillov. *Samsung Unveils 512GB CXL Memory Expander 2.0*. May 11, 2022. URL: https://www.tomshardware.com/news/samsung-unveils-512gb-cxl-memory-expander-2-0.

[74] G. Singh et al. “Near-memory computing: Past, present, and future.” In: *Microprocessors and Microsystems* 71 (2019), p. 102868.

[75] SuSE Labs. *numa(3) — Linux manual page*. Dec. 2007. URL: https://man7.org/linux/man-pages/man3/numa.3.html.

[76] The Go Authors. *The Go Programming Language Specification*. June 29, 2022. URL: https://go.dev/ref/spec.

[77] The IEEE and The Open Group. *The Open Group Base Specifications Issue 7 IEEE Std 1003.1™-2008*. 2008. URL: https://pubs.opengroup.org/onlinepubs/9699919799.2008edition/nframe.html.

[78] L. Werling, C. Schwarz, and F. Bellosa. *Towards Less CPU-Intensive PMEM File Systems*. Sept. 21, 2021. URL: https://www.betriebssysteme.org/wp-content/uploads/2021/09/FGBS_Herbst2021_Folien_Werling.pdf.

[79] W. A. Wulf and S. A. McKee. "Hitting the memory wall: Implications of the obvious." In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[80] Y. Xiang et al. "EMBA: Efficient memory bandwidth allocation to improve performance on intel commodity processor." In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–12.

[81] J. Xu and S. Swanson. "{NOVA}: A Log-structured File System for Hybrid {Volatile/Non-volatile} Main Memories." In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 323–338.

[82] J. Yang et al. "An empirical guide to the behavior and use of scalable persistent memory." In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 2020, pp. 169–182.

[83] M. Zea. *PCI-SIG® Releases PCIe® 6.0 Specification Delivering Record Performance to Power Big Data Applications*. Jan. 11, 2022. URL: https://www.businesswire.com/news/home/20220111005011/en/PCI-SIG%C2%AE-Releases-PCIe%C2%AE-6.0-Specification-Delivering-Record-Performance-to-Power-Big-Data-Applications.

[84] B. Zhang et al. "NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems." In: *Proceedings of the VLDB Endowment* 15.6 (2022), pp. 1187–1200.