Karlsruhe Institute of Technology

Hardware-Assisted Memory Bandwidth Management for CXL-based Memory Devices

Masters's Thesis
of

Linus Kämmerer

at the
Department of Informatics
Institute of Computer Engineering
Operating Systems Group

First examiner:	Prof. Dr.-Ing. Frank Bellosa
Second examiner:	Prof. Dr. Wolfgang Karl
First advisor:	Daniel Habicht, M.Sc.
Second advisor:	Yussuf Khalil, M.Sc.

29. April 2025 – 29. October 2025

Statutory Declaration

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, 29.10.2025

Abstract

Compute Express Link (CXL) is a new interconnect protocol that enables adding memory to a host besides native DRAM. However, the latencies of CXL devices are not on par with DRAM directly attached to the system memory bus, especially not in overload situations, i.e., when memory requests are sent out faster to the CXL device than it can handle them, resulting in exponentially growing queueing delay. In this thesis, we aim to reduce CXL access latencies caused by queueing delay, thereby also reducing stall cycles of the CPU waiting for CXL memory requests, increasing the throughput of the CPU. For this, we implement hardware memory access counters on an FPGA-based CXL device and use them in combination with Intel Processor Event Based Sampling (PEBS) to get a per-process estimate of the CXL bandwidth usage. To avoid overloading the CXL device, we limit the CXL bandwidth of processes by reducing their CPU time. We implement two approaches to enforce CPU limits: A custom scheduler using the eBPF-based `sched_ext` interface of the kernel, and a userspace program that uses Linux control groups (cgroups) to limit CPU resource usage. We evaluate how both approaches perform in keeping the latency low in overload situations while still utilizing most of the bandwidth of the CXL device, and how the CPU throughput is affected. As a result, we achieve about 30 % lower CXL memory access latencies under load and a reduction of the CPU cost for reads of about 40 % compared to an unmodified Linux.

Acknowledgements

First of all, I would like to thank my parents for gifting me a book[1] about Python 2 at the end of elementary school as well as providing me with a Linux PC, thereby introducing me to programming. Without this, I might have never become fascinated by computers and programming languages, let alone started pursuing a degree in computer science. Of course, I would also like to thank my family for all education and support they have provided.

I would like to thank my fellow students from Technische Universität Braunschweig, Karlsruhe Institute of Technology, and Aalto University (Helsinki), who became friends during my Bachelor's, Master's, and semester abroad, respectively. Together, we prepared for exams, collaborated on private software projects, and debugged countless errors. Equally, I would like to thank my non-computer science friends and my girlfriend for being a balance to my studies.

I would like to thank Prof. Dr.-Ing. *Frank Bellosa* for running the operating systems group at the institute of computer engineering at KIT, his teaching activities, and for examining this thesis. In particular, I would like to thank my thesis supervisors, *Daniel Habicht* and *Yussuf Khalil*, for offering this thesis to me, during which I have learned a lot. Especially, I would like to thank them for their support in dozens of organizational, technical, and writing matters during my thesis.

Thank you!

[1] "Python für Kids" by Gregor Lingl. mitp Verlag (2008). ISBN 978-3826686733.

Contents

Chapter 1

Introduction and Motivation

In recent years, CPUs have evolved faster than memory, so that the number of DRAM slots available on a mainboard is the limiting factor for memory capacity and bandwidth [20, 83]. In environments with high memory demands, such as big data, scientific computing, and machine learning, memory has thus become a bottleneck [20]. The development of the new Compute Express Link (CXL) interconnect protocol aims to alleviate this problem. It allows CXL devices like memory expanders and hybrid SSDs to be connected to the CPU via the electrical interface of Peripheral Component Interconnect Express (PCIe) [15, 20]. By providing a separate way of connecting memory to a CPU, CXL can be used to enhance the memory capacity and bandwidth of a system. In contrast to PCIe, CXL memory accesses are cacheable, coherent, and have a lower latency [15, 20].

However, due to the physical distance between a CXL device and the CPU, CXL memory accesses have a higher latency than accesses to DRAM. Additionally, if the used CXL memory bandwidth reaches a certain threshold, the access latency increases exponentially [14]. In this overload situation, the CPU sends out memory requests faster than the CXL memory can process and reply to them. Consequently, the requests are buffered, leading to queueing delay [14]. Since memory requests to CXL devices are synchronous, the CPU waits for the completion of memory requests by stalling. This results in wasted CPU time and energy [118]. Although Tootoonchian et al. [114] as well as Das Sharma et al. [20] showed that the DRAM access latency also increases if the bandwidth is saturated, the impact of CXL memory accesses is higher because the base latency is already twice as high compared to DRAM [83]. Consequently, it would be beneficial to avoid overload

situations by submitting CXL memory requests less frequently and instead use the CPU time to perform other meaningful work that does not rely on CXL memory. The goal of this thesis is therefore to optimize the bandwidth-latency ratio by controlling the bandwidth to keep the access latency low and utilizing the CPU time instead of stalling.

To avoid overloading the bandwidth of the CXL memory device, the OS needs to know when this situation occurs. Neither latency nor bandwidth usage information of CXL devices is directly known to the host, as discussed in detail in Section 3.3. Furthermore, the per-process CXL bandwidth is necessary to find the biggest culprits of the CXL bandwidth overload and throttle their CXL bandwidth accordingly. For this, the bandwidth monitoring solution needs to differentiate between DRAM and CXL accesses as well as identify the originating CPU core of the memory traffic, which is not possible with current tools like Intel Memory Bandwidth Monitoring (MBA) [30, 35:§19.18.1]. To then reduce the CXL bandwidth, simple throttling of the memory request rate by the OS may reduce the queueing delay at the CXL device, but the CPU may still stall for now artificially delayed CXL requests, as experienced under Intel Memory Bandwidth Allocation (MBA) [23, 29]. In contrast, we seek to avoid CPU stall cycles and use the CPU time to do meaningful work instead.

In this thesis, we implement hardware memory access counters on an FPGA-based CXL device and combine them with per-process profiling information from Intel Processor Event-Based Sampling (PEBS) [35:§21] to estimate the used CXL bandwidth per task. To keep the latency from increasing exponentially, we limit the total CXL bandwidth to a certain threshold. To do so, we use the amount of CPU time allocated to a task as a control mechanism for its CXL bandwidth: If we lower the CPU time of a task by a certain factor, we also reduce its bandwidth by the same factor because the task has less time to submit memory requests [123]. This results in an effective mechanism to limit the CXL bandwidth, and thereby also the CXL memory access latency. By evaluating our approach using the fio memory benchmark [6], we show about 30 % reduction in CXL memory access latencies under load and about 40 % lower CPU cost for accessing CXL memory.

In the following, we discuss related work in Chapter 2 and provide background knowledge in Chapter 3. In Chapter 4, we present the design of our approach consisting of the memory access counters on the FPGA-based CXL device, the per-process sampling as well as the CXL bandwidth control. Chapter 5 goes into the

details of implementing our approach, covering the modifications to the FPGA and Linux kernel. Also, it describes the CXL bandwidth control implementations. In Chapter 6, we evaluate our CXL bandwidth monitoring, the CXL bandwidth limitation and its impact on the CXL memory access latency. We further examine the CPU throughput and the overhead of our approach. Chapter 7 then discusses limitations of our work and areas for future work before drawing a conclusion in Chapter 8.

Chapter 2

Related Work

In this chapter, we refer to work that has similarities to ours, but discuss how the respective approach or goal of the authors differs from the approach presented in this thesis.

Das Sharma et al. [20] showed the potential of CXL memory expanders as bandwidth expanders by interleaving memory accesses between CXL and DRAM. In this scenario, they measured a 50 % higher bandwidth and lower latency than when only using DRAM because the load is distributed between the two memories. Sun et al. [111] also measured an improvement of using CXL for certain types of applications as well, but also showed that application performance can decrease due to the higher latency of CXL memory. This highlights the importance of controlling the latency.

Different approaches to obtaining the memory bandwidth of processes have been conducted by previous research. Lee et al. [64] explored estimating the number of memory accesses using Access Bit Tracking. Thereby, the `accessed` bit of Page Table Entries (PTEs) is cleared. If the page is accessed the next time, the CPU sets the bit again. By regularly inspecting whether this bit has been set, the OS can infer the access frequency of a page. Linux also provides a default implementation for access bit tracking called Data Access MONitor (DAMON) [50]. Lee et al. [64] reduced the overhead of access bit tracking by using a spatial sampling method, i.e., only sampling one page of a region and inferring that the whole region was accessed with that pattern.

Agarwal, and Wenisch [2] found that the overhead for Access Bit Tracking is too high because of the required frequent clearing of the `accessed` bit in the PTE. Instead,

they used Access Bit Tracking with low precision only to determine pages that get accessed at all. For those pages, they inspect page table walks to determine their access frequency. Therefore, they first poison a PTE of a page, which can be done by setting a reserved bit. The PTE is then removed from the translation lookaside buffer (TLB) by flushing it. When the page is accessed the next time, the TLB miss causes a TLB walk. Because the PTE is poisoned, it will trigger a protection fault, which can be intercepted by BadgerTrap [24]. By counting the number of these protection faults, the number of TLB misses can be counted. However, the overhead of Access Bit Tracking increases significantly with the size of the address space [83, 99], which stands in conflict with the idea of CXL memory expanders offering large amounts of memory.

In their work around transparent page placement (TPP), Maruf et al. [83] explored moving less frequently accessed pages from DRAM to slower CXL memory, while maintaining nearly no performance degradation. For this, they used Intel Processor Event-Based Sampling (PEBS) to sample the memory access behavior of tasks. By storing the per-page activity, they obtain an activity history, which allows them to move cold memory pages to CXL memory. In general, this approach is focused on the hotness of pages, not the tasks accessing them. As pages can be shared between processes, the hotness of a page cannot be used to attribute the bandwidth usage to individual processes. In our approach, we also increase the precision of the PEBS-based bandwidth estimation by taking into account the memory access sizes of instructions sampled by PEBS as well as combining it with hardware memory access counters on our FPGA-based CXL device.

Sun et al. [110] adapted an FPGA-based CXL device to include Page Access Counter (PAC) and Word Access Counter (WAC) to get precise profiling information about the number of CXL accesses. These access counters are exposed via a CXL.io memory-mapped IO area to the host. Similarly, Zhou et al. [126] also implemented per-page memory access counters on an FPGA-based CXL device. Different from the focus of our work, both approaches count the number of memory accesses per page (or also per word). We, however, want to attribute the used CXL bandwidth to individual processes. If processes share memory pages, per-page counters cannot be used to differentiate between accesses from different processes. Although both works showed that it is possible to have many performance counters on a modern FPGA while keeping the target clock frequency, they did not have the challenge that

the CXL device had to be mapped into the address space multiple times, as we do it to represent monitoring groups as NUMA nodes.

Fried et al. [23] developed the Caladan scheduler to minimize tail latencies of applications. Typically, resource partitioning is used to avoid different applications from influencing each other performance. Instead of resource partitioning, they partitioned tasks into low-latency (LC) and best-effort (BE) tasks, where LC tasks are prioritized while BC tasks may use the remaining memory bandwidth. To identify overload situations, they retrieve the total used DRAM bandwidth from the CPU, which can be done without hardware modifications, and check if it exceeds a certain bandwidth threshold. Additionally, they use per-core last level cache (LLC) miss rate counters exposed by the performance monitoring unit of a CPU as a metric for inferring the bandwidth share of individual tasks. For our CXL bandwidth control, we also need to differentiate between accesses to DRAM and CXL memory, which prevents us from using these counters. Caladan sets itself apart from other related work by directly identifying the source of interference instead of slowly and incrementally converging to a new resource allocation. Similarly, we use the per-process CXL bandwidth obtained from the combination of PEBS and FPGA access counters to directly identify tasks with CXL usage and compute their optimal CPU time limits. Fried et al. further mention that Intel Memory Bandwidth Allocation (MBA) [35:§19.19.7] is not suitable because it limits memory bandwidth usage of processes by injecting delays into memory accesses, resulting in a poor CPU utilization. This is also contrary to our goal of improving CPU throughput. In contrast to setting CPU time limits, their work uses core allocation (i.e., limiting the number of cores allocated to an application) to limit the bandwidth of tasks. For us, core allocation is not sufficient because even a few applications each using a single CPU core can saturate the low bandwidth of our CXL device, as detailed in the evaluation in Chapter 6. Another difference to our approach is that Caladan requires its own runtime to be linked into all running processes. The runtime provides green threads, which are executed via a work-stealing thread pool. For Caladan to work best, applications are required to use these green threads for concurrency and need to spawn a new green thread per request to allow the runtime to measure the request processing times and queueing delays. For tasks not using the runtime's green threads, Caladan uses time multiplexing by only partially scheduling them onto a

CPU. This is similar to our approach, but to the best of our knowledge, the authors of Caladan did not include this feature in their evaluation.

Yi et al. [123] focused on the performance optimization of tasks using persistent/non-volatile memory (NVM). They use Intel Memory Bandwidth Monitoring (MBM) [35:§19.18] to retrieve the total memory bandwidth. For read accesses of each process, they utilize the NVM and DRAM access counters of the CPU. To get the number of writes conducted by each process, they use two approaches. For applications using the PMDK library to access the NVM memory, the authors hook certain library functions that flush cache lines or perform non-temporal stores to count the number of accesses and report them to the kernel. If the PMDK library is not used, they resort to estimating the bandwidth using PEBS-based memory access sampling. The write bandwidth of DRAM can then be calculated from all other values. Additionally, they periodically issue a few writes to determine the latency, which is used as a metric for bandwidth interference. However, they note that MBA works by injecting delays into the application memory accesses, which can lead to a different resulting bandwidth depending on the application. Throttling of bandwidth-hungry tasks is either done immediately or only if the measured memory access latency is above a certain threshold. Tasks are throttled via Intel Memory Bandwidth Allocation (MBA) [35:§19.19.7] if they use an excessive amount of DRAM bandwidth, and via a cgroup CPU quota [51] if they use too much NVM memory bandwidth. The latter approach using cgroups CPU limits is similar to one of our CXL bandwidth control approaches described in Section 5.4.2. For MBA, they found that it is more effective for limiting the DRAM bandwidth, but less effective in limiting NVM bandwidth when compared to the cgroups-based CPU limitation. Specifically, they found that CPU time limitations proportionally reduce the resulting memory bandwidth of the application, which is an essential characteristic for our cgroups-based CPU time bandwidth limiting approach.

Petrucci et al. [97] used a CXL FPGA to log page accesses and make this log available to the OS, which can then use this information for page promotion in memory tiering systems. They found their approach to better identify hot pages than when using PEBS. However, we question the overhead for transferring and processing the access logs on the host, and consider our hardware memory access counters as more efficient.

Chapter 3

Background

In this chapter, we provide background information on topics that are important in the context of this thesis. Specifically, this includes an overview of CXL (§ 3.1), Chisel and FPGAs (§ 3.2), the Performance Monitoring Unit (PMU) built into modern CPUs (§ 3.3), Linux Loadable Kernel Modules (LKMs) (§ 3.4), Extended Berkely Packet Filters (eBPF) (§ 3.5), and the eBPF-based Linux scheduler interface `sched_ext` (§ 3.5.1).

3.1 Compute Express Link (CXL)

CXL is a new interconnect protocol based on Peripheral Component Interconnect Express (PCIe), allowing to connect devices such as memory expanders and hybrid SSD to a host [15, 20]. CXL adapted the physical, link, and transaction layers from PCIe into its own protocol stack called Flex Bus [15]. It emerged from various proprietary protocols like OpenCAPI, GenZ, and CCIX [20].

Because CXL uses the physical and electrical interface of PCIe, memory extension devices can be added to a system easily [20]. This provides a separate way to extend the memory of a system besides native DRAM [20]. Although the DRAM capacity available to a CPU increased over the years, the number of DRAM slots on a mainboard and the capacity per DRAM DIMM did not grow as fast as the CPU core count, effectively reducing the memory available per core [20]. Especially for memory-intensive workloads like scientific computing and machine learning, a limited amount of memory can become a bottleneck. By offering another way to add memory to a system, CXL provides a solution to the memory wall [121].

The focus of CXL is to provide a communication protocol optimized for storage devices by providing load/store access. For this, a CXL request only transfers a cache line of 64 bytes [15, 20]. Another important factor for boosting the performance of CXL is cache coherence, which allows the CPU to cache frequently accessed data in its caches [15, 20]. Together, these improvements over PCI lead to an increased performance [4, 43, 111].

3.1.1 Access Latency

Compared to DRAM, the access latency of CXL devices is higher. While DRAM has an access latency of 80-140 ns [4:figure 7, 83:figure 2], the idle access latency of a CXL memory expander is already twice as high [20:§6.1, 65, 103, 109, 113], which is comparable to the latency of a remote-NUMA DRAM access [20:§6.1, 82, 83, 87, 109, 113]. The access latency of CXL hybrid SSDs is typically even higher than that of CXL memory expanders. Samsung [103] as well as Yoo et al. [124] reported access latencies of around 600 ns for first hybrid CXL SSD prototypes, which is around six times as high as native DRAM [4:figure 7, 83:figure 2]. The performance of CXL devices also varies depending on the CXL controller [111] and, in the case of a hybrid SSD, its cache management [63].

If the data of a CXL memory access is available in a CPU cache, the access latency is low and the access latency can usually be hidden by out-of-order execution [28]. In case data is not available in a cache, an actual memory access to the CXL device has to be performed. If the CPU is not able to fully hide the access latency by instruction reordering [28], the CPU pipeline needs to stall, i.e., execute NOP instructions [28]. This is because memory requests are synchronous, forcing the CPU to wait for them to complete[2] [28:§3]. This stands in contrast to communication with IO devices, which is typically asynchronous and interrupt-based, thus allowing the CPU to do other work in the meantime [28:figure C.26]. Despite being based on PCIe, CXL memory requests are synchronous and can thus also cause CPU stalls [65].

Although future optimizations of the CXL protocol and hardware improvements might decrease the latency of CXL devices, CXL communication will always have longer signal paths (i.e., the physical distance from the CPU to a CXL device) than

[2] The CPU does not necessarily need to wait for writes to be completed under a *write back* cache policy. However, if a *write allocate* cache policy is used, the cache line to be written is first read into the cache and then modified, making a write behave like a read [28:B.1]. Write allocate is also used for CXL on Intel CPUs [110, 111].

DRAM, inherently making CXL a medium with higher access latency. However, CXL is also not necessarily supposed to directly compete with DRAM or supersede it. Instead, CXL is often considered as a background store for memory tiering scenarios, where cold pages are moved from the native DRAM to the CXL device instead of moving them to a much slower swap space residing on an SSD [2, 64, 110, 126]. Nevertheless, optimizing CXL performance as best as possible can still have a big impact if the bandwidth of a CXL device is saturated, because the access latency increases significantly over the idle latency [4, 14, 20] due to queueing delay, i.e., buffering of memory requests [15], as shown in Figure 1.

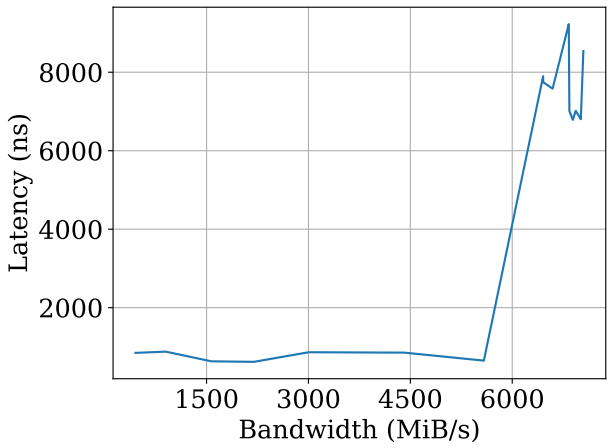


Figure 1: CXL memory access latency of our FPGA-based CXL device depending on the used bandwidth, measured using the Intel Memory Latency Checker (MLC) [116].

3.1.2 CXL Device Types

There are different variations of CXL memory devices. For example, CXL memory expanders typically feature DRAM, which they expose as memory via CXL [20]. A different concept are hybrid SSDs, which offer flash memory that can be accessed via a file system while also allowing a direct mapping to physical memory [63]. In contrast to PCIe, which is interrupt-based, requests to CXL devices use load/store accesses. In general, CXL provides three protocols and three device types.

The *CXL.io* protocol is similar to PCIe in that it uses “non-coherent load-store semantics” for “device discovery, status reporting, virtual to physical address trans-

lation, and direct memory access (DMA)" [20]. Every CXL device type implements CXL.io for bootstrapping purposes [20].

The *CXL.mem* protocol is used to expose the memory of the CXL device to the CPU and allows this data to be cached by the host [20]. CXL type 2 devices as well as CXL type 3 devices, also called memory expanders, use this protocol to provide additional memory capacity and bandwidth to the host [20, 111, 113, 121].

The *CXL.cache* protocol allows a CXL device to cache memory from the host while being cache-coherent [20]. This is used by CXL type 1 and 2 devices in the form of accelerators like FPGAs and smart NICs to cache data from the host for local computations on the CXL device [20].

The main focus of this thesis are CXL type 3 devices, i.e., memory expanders using the CXL.mem protocol [43, 121]. Although we use an FPGA-based CXL device for our implementation, we do not use it as an application-specific accelerator (suggesting a CXL type 2 device [109]), but as a modifiable memory expander implementation instead of an ASIC. For a CPU to be able to communicate with a CXL device, the CPU needs to have hardware support for CXL [20], which is available starting from the 4th generation of Intel Xeon Scalable CPUs (codename Sapphire Rapids) [32] and the 4th generation of AMD EPYC CPUs (codenames Genoa and Bergamo) [1]. With the higher per-pin bandwidth of CXL compared to DRAM, memory expanders can be used to overcome the memory wall of DRAM [14, 87, 121]. When servers are being upgraded to DDR5, old DDR4 DIMMs can be reused in CXL memory expanders to provide additional memory capacity to servers [84, 85]. Hybrid CXL SSDs backed by flash memory can also offer persistent memory [104]. Additionally, CXL versions 2.0 and 3.0 added support for memory pooling and memory disaggregation [20], allowing multiple servers to share CXL devices, which can reduce resource stranding, i.e., unused resources in a data center [65]. For an in-depth introduction to CXL, we refer to "An Introduction to the Compute Express Link (CXL) Interconnect" [20] and the CXL specification [15].

3.2 Chisel and FPGAs

Field Programmable Gate Arrays (FPGAs) allow for programmatically modifying the configuration of a hardware device, thereby allowing prototyping without needing to manufacture an actual Application-Specific Integrated Circuit (ASIC) [11]. In this

thesis, we use an FPGA to implement a CXL memory expander. Typically, FPGAs are programmed using Verilog or VHDL. We, however, use Chisel [13] to extend the memory expander FPGA implementation by Khalil [60] with hardware CXL memory access counters. Chisel is a relatively new hardware description language (HDL) compared to Verilog and VHDL, as it was introduced by Bachrach et al. in 2012 [8]. It uses a different approach compared to traditional HDLs in that it is implemented as a domain-specific language (DSL) on top of the JVM-based Scala programming language [106]. When executed, the Chisel/Scala program generates Verilog code using LLVM's CIRCT (Circuit IR Compilers and Tools) compiler [81]. The generated Verilog code is then compiled using the FPGA vendor's tools as usual [106]. In our case, we use the Intel Quartus Prime [38] toolchain to synthesize the bitstream for our Intel/Xilinx FPGA. Also, the tools for flashing the FPGA from the Verilog workflow are used as usual. For a comparison between Chisel, VHDL, and Verilog, we refer to [106:Appendix A].

3.3 Performance Monitoring Unit (PMU)

The Performance Monitoring Unit (PMU) of a CPU is a set of hardware counters that count certain events in the CPU [36:§2.1.7]. The counters are implemented via model-specific registers (MSRs) [3]. On the one hand, there are *fixed counters*, which are statically assigned to the events they are counting, e.g., clock cycles [34:§22.3]. On the other hand, there are *programmable counters*, which can be configured in what events they should count, e.g., interrupts or cache loads [34:§22.3, 36:§2.1.7]. For Intel processors, which are the focus of our work, there is a list of available events [19]. The OS can communicate and configure the PMU by using special instructions to modify the MSRs[3] while the Linux tool perf [79] acts as a bridge to userspace, allowing applications (running as root) to configure and read the performance counters. Hardware events are further categorized into *core* and *uncore* events [34:§22.3]. If an event is core, it means that it is counted per individual CPU core [34:§22.3]. If an event is uncore, there is only one counter shared by all cores of a CPU [35:§21.3.1.2]. In this case, event counts cannot be attributed to individual cores, and thus also not to specific processes running on those cores.

[3] `WRMSR` for writing to a model specific register, `RDMSR` for reading from a model specific register, and `RDPMC` to read performance monitoring counters, see [37:§5.20].

For this thesis, we are interested in attributing CXL bandwidth usage to individual tasks. Therefore, we need to differentiate between memory requests targeting DRAM and CXL memory, as well as obtaining information which CPU core performed an access to be able to attribute it to a specific task [105:§2.2.2]. On the one hand, there is the PMU event `MEM_LOAD_RETIRED.LOCAL_PMM`, which is a core event, but is only counting loads and only for PMM (Intel Optane), but not for CXL-based devices [19]. On the other hand, there is the CXL-related event `UNC_CHA_TOR_INSERTS.IA_MISS_CXL_ACC`, which counts "all requests issued from IA cores to CXL accelerator memory regions that miss the LLC" [19], but as an uncore event, it cannot be used to attribute CXL bandwidth to individual cores [19]. Another example is the event `UNC_CXLCM_TxC_PACK_BUF_INSERTS.MEM_DATA` used by the Intel Performance Counter Monitor (Intel PCM) [39, 120], which is an event related to CXL, but again an uncore event[4] [59]. To the best of our knowledge, the intersection of an event being a core event and being CXL-related is unfortunately empty. Therefore, we cannot simply derive the per-process CXL bandwidth usage from PMU counter values but need to rely on a sampling-based approach instead.

If a PMU counter overflows, the OS gets a Performance Monitoring Interrupt (PMI) [35:§21.6.3]. This can be used in the form of *interrupt-based event sampling* to periodically inspect the processor state [35:§21.6.3]. Initially, the counter is set to a reset value of $-r$ to cause the counter to overflow after r events[5] [3]. On overflow, the interrupt handler can then increment a software counter and reset the hardware event counter. Also, the interrupt handler can save some additional information like the current instruction [122]. Due to instruction pipelining in the CPU, there can be a *skid*, i.e., a difference between the instruction that caused the event and the instruction pointer when the CPU is interrupted [34:§22.5.1, 122]. Another problem is that the interrupt and its handling have a performance overhead [9, 83, 99].

3.3.1 Processor Event-Based Sampling (PEBS)

An optimization to the interrupt-based sampling approach is the Processor Event-Based Sampling (PEBS) feature of Intel CPUs [3, 35]. AMD's Instruction Based Sampling (IBS) [25] provides similar features, but is out of scope for this thesis

[4] See `tools/perf/pmu-events/arch/x86/emeraldrapids/uncore-cxl.json` in the Linux code [59]

[5] The counter value is an unsigned integer, thus $-r$ in the two's complement is equal to $2^n - r + 1$, causing the overflow to be triggered after exactly r events occurred.

since we focus on Intel CPUs. The main feature of PEBS is that it does not trigger an interrupt when the counter overflows, but invokes a CPU micro-code routine called *PEBS assist* instead [3], which writes a sample into a memory buffer [3, 35:§21.6.3]. Only when the PEBS buffer is becoming full, the OS gets an interrupt and empties the buffer, which reduces the number of interrupts and therefore also the overhead [3]. Because the CPU does the sampling directly without an interrupt, there is less skid [3, 35:§21.9.4]. Modern Intel CPUs also feature precise events where the skid is one [34:§22.5.1] or the Precise Distribution (PDist) where the skid is zero [35:§21.9.6]. Therefore, PEBS is sometimes also referred to as *Precise* Event-Based Sampling (PEBS) [3, 34:§B.6.4.1].

For running PEBS with minimal overhead, the optimal buffer size needs to be found. On the one hand, if the PEBS buffer is too small, the OS needs to handle the interrupt and empty the buffer frequently, reducing the benefit of the interrupt-free sampling of PEBS [3]. On the other hand, if the PEBS buffer is too large, it can cause cache pollution and additional memory IO when application data and the PEBS buffer contend for cache space [3]. Additionally, the PEBS samples can be older. Furthermore, the sampling frequency also needs to be considered. Akiyama, and Hirofuchi [3] found that the execution of the PEBS assist routine takes around 200-300 ns, which would be around 1000 cycles for a processor running at 4 GHz. A comparison with a typical CXL.mem idle access latency of around 170-250 ns [83] shows that sampling each memory access with PEBS would substantially increase the access latency of each memory access, and is therefore not feasible. Instead, only every n-th access is sampled using the reset value that can be configured. Zhou et al. [126] found that sampling every 100th access causes an overhead of 15%. Yi et al. [122] suggested that the sampling period should be a prime number (e.g., 10007, 2347, or 991) to avoid any regular pattern, which would result in always the same instructions being sampled. The sampling frequency and the size of the PEBS buffer also determine in which interval our CXL bandwidth limitation implementation receives new CXL bandwidth usage information and thus sets a lower limit for the reaction time to bandwidth changes.

3.4 Linux Loadable Kernel Modules (LKMs)

Loadable Kernel Modules (LKMs) are kernel code with full privileges that can be dynamically loaded into the kernel. A common use case are device drivers [16]. An LKM is made up of a C source code file that defines at least an `init` and a `cleanup` function. These functions are called when loading the module into the kernel or when removing it from the kernel, respectively. To compile an LKM, the header files for the current kernel version need to be installed and linked into the LKM [102:§2]. A special Makefile is used to interface with the kernel build mechanism `kbuild`. After compilation, an LKM can then be dynamically loaded into the kernel via `insmod` or `modprobe` and removed via `rmmod` [47, 102]. For further details about kernel modules, we refer to *The Linux Kernel Module Programming Guide* by Salzman et al. [102].

3.5 Extended Berkely Packet Filter (eBPF)

The Extended Berkely Packet Filter (eBPF) infrastructure allows running *sandboxed* programs inside the kernel [21, 52]. It emerged from Berkely Packet Filters (BPF), which allowed filtering network packets [100]. Today, eBPF programs can be attached to various hooks in the kernel: “Pre-defined hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others. If a predefined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications.” [21]. BPF maps implement data structures like arrays or hash maps that can also be shared and accessed from user space [21]. Code is typically written in an eBPF-compatible subset of the C programming language, denoted by the file extension `.bpf.c`. Other languages such as Rust can be used to generate eBPF bytecode as well [7]. When loading an eBPF program into the kernel, a verifier ensures the safety of the eBPF program by checking that it does not contain unbounded loops or illegal memory accesses that could corrupt the kernel [100]. Often, the libbpf library [66] is used by a userspace program to load an eBPF program into the kernel. According to the kernel documentation regarding libbpf [53], a typical workflow consits of an *open phase* where the eBPF object file is parsed and global variables are assigned, a *load phase* where the eBPF program is loaded into the kernel and verified by the kernel eBPF verifier, and the *attachement phase* where the program is attached to hook points in the kernel, causing it to be executed.

Additionally, the libbpf library supports the compile once - run everywhere (CO-RE) feature powered by BPF Type Information (BTF) included in eBPF programs [53]. This allows eBPF programs to run on different kernel versions than they were compiled for. If the layout of struct fields and their offsets differ, libbpf matches the referenced fields with fields from the current kernel based on the name and rewrites the eBPF program before loading it into the kernel [53]. This means that no or less maintenance work has to be spent for porting a `sched_ext` scheduler to a new kernel version in comparison to a kernel module.

We refer to the book "Introduction to eBPF" [100] for further details about eBPF, although it focuses on the development of eBPF programs with the help of the BCC toolchain [42]. For the raw Linux eBPF system call interface, we refer to the Linux BPF documentation [52], the man pages [68, 69], and to ebpf.io [21]. Since we use the clang C compiler in combination with the libbpf library [66], we also refer to ebpf.io [21], as it provides helpful documentation of libbpf functions.

3.5.1 sched_ext Scheduler Interface

The Linux kernel offers the `sched_ext` interface for writing custom schedulers using eBPF [21, 52, 70, 70]. For this, the kernel needs to be built with the configuration option `CONFIG_SCHED_CLASS_EXT` enabled, which requires a kernel with version 6.12 or higher [70]. The advantage of `sched_ext`, being powered by eBPF, is that the scheduler can be loaded into the kernel at runtime without requiring a kernel recompilation. Additionally, overhead is rather low because the code is transformed into native machine code by the eBPF JIT and avoids context switches to userspace [100:§1]. Furthermore, the kernel installs a watchdog to monitor that tasks do not starve, and restores the default scheduler otherwise [70].

A `sched_ext` scheduler is an eBPF program that provides an instance of the `struct sched_ext_ops`, which is a struct containing function pointers [70]. This can be considered similar to an interface in object-oriented programming languages [16:§3]. The struct and its functions can be found in the Linux source code in `kernel/sched/ext.c` [59]. If a function pointer is set to `NULL`, a default implementation will be used [22]. During scheduling, the Linux scheduler framework then invokes the provided functions [70].

Chapter 4

Design

In this chapter, we outline the concept of our approach. In general, the goal of this thesis is to optimize the scheduling of tasks in such a way that the available bandwidth of the CXL device is utilized but not over-utilized, which would lead to queueing delay. We aim to avoid this situation because it offers only marginally higher memory bandwidth while causing exponentially higher access latencies, causing the CPU to waste time by stalling. Consequently, tasks with high CXL memory access bandwidth consumption should be throttled, and other, less CXL-intensive tasks should be scheduled instead. To schedule tasks accordingly, the share of the bandwidth used by each task is required. Since CXL accesses use load/store instructions, the OS is not involved nor aware of the ratio between DRAM and CXL memory accesses [126]. Hence, we extend the FPGA-based implementation of a CXL memory expander by Khalil [60] with memory access counters. Ideally, each task would be uniquely mapped to a dedicated pair of such hardware read/write memory access counters. However, there can be hundreds or thousands of tasks running on a server, but the hardware space on a circuit, or on an FPGA in our case, is constrained, which limits the amount of memory access counters. Therefore, we group tasks into monitoring groups that share a pair of read/write performance counters. The FPGA-based CXL device exposes each monitoring group as a virtual copy of its native device-physical address space. As an abstraction for monitoring groups in the OS, we use virtual memory-only NUMA nodes to which tasks can be memory-bound to. To get an estimate of the bandwidth distribution of the tasks inside a monitoring group, we use Intel's Processor Event-Based Sampling (PEBS) [35] to estimate the memory access behavior of the tasks. By combining the per-monitoring group hardware access counters of the FPGA-based CXL device and the PEBS-based memory

access sampling, we compute an estimate of the used CXL bandwidth of a task. To keep the CXL device at its optimal operating point, we limit the CPU time of CXL-intensive tasks. By throttling these tasks, we prevent CXL memory requests from being submitted faster than the CXL device can process them, avoiding queueing delay. Instead of waiting (i.e., stalling/NOP-ing) for queued CXL memory requests to complete, the saved CPU time can be used to do meaningful work instead by scheduling tasks that use less or no CXL bandwidth. At the same time, CXL-intensive tasks should not starve, and the bandwidth of the CXL device should still be utilized up to a configurable threshold. It is not the goal to only throttle the bandwidth usage of programs by injecting idle cycles, which is done by Intel Memory Bandwidth Allocation (MBA) [29] [23, 35:§19.19.7], as discussed in Section 2. Wasting CPU cycles by idling or stalling is exactly what we want to avoid. Instead, we aim to increase the CPU throughput.

Figure 2 shows an overview of our approach. In the following sections, we first provide an overview of the hardware support for CXL bandwidth monitoring on our FPGA (§ 4.1). Then, we describe the PEBS-based per-process CXL bandwidth estimation (§ 4.1.1) before explaining the CXL bandwidth control (§ 4.3).

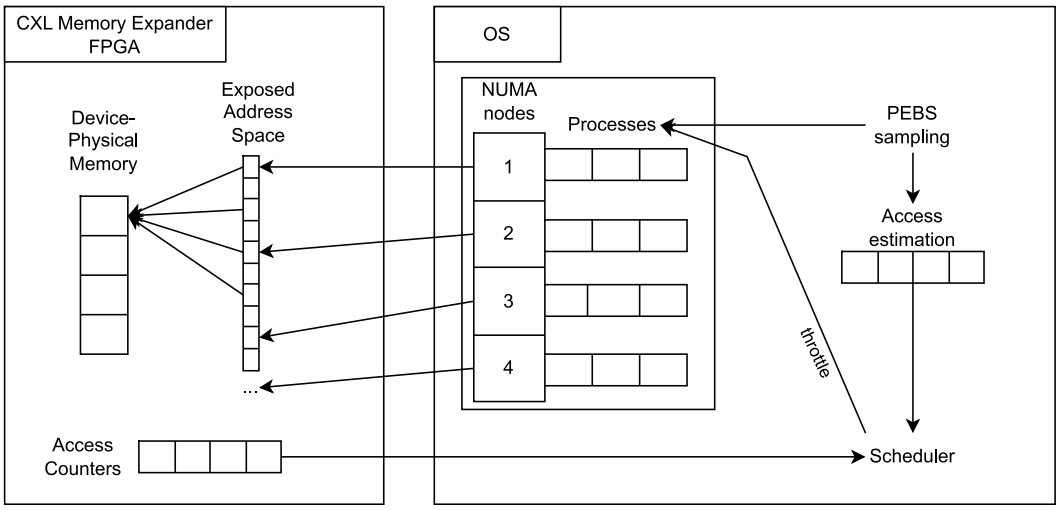


Figure 2: Overview of our approach. The CXL FPGA is extended with memory access counters, each abstracted as a virtual copy of the device-physical address space of the CXL device. In the OS, these monitoring groups are abstracted as virtual memory-only NUMA nodes. PEBS-based sampling is used to estimate the bandwidth share of tasks inside one monitoring group. By combining both metrics, the scheduler throttles tasks to keep the CXL bandwidth below a certain threshold.

4.1 CXL FPGA Bandwidth Monitoring

We modify the FPGA-based CXL device by Khalil [60] so that it exposes multiple virtual copies of its address space. The resulting address space exposed to the OS is then n times as large as the actual address space of the RAM installed on the CXL FPGA. In the following, we will use the term *device-physical address (space)* to refer to the actual installed memory on the CXL FPGA, and *host-physical address (space)* to refer to the view of the OS involving the virtual copies of the CXL FPGA. On the CXL FPGA, an incoming host-physical address is mapped to a device-physical address. For this, the lower bits of the device-physical address are used to address the memory on the FPGA. The higher bits serve as an identifier for the accessed virtual copy and are used to account the access to the correct monitoring group and increment the corresponding memory access counter.

4.1.1 CXL FPGA Memory Access Counters

To count the number of memory accesses on the CXL device, we implement two counters per monitoring group on the CXL FPGA that count the number of completed load or store memory accesses, respectively. The CXL FPGA exposes these counter values via a memory-mapped IO area that the host can access. To retrieve the counter values in the OS, we implement a kernel module. By regularly fetching these CXL access counters from the FPGA, the CXL bandwidth control can calculate the bandwidth of the current interval. The interval in which the counter value is read needs to be chosen such that it does not cause too much overhead on the OS side while being frequent enough to quickly react to changes in the bandwidth usage [23].

4.2 Per-Process PEBS-based Memory Access Sampling

The CXL hardware memory access counters we implement on the FPGA CXL only provide the CXL bandwidth usage of each monitoring group. The number of these monitoring groups is limited by hardware constraints. To get an estimate of the bandwidth distribution between processes inside each monitoring group, we sample memory accesses using PEBS by adapting the Linux kernel module developed by Werling et al. [117]. Although the tool perf [71] also allows profiling memory accesses, the kernel module has more control over the PMU, allowing live processing of data from the sampling. Additionally, the module can already distinguish between two memory types (DRAM and persistent memory) by translating virtual to physical addresses. Depending on the sampling frequency, every n-th memory access is then sampled by PEBS, and the sample is written into the PEBS buffer. In our case, the OS also needs to empty the buffer on a context switch to guarantee that CXL memory accesses are accounted to the correct process. The memory accesses sampled by PEBS are virtual addresses, so they need to be resolved to physical addresses to check if they belong to DRAM or to CXL memory. For actual memory accesses, this lookup is done automatically by the Memory Management Unit (MMU) [112:§3.3] but for the virtual addresses sampled by PEBS, we need to manually do this lookup in software, i.e., walk the page table to find the physical address. To account for the sampling interval, i.e., that only every n-th load/store is sampled, the counter is incremented by the sampling interval, assuming that the one sampled read/write represents n non-sampled read/writes, as done by Werling et al. [117].

Note that PEBS samples, even after verifying that these lie inside a CXL memory region, contain all memory accesses, regardless of whether they were served by a CPU cache or if they had to reach out to the CXL device. Thus, the resulting bandwidth usage distribution inside the monitoring groups is only an approximation. By combining it with the hardware access counters of monitoring groups, we improve the precision of the CXL bandwidth estimation of processes.

4.3 CXL Bandwidth Control

To keep the CXL memory access latency from increasing exponentially, we limit the CXL bandwidth to a bandwidth threshold. For this, the memory access counters from the FPGA-based CXL device and from the PEBS-based sampling have to be each converted to a bandwidth (§ 4.3.1) before combining them (§ 4.3.2). Based on the CXL bandwidth of individual tasks, we can then calculate their CPU limit to reduce the total CXL bandwidth (§ 4.3.3).

4.3.1 Bandwidth Calculation from Counter Deltas

Both the hardware memory access counters of the FPGA-based CXL device for each monitoring group and the per-process counters from the PEBS-based sampling result in integer counter values. By storing an old value together with its timestamp, we can calculate the bandwidth in this interval. Let c_1 and c_2 be memory access byte count values at timestamps t_1 and t_2, respectively. The bandwidth b can then be calculated as:

$$b = \frac{\Delta c}{\Delta t} = \frac{c_2 - c_1}{t_2 - t_1} \tag{1}$$

4.3.1.1 Exponentially Weighted Moving Average

A task can change its CXL usage behaviour over time. For example, it could try to acquire a semaphore/mutex, resulting in the process being paused until the requested resources are available. In this period, the process did not perform any memory operations on the CXL device. The next time the process is scheduled, it could use an extensive amount of memory bandwidth, however. Therefore, it makes sense to consider the history of bandwidth usage of a process. An efficient mechanism is the usage of an exponentially weighted moving average because only

one value needs to be stored. According to the "NIST/SEMATECH e-Handbook of Statistical Methods" [27:§6.4.3], it is defined as

$$\begin{aligned} s_0 &= x_0 \\ s_t &= \alpha \cdot x_t + (1 - \alpha) \cdot s_{t-1} \end{aligned} \tag{2}$$

where $0 < \alpha < 1$ is the *smoothing factor*, which determines the ratio with which the current value and the previous running average are weighted.

4.3.2 Combination of PEBS and FPGA-based Bandwidth

The PEBS-based access sampling of a task is just an estimation. The CXL FPGA memory access counters are precise, but if multiple processes share a CXL monitoring group, they also share the CXL memory access counter on the FPGA. Therefore, the PEBS-estimated CXL bandwidth of a task is used to calculate a share of the total bandwidth of the monitoring group. The combination of these two values is more precise than if only a single value is used.

Let a task t from the list of all tasks T be bound to monitoring group $g(t)$. Further, let the task t have a PEBS-estimated CXL bandwidth of $b_{\text{PEBS}}(t)$ and its monitoring group a CXL FPGA-measured bandwidth of $b_{\text{FPGA}}(g(t))$. Then, the combined bandwidth estimation of a task can be computed as a share of its PEBS-estimated bandwidth among the PEBS-estimated bandwidth sum of all tasks also using this monitoring group, multiplied by the CXL FPGA-measured bandwidth of its monitoring group:

$$b(t) = b_{\text{FPGA}}(g(t)) \cdot \frac{b_{\text{PEBS}}(t)}{\sum\limits_{t' \in T \mid g(t') = g(t)} b_{\text{PEBS}}(t')} \tag{3}$$

4.3.3 CPU Time Limitation

In our approach, there is a device-specific CXL bandwidth threshold b_{thres} under which the CXL memory access latency stays relatively low. If the total CXL bandwidth b_{total} exceeds the bandwidth threshold, the latency increases exponentially. Consequently, we limit the total CXL bandwidth to the bandwidth threshold by throttling the CPU time of individual tasks. We can only consider the total CXL bandwidth to determine the CPU limit of a task § 4.3.3.1, or take into account the individual bandwidth of each task § 4.3.3.2.

4.3.3.1 Task Throttling Based on the Total Bandwidth

For a task t, we calculate its CPU limit based on the bandwidth it contributed to the overload. This *task overload bandwidth* is the bandwidth share of a task ($\frac{b(t)}{b_{\text{total}}}$) times the total overload bandwidth ($b_{\text{total}} - b_{\text{thres}}$). By dividing the task overload bandwidth through the previously used bandwidth of the task, we get the CPU limit for the task, i.e., the CPU usage percent the task may still use. Equation 4 shows the corresponding equation.

$$
\begin{aligned}
b_{\text{total}} &= \sum_{g \in \text{ monitoring groups}} b_{\text{FPGA}}(g) \\
\text{cpu\_limit}(t) &= \text{cpu\_limit}(t) \cdot \left(1 - \left(\frac{\frac{b(t)}{b_{\text{total}}} \cdot (b_{\text{total}} - b_{\text{thres}})}{b(t)}\right)\right) \\
&= \text{cpu\_limit}(t) \cdot \left(1 - \left(\frac{b(t)}{b_{\text{total}}} \cdot (b_{\text{total}} - b_{\text{thres}}) \cdot \frac{1}{b(t)}\right)\right) \\
&= \text{cpu\_limit}(t) \cdot \left(1 - \left(\frac{b_{\text{total}} - b_{\text{thres}}}{b_{\text{total}}}\right)\right) \\
&= \text{cpu\_limit}(t) \cdot \left(\frac{b_{\text{thres}}}{b_{\text{total}}}\right)
\end{aligned}
\tag{4}
$$

By simplifying the equation, the bandwidth of the task ($b(t)$) gets canceled out. In this case, each task is forced to reduce its CPU time by the same percentage, which also causes each task to reduce its CXL bandwidth by the same percentage. However, this means a task with negligible CXL bandwidth usage has to reduce its CPU usage by the same amount as a task that uses lots of CXL bandwidth and is thus much more responsible for the overload situation.

4.3.3.2 CPU Limitation Based on Individual Task Bandwidth

To make the CPU throttling more fair, we want to ensure that the biggest culprits of the CXL bandwidth have to sacrifice more CPU time than tasks that only contribute very little to the overload. At the same time, we also need to deal with situations where there are many tasks with low individual bandwidth, but in sum still overwhelm the CXL device. In a naive approach, we could multiply the general bandwidth reduction factor (e.g., 20 %) needed for reaching the target bandwidth threshold with the bandwidth share of the task (e.g., 50 %) to obtain the CPU

reduction factor for the specific task. However, to reach the target bandwidth, the CPU usage has to be reduced by the general CPU usage reduction factor on average. Here, that is obviously not the case, since each task is throttled less than the total reduction factor (e.g., $50 \sim \% \cdot 20 \sim \% = 10 \sim \% < 20 \sim \%$). To circumvent this problem, we developed the following algorithm:

First, we create an array of all tasks using CXL and sort it by the bandwidth usage of each task. Then, we search the first index where the bandwidth *sum* of the left part of the array is larger than the sum of the right part of the array, while ensuring that each part has at least one entry. At this point, the two parts have a CXL bandwidth sum that is as similar as possible. We call this the *balance point* of the array. This is different from the mean and the median, which do not ensure that the sum of both parts is as equal as possible.

The left part of the array now consists of tasks that use less bandwidth than tasks in the right part of the array, since the array is sorted. Because the bandwidth sum of the two parts is more or less equal, we can lower the CPU reduction factor for tasks in the left part if we increase the reduction factor for tasks in the right part of the array by the same amount. This allows us to reduce the CPU time of tasks with higher CXL bandwidth more than tasks with only marginal CXL bandwidth. In total, we still reduce the CXL bandwidth by about the same amount as if we would reduce the CPU time of all tasks by the same factor. For this, it is important that the sum of the left and the right arrays is as similar as possible. If the sums of the two parts have a big difference, increasing the CPU reduction factor of one side does not have the same effect on the CXL bandwidth as decreasing the factor by the same amount for the other side.

Instead of using a fixed factor to treat the two parts differently, we use the difference between the averages of the two parts as a fraction ($1 - \frac{\text{avg(left)}}{\text{avg(right)}}$) to determine how much we change the general reduction factor for the two parts. This way, if the two parts have a similar average bandwidth, the difference is not that much. However, if the tasks on the left side have considerably less average bandwidth than tasks on the right, the difference of the reduction factor for the two sides is also larger. This ensures that in a workload consisting of tasks with different bandwidth usages, tasks with higher bandwidth have to reduce their CPU time more. We recursively repeat the algorithm for both parts of the array until the left and right part of the array have one item each. By doing so, we set different reduction factors for tasks depending on

their bandwidth while ensuring that the overall bandwidth is reduced by the amount needed to avoid the overload.

Figure 3 shows an example of the algorithm for 3 processes. The bandwidths of the processes are 0.5, 1.2, and 3.3 GiB/s, respectively. In sum, the processes have a bandwidth of 5 GiB/s, which is 1 GiB/s more than the allowed bandwidth threshold of 4 GiB/s, resulting in a general reduction factor of 0.2. The balance point index splits the array into two parts, with the left part containing the processes with a bandwidth of 0.5 and 1.2 GiB/s, and the right part containing the process with a bandwidth of 3.3 GiB/s. The mean difference factor between the two parts is computed as 0.7424, so that the reduction factor of the left sides becomes 0.0515, while it becomes 0.3485 for the right side. This shows how the algorithm increases the reduction factor for processes with more CXL bandwidth while it decreases it for processes with less CXL bandwidth. The left part is split again, following the same principle, while the right part already only has one element. Eventually, the individual reduction factor for each process is determined. By taking the inverse, this leads to the new CPU limit for each process. For brevity, multiplying this value with the current CPU usage of each process is omitted here, and we assume a previous CPU usage of 100 %. The figure shows that the higher the CXL bandwidth usage of a process, the lower its new CPU limit. Tasks with low CXL bandwidth usage have to sacrifice less CPU time, which is a difference to calculating the CPU limit based on the total CXL bandwidth (§ 4.3.3.1). As we consider the CPU usage and CXL bandwidth usage to be proportional, the same rule also applies to the CXL bandwidth. In total, the new resulting total CXL bandwidth is 3.74 GiB/s, which is lower than the optimal bandwidth threshold of 4 GiB/s. This shows that the algorithm only meets the desired bandwidth approximately, which is caused by the natural inequality of the bandwidth sum of the left and right array parts.

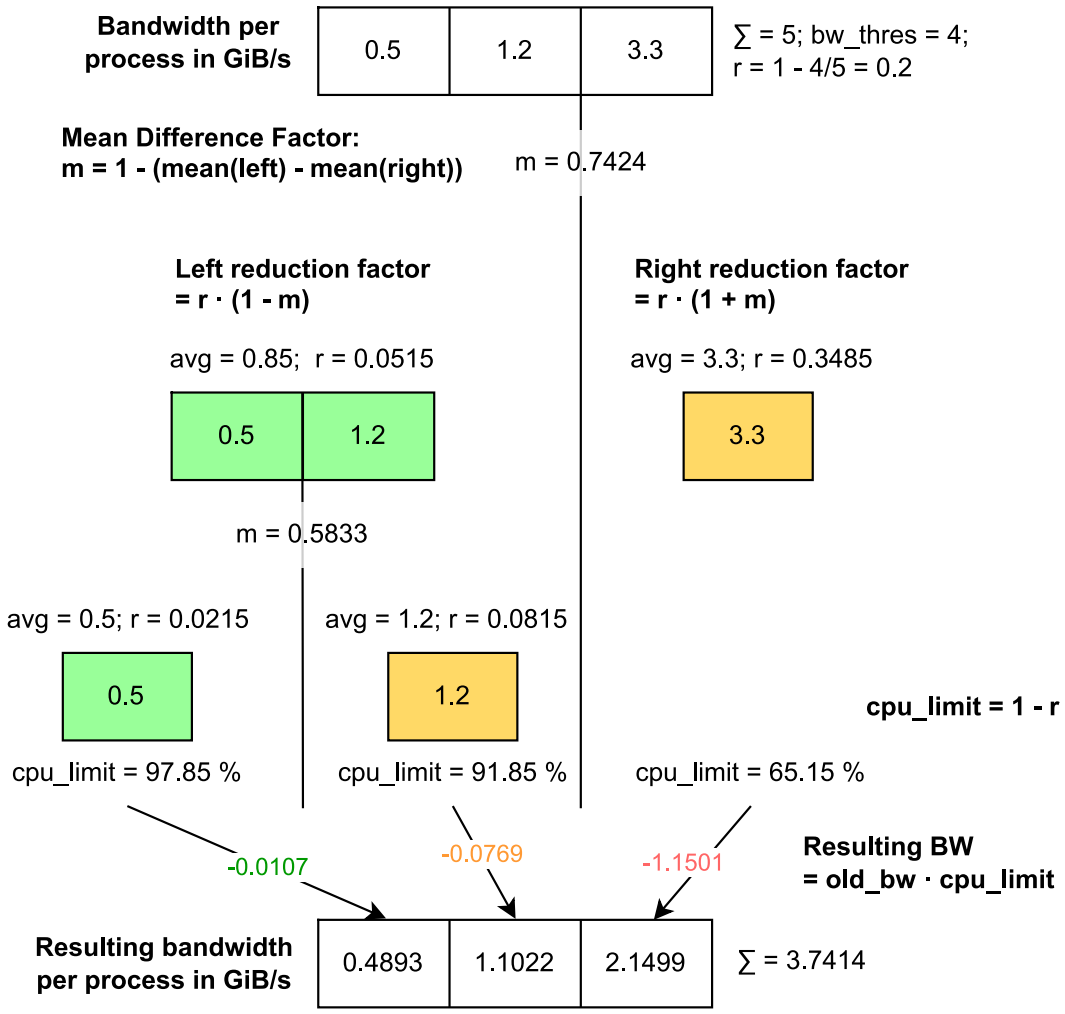


Figure 3: Schema of the algorithm used to determine the CPU limit of tasks by their individual CXL bandwidth.

Chapter 5

Implementation

After having provided an overview of our design in Chapter 4, we now cover the actual implementation of our approach. We start with explaining how we add memory access counters on the CXL FPGA (§ 5.1). Based on this, we present modifications to the kernel page allocator necessary to avoid aliasing between monitoring groups (§ 5.2). Then, we describe how to use PEBS-based sampling to estimate the CXL bandwidth of individual processes (§ 5.3). Finally, we cover the CXL bandwidth control implementations (§ 5.4).

5.1 CXL FPGA Bandwidth Monitoring

To enable the FPGA-based CXL device to distinguish between accesses to different monitoring groups, it needs to expose multiple virtual copies of its physical address space. Depending on the copy that is accessed, the access can then be attributed to the corresponding memory access counter. For the implementation, we use an Intel/Altera Agilex 7 I-Series FPGA [40].

5.1.1 Exposing CXL Memory Space Copies

To expose multiple virtual copies of the CXL FPGA memory, it announces a memory size n times larger than its actual physical memory address space. A CXL device uses Control and Status Registers (CSRs) to communicate its capabilities and metadata to the host OS. The CSRs are implemented as a PCIe configuration space using Base Address Registers (BARs). Specifically, the CXL Range Registers [15:§8.1.3.8] of the PCIe Designated Vendor-Specific Extended Capability (DVSEC) [15:§8.1.3] inform the OS about the size of the CXL memory region. To expose n copies, we

multiply the physical address size of the FPGA times n and assign this value to the `Memory_Size_High` [15:§8.1.3.8.1] and `Memory_Size_Low` [15:§8.1.3.8.2] fields of the CSRs.

For example, exposing 4 copies of the 16 GB memory from our CXL FPGA by setting $n = 4$, followed by compiling and flashing the FPGA as well as rebooting the computer results in the OS reporting 64 GB of CXL memory, as shown in Listing 1.

```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39
node 0 size: 128495 MB
node 0 free: 117186 MB
node 1 cpus:
node 1 size: 65536 MB
node 1 free: 65534 MB
node distances:
node   0   1
  0:  10  14
  1:  14  10
```

Listing 1: Exposing 4 copies from the CXL FPGA results in the OS reporting $4 \cdot 16\ \text{GB} = 64\ \text{GB}$ of CXL memory.

We experimentally determined that our Intel/Altera Agilex 7 I-Series FPGA [40] can handle 512 copies, each with its own read and write counter. However, for 1024 copies, the Intel Quartus FPGA compiler fails to route the design.

5.1.2 Host-Physical to Device-Physical Memory Address Mapping

Due to CXL FPGA announcing a larger address space than its underlying physical address space, the addresses of memory requests sent by the host need to be mapped back to the memory physically available on the CXL device. In the host, these virtual copies of the CXL memory representing monitoring groups are abstracted as NUMA nodes. If the host now issues a request for data at offset 0x42 inside the second NUMA node starting at address 0x2000, the resulting host-physical address is 0x2042. The CXL FPGA has to extract the number of the monitoring group from this address to account the access to the correct counter. Additionally, the FPGA has to extract the offset inside the monitoring group and use it to access its memory. This mapping from host-physical to device-physical address space is also shown

in Figure 4. To implement this mapping, the two parts are extracted from the device-physical address using bitshifts.

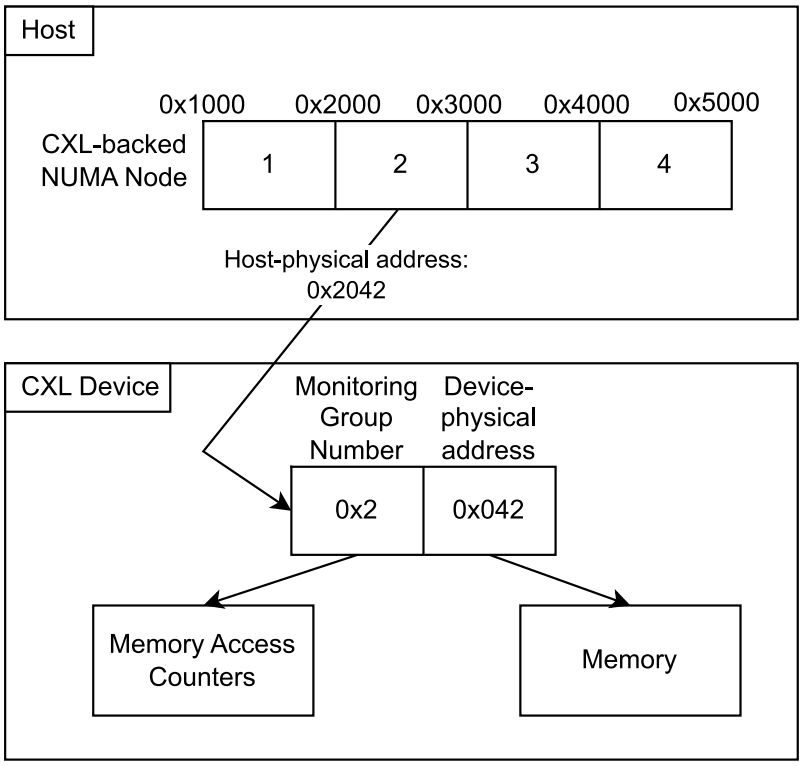


Figure 4: Mapping from a host-physical address device-physical address used to access the memory on the CXL device, as well as extracting the monitoring group number for accounting the access in the corresponding memory access counter.

5.1.3 CXL Memory Access Counters

To count read and write memory requests on the CXL FPGA, we create a new class responsible for managing the CXL memory access counters. We create separate arrays of registers for read and write counters, both with a length matching the number of counters configured. The counter class has an input for the accessed memory address as well as an increment enable signal that signals if the address is valid and the counter corresponding to this address should be incremented. The inputs are then connected to the existing memory request handling logic for reads and writes, respectively. If an address is accessed, we determine the index of the

monitoring group by removing the lower bits containing the memory offset inside the copy by shifting. This is part of the host-physical to device-physical address mapping already shown in Figure 4. The number of bits to shift depends on the number of monitoring groups and the number of counters. Usually, the number of counters and the number of monitoring groups are equal, so there is a counter pair for each copy. However, allowing the variables to be set individually allows to have multiple counter pairs for one copy. This enables us to split the existing device-physical address space into multiple regions without using the virtual copies feature. We use this to test our counting logic, based on the findings in Section 5.2.

To expose the counter values to the host, we modify the class responsible for handling the CXL Control and Status Registers (CSRs). When the host reads from the user-defined part of BAR0 starting at address `0x18100`, we respond with the value of the corresponding memory access counter by connecting its registers with a memory-mapped bus. To determine the index of the counter, we use the accessed address and shift it right by 4. This has the same effect as dividing by 16. These 16 bytes are split into the read and write counter, each having 8 bytes. We determine if the read or write counter is requested by checking if the value of the lower 3 bits is 0 or 8. For the host, the counters are thus laid out in memory according to the following pattern: `read#0 write#0 read#1 write#1 read#2 write#3...`.

5.1.4 Counter Retrieval Kernel Module

To access the counter values in the OS, we implement a Loadable Kernel Module (LKM), as described in Section 3.4. When the module is loaded into the kernel, it obtains a handle to the PCI device using the corresponding vendor and device IDs, enables the device, and retrieves the addresses of its BARs. The BAR is then mapped as an MMIO region into the kernel virtual memory space. Subsequently, it exposes the counter values to the rest of the system in two ways.

The module declares a function, which can be used to retrieve the read or write counter value for a specific NUMA node id (nid). This function calculates the address for the given counter and performs an MMIO read via `readq()` [119]. Our function is also marked as a BPF function, so it can be called by eBPF programs. This is later used by our eBPF-based `sched_ext` CXL bandwidth control scheduler (§ 5.4.1).

The module also exposes the counter values via a debugfs [17] entry located at `/sys/kernel/debug/cxl-counter-driver/counter-dump`, making the counter

values available to user space. For this, it creates a debugfs directory and a debugfs file as well as registers a callback function that is invoked when the file is read. It retrieves the counter values for all monitoring groups using the function described previously and responds with values formatted as ASCII text. An example output of reading this file is shown in Listing 2.

```
read write
7467077175424 100640100736
714993795584 45960215552
```

Listing 2: Example output of reading `/sys/kernel/debug/cxl-counter-driver/counter-dump`, listing the read and write counter values for two CXL-backed NUMA nodes.

5.1.5 Virtual NUMA Nodes for CXL Copies via ACPI Override

By default, a CXL memory device is exposed as a CPU-less memory-only NUMA node, as shown by `numactl --hardware` in Listing 3. We find this to be a suitable abstraction for CXL because of the similar access latency compared to a memory access to a remote socket, which is usually abstracted as a NUMA node [20].

```
$ numactl --hardware
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39
node 0 size: 128495 MB
node 0 free: 117186 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 16384 MB
node distances:
node   0   1
  0:  10  14
  1:  14  10
```

Listing 3: The CXL memory region exposed by the CXL-FPGA, as reported by `sudo cat /proc/iomem`.

Since our CXL FPGA exposes multiple virtual copies of its address space, each with its own access counter pair, we want to abstract each of these virtual CXL devices as one NUMA node. This allows us to bind processes to the memory of a specific NUMA node, causing all their CXL memory accesses to be accounted to one memory access

counter pair. For our CXL bandwidth control, the memory binding of a process is used to obtain the hardware memory access counter values from the corresponding monitoring group.

To create virtual NUMA nodes, we experimented with the `numa=fake` option [54]. However, this only splits the System RAM into n virtual nodes, but not the CXL memory. Alternatively, the page "Upgrading ACPI tables via initrd" [55] in the Linux documentation describes how to override the Advanced Configuration and Power Interface (ACPI) data that describes the hardware to the OS [115]. This contains information regarding *proximity domains*, which Linux uses to create NUMA nodes accordingly. Consequently, we split the one proximity domain corresponding to the memory region exposed by our CXL FPGA and into multiple proximity domains with smaller memory regions. Following the instructions from the kernel documentation [55], first a dump of the ACPI data is created by using `acpidump` [91], which is then extracted into individual files by `acpixtract` [90], and then the files are disassembled via `iasl` [89]. To cause the OS to create virtual NUMA nodes, the ACPI tables have to be modified as described in the following sections. We refer to the ACPI documentation regarding the "ACPI System Description Tables" for more information [115].

5.1.5.1 System Resource Affinity Table (SRAT)

The `srat.dsl` file contains the *System Resource Affinity Table*. An excerpt of one entry from the `srat.dsl` file is shown in Listing 4. The entry declares the proximity domain number as well as the `Base Address` and the `Length`. To split the one large memory area exposed by the FPGA into multiple virtual proximity domains, we declare multiple *Memory Affinity* entries, each with a size of 0x400000000 (i.e., 16 GB) and adjusted base addresses. The number of these entries needs to correspond to the number of virtual copies by the CXL FPGA.

```
[7830h 30768   1]                Subtable Type : 01 [Memory Affinity]
[7831h 30769   1]                       Length : 28

[7832h 30770   4]             Proximity Domain : 00000001
[7836h 30774   2]                    Reserved1 : 0000
[7838h 30776   8]                 Base Address : 0000002080000000
[7840h 30784   8]               Address Length : 0000004000000000
[7848h 30792   4]                    Reserved2 : 00000000
[784Ch 30796   4]        Flags (decoded below) : 00000001
                                       Enabled : 1
                                  Hot Pluggable : 0
                                   Non-Volatile : 0
[7850h 30800   8]                    Reserved3 : 0000000000000000
```

Listing 4: Excerpt of one *Memory Affinity* entry from the `srat.dsl` file.

5.1.5.2 Heterogeneous Memory Attributes Table (HMAT)

The `hmat.dsl` file contains the *Heterogeneous Memory Attributes Table*, which specifies access latencies to proximity domains. The `Data Type` field specifies the property, while the `Entry` field is the value multiplied by the `Entry Base Unit`, which works as a scaling factor, in this case 0x64 = 100. An excerpt of one entry from the `hmat.dsl` file is shown in Listing 5. The `Data Type` 0x04 indicates bandwidth information. The first `Entry` specifies the bandwidth from the implicit initiator proximity domain 0 to the target proximity domain 0 to be $0\text{xA3D} \cdot 0\text{x64} = 262100\ \text{MiB}/s$.

To incorporate additional proximity domains, we copy the access latency and bandwidth information of proximity domain 0 as initiator and proximity domain 1 as target for our n virtual proximity domain copies. While doing so, we need to update the `Length` field.

```
[10Ch 0268   2]               Structure Type : 0001 [System Locality Latency and
Bandwidth Information]
// ...
[110h 0272   4]                      Length : 00000036
// ...
[115h 0277   1]                   Data Type : 04
// ...
[124h 0292   8]              Entry Base Unit : 0000000000000064
[12Ch 0300   4] Initiator Proximity Domain List : 00000000
[130h 0304   4] Target Proximity Domain List : 00000000
[134h 0308   4] Target Proximity Domain List : 00000001
[138h 0312   4] Target Proximity Domain List : 00000001
[13Ch 0316   2]                        Entry : 0A3D
[13Eh 0318   2]                        Entry : 012C
```

Listing 5: Excerpt of one *System Locality Latency and Bandwidth Information* from the `hmat.dsl` file.

5.1.5.3 System Locality Information Table (SLIT)

The `slit.dsl` file contains the *System Locality Information Table*, which is an $n \times n$ matrix listing the distance between all combinations of nodes. Note that the distance from one node to itself (the diagonal) needs to be set to $10 = 0\text{x}0\text{A}$, as described in the ACPI documentation [115:§5.2.17]. For a modification, the `Table Length` field at the beginning of the file and the `Localities` field specifying the number of localities need to be updated accordingly. Additionally, a `Locality` row needs to be added and an entry for each `Locality`, which also requires the `byte length` at the beginning of each line to be adjusted.

```
[024h 0036   8]                  Localities : 0000000000000005
[02Ch 0044   5]                Locality   0 : 0A 0E 0E 0E 0E
[031h 0049   5]                Locality   1 : 0E 0A 0E 0E 0E
[036h 0054   5]                Locality   2 : 0E 0E 0A 0E 0E
[03Bh 0059   5]                Locality   3 : 0E 0E 0E 0A 0E
[040h 0064   5]                Locality   4 : 0E 0E 0E 0E 0A
```

Listing 6: An excerpt of the modified *System Locality Information Table* from `slit.dsl` for 4 fake NUMA nodes.

5.1.5.4 Custom initramfs/initrd

Modifying the ACPI files causes the offsets specified in the files to be wrong. We develop a Python script that recalculates the correct offsets. Note that this does not

fix the `Length` field of individual entries nor the `Table Length` field at the beginning of each file. These need to be updated manually.

After the modifications, the changed files are compiled to `.aml` files using `iasl` [89]. A `.cpio` archive containing the modified ACPI files is created by the tool `cpio` [67]. Finally, the original initramfs/initrd for the corresponding kernel version is appended to the `.cpio` archive. To use the custom initramfs/initrd, a custom GRUB entry referencing the custom initramfs/initrd file needs to be created and selected upon boot.

If the kernel is booted with the CXL FPGA exposing multiple copies of its physical memory, the kernel crashes with a general protection fault during boot, likely because different NUMA nodes access the same device-physical addresses and mutually corrupt their memory and metadata stored inside it. Consequently, the CXL FPGA is operated as a normal CXL memory expander, and the ACPI information is used to split its region into multiple smaller regions, each corresponding to a virtual NUMA node. We present a potential fix for the memory aliasing problem in Section 5.2.

The modified ACPI information with the virtual proximity domains causes the Linux kernel to create virtual NUMA nodes accordingly, as shown in Listing 7. Compare this to the single NUMA node for the CXL memory in Listing 3.

```
$ numactl --hardware
available: 5 nodes (0-4)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39
node 0 size: 128412 MB
node 0 free: 124880 MB
node 1 cpus:
node 1 size: 4096 MB
node 1 free: 3902 MB
node 2 cpus:
node 2 size: 4096 MB
node 2 free: 4094 MB
node 3 cpus:
node 3 size: 4096 MB
node 3 free: 4094 MB
node 4 cpus:
node 4 size: 4096 MB
node 4 free: 4093 MB
node distances:
node   0   1   2   3   4
  0:  10  14  14  14  14
  1:  14  10  14  14  14
  2:  14  14  10  14  14
  3:  14  14  14  10  14
  4:  14  14  14  14  10
```

Listing 7: (Virtual) NUMA topology showing 4 CXL-backed NUMA nodes besides the native NUMA node 0, as discovered by the kernel.

5.2 Patching the Kernel Page Allocator

As already mentioned, if processes access CXL memory via two different NUMA nodes, they actually work on the same device-physical address on the CXL FPGA if they use the same offset inside their NUMA node memory region. Figure 5 shows how the pages of different NUMA nodes are mapped to the same page on the CXL FPGA. The kernel page allocator is not aware that different host-physical addresses are aliases of each other on the CXL FPGA. This causes the allocator to hand out aliased pages, leading to access conflicts in the form of data corruption.

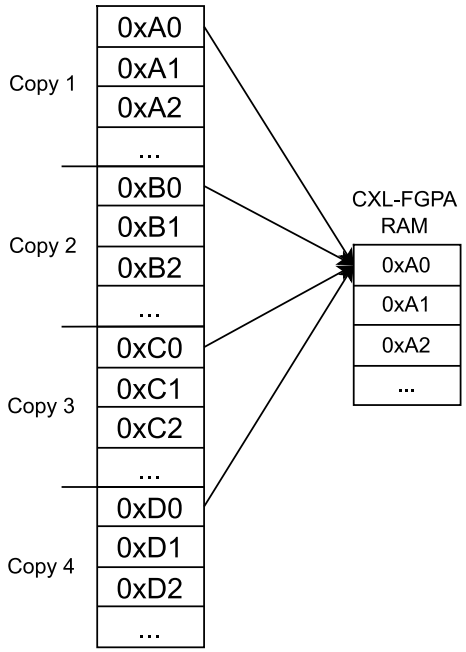


Figure 5: Aliasing of memory addresses. Memory accesses to the same offset in each NUMA node result in the same device-physical address on the CXL FPGA.

Operating the CXL FPGA as a normal CXL memory expander and splitting its actual memory region into multiple smaller regions via modified ACPI information causes virtual NUMA nodes to be created whose memory regions do not overlap on the device-physical memory. This avoids the aliasing problem, but each NUMA node only has one-nth of the total memory size of the CXL FPGA. Since we use the NUMA node as an ID for the monitoring group of the CXL memory access counters on the CXL FPGA, an application can only access memory from one virtual CXL NUMA node. This limits the amount of memory one application can use to one-nth of the original size of the CXL FPGA. This leads to a trade-off: On the one hand, more memory regions and thus more NUMA nodes mean better performance isolation for more applications. On the other hand, fewer memory regions mean more total space per NUMA node and thus per application.

Instead of splitting the memory into equally-sized parts, it could also be split in parts of different sizes, so that applications can use a NUMA node that fits their memory needs, leaving larger nodes for more memory-demanding applications. For example,

inspired by the geometric sequence $\sum_{i=1}^{n} \left(\frac{1}{2}\right)^i$, the memory could be split into parts of $\left(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right)$ the size of the total memory. However, this approach is still static in that the memory splitting has to be decided upon at boot time of the kernel, and applications are bound to a particular NUMA node when spawning.

For dynamic memory splitting, we patch the Linux kernel page allocator located in `linux/mm/page_alloc.c`. The page allocator is responsible for the virtual-to-physical address mapping on a *page-level granularity* [59]. User space applications may additionally use their own allocator for heap allocations. The approach for modifying the page allocator is to redirect all page allocations targeting virtual CXL NUMA nodes (i.e., nid > 1) to the first virtual CXL-backed NUMA node with the nid 1, which we call the *allocation node*. After the allocator chose a page and a corresponding page frame number (PFN), the corresponding page with the same offset in the initially requested node is computed, and a corresponding `struct page` is returned. This avoids aliasing by ensuring that one device-physical page is not allocated to multiple nodes. Similarly, the free function performs the inverse mapping to free the page on the allocation node with nid 1. Figure 6 visualizes the redirection of the NUMA node ID of the page allocator.

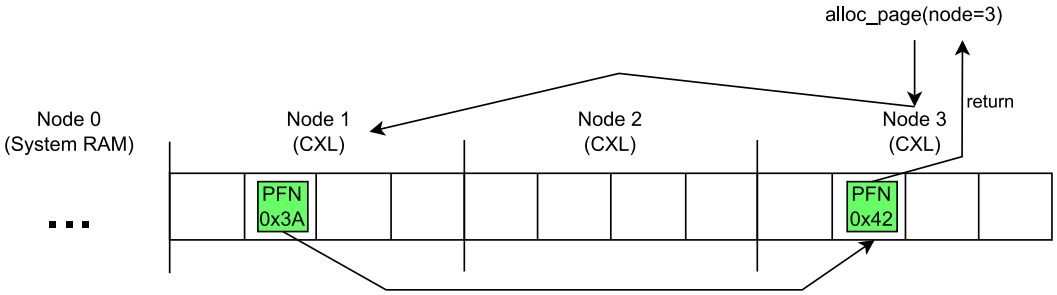


Figure 6: The patched kernel page allocator. An allocation with preferred node id 3 is redirected to node 1. The allocator finds a page with a page frame number (PFN) of 0x3A to be free. The patched allocator calculates the PFN for the initially requested node and returns a corresponding page.

Specifically, we determined `__alloc_pages_noprof()` to be central to the allocator. If it is called with a preferred NUMA node ID (NID) of one of our virtual CXL-backed NUMA nodes, we first kick the initialization of the preferred nid by calling `prepare_alloc_pages()` for this nid because the node might not be initialized

yet during boot. Then, we use the normal allocator to allocate a page on the allocation node (i.e. NUMA node 1). To obtain the corresponding PFN on the preferred nid, we use `page_to_pfn(page) - node_start_pfn(page_to_nid(page))` to compute the PFN offset inside the allocation node, followed by adding `node_start_pfn(preferred_nid)` to get a PFN inside the preferred nid. After ensuring that the calculated PFN is valid using `pfn_valid()` and that the calculation actually returned a page inside the bounds of the preferred nid, we call `prep_new_page()`, for example, to increment the reference count of the page and unpoison the page. We do the reverse mapping in `free_unref_page()` to free the page on the allocation node where it was also allocated. Without this, the allocator would attempt to free the page on the preferred nid, where it was never officially allocated.

The allocator patch can be enabled via the kernel configuration option `CXL_ALIASING_ALLOCATOR_SUPPORT`. With the patch enabled, we found the kernel to panic during boot with list corruption errors, general protection faults, or CPU soft lockups, indicating that the patch is not fully correct. Because of the memory corruption caused by aliased memory accesses, kernel panics can occur at completely different spots in the kernel code, making it hard to find bugs or loopholes in the implementation of the page allocator patch. We assume that our patch misses some alternative entry/exit points to the allocator. For example, we found `alloc_pages_bulk_noprof()` to circumvent the standard `__alloc_pages_noprof()` by default and only use it as a fallback. We patched it to always fall back to `__alloc_pages_noprof()`. Unfortunately, we have not found all such occurrences so that the kernel boots and runs stably with the patch. Consequently, we resort to splitting the actual memory of the CXL FPGA into multiple smaller NUMA nodes for testing.

5.3 PEBS-based Memory Access Sampling

Our PEBS-based sampling is based on the project for efficient persistent memory (PMEM) accounting by Habicht [26]. This, in turn, is “partially based on the PMU driver for Intel CPUs found in the Linux kernel” [45], and on `simple-pebs` from the pmu-tools project [62]. The implementation of the PEBS-based sampling consists of a kernel patch and a kernel module responsible for the PEBS-based CXL

bandwidth estimation. These two parts are detailed in Section 5.3.1 and Section 5.3.2, respectively. The original implementation [26] had to be ported to support our Intel Xeon Scalable (4th generation) CPU with the Sapphire Rapids microarchitecture and a Linux kernel version 6.12.

5.3.1 Kernel Patch

Our patch for the Linux kernel in version 6.12 is based on the patch for the Linux kernel in version 5.15 from the efficient PMEM accounting project [26]. It modifies parts of the kernel to enable our PEBS-based bandwidth estimation kernel module (§ 5.3.2). It can be enabled via the Linux config option `EFFICIENT_CXL_ACCOUNTING` [56]. The kernel build infrastructure [56] automatically defines macros for this configuration option, which is used to conditionally compile the code of the patch.

In the following, we give an overview of its modifications. To enable full access to the PMU for our PEBS-based CXL bandwidth estimation module, the patch disables access to the PMU for perf [79]. Without the patch, perf would also have access to the PMU and could modify its configuration, which could disturb and/or stop our accounting. Additionally, the patch adds the fields for the PEBS read and write counters to the `struct task_struct`, which holds the information about a task. These added fields are used by our PEBS-based CXL bandwidth estimation kernel module (§ 5.3.2) to store the PEBS CXL access counter values per task. The patch also adds the procfs file `/proc/PID/cxl_io` to expose the per-task PEBS CXL access counters to user space. If this special file is read, the kernel responds with the values of the access counters stored in the `struct task_struct` of the respective PID. This is similar to the procfs entry `/proc/PID/io` [72], which lists the total (i.e., DRAM and CXL) amount of bytes read and written for a process. Moreover, the patch exposes some kernel functions to our kernel modules that are usually internal to the kernel and deal with setting up a PEBS buffer in the CPU entry area (CEA). Furthermore, it adds functions to register a callback function to be called when the scheduler switches tasks. This is used by the PEBS-based CXL bandwidth estimation kernel module (§ 5.3.2) to process the PEBS buffer before switching tasks so that PEBS samples are accounted to the correct task.

5.3.2 PEBS-based CXL Bandwidth Estimation Kernel Module

The PEBS-based CXL bandwidth estimation kernel module is responsible for using PEBS to sample memory loads and stores of tasks, inspect if the memory accesses

target CXL memory, and increment the corresponding access counter of the task. We use Zydis [127] to disassemble instructions sampled by PEBS. This reveals the memory access size (i.e., 1, 2, 4, or 8 byte). Although the underlying memory accesses to CXL hardware are usually a cache line of 64 bytes[6] [20], we use the instruction's memory access size as a proxy for the consumed bandwidth. For example, if a loop iterates over `uint64_t`, each having 8 bytes, a new cache line needs to be loaded every 8 memory accesses. In contrast, if the loop iterates over `uint8_t`, each having only one byte, a new cache line only needs to be loaded every 64 memory accesses. However, inferring the actual CXL bandwidth from the instruction memory access sizes is only an approximation.

When the module is loaded into the kernel, it first allocates and sets up a debug store for the PEBS records to be saved into [35:§19.4.9] as well as initializes Zydis [127] for instruction decoding. It further installs handlers for when the scheduler switches tasks (as explained in Section 5.3.1) and for when catching non-maskable interrupts (NMI), which are triggered when the PEBS buffer is becoming full and reaches a certain threshold [35:§21.3.1.1.1]. Next, the module configures the PMU and PEBS by writing into specific *model-specific registers* (MSRs). Chapter 21 of Intel Architectures Developer's Manual [35] explains performance monitoring and how to set it up. Specifically, we configure two counters to count loads and stores occurring in userspace, respectively [35:§21.2.3]. For this, we use the events `MEM_INST_RETIRED.ALL_STORES` and `MEM_INST_RETIRED.ALL_LOADS` [19]. We then enable PEBS records to be generated when the counters overflow [35:§21.3.1.1.1]. To achieve the desired sampling interval, we set the counter reset value [35:§21.6.8.2], as explained in Section 3.3.

The kernel module [26] was originally implemented for performance monitoring version 3, but the Intel Sapphire Rappids CPU used in our setup uses version 5 [35:21.3.10]. Version 4 introduced adaptive PEBS [35:§21.9.2], which reduces PEBS records to basic information with configurable additional information [35:§21.9.2.2]. The basic information does not include the accessed memory address, which is crucial for our approach to be able to distinguish between CXL and DRAM accesses and only account CXL accesses. Consequently, we activate adaptive

[6] Non-temporal stores may be an exception.

PEBS [35:§21.9.2.1] and configure it so that memory-related information is included in PEBS records [35:§21.9.2.3].

We process the PEBS records in the PEBS buffer if a non-maskable interrupt (NMI) is generated due to the PEBS buffer becoming full [35:§21.6.8.5] or when our function is invoked due to the scheduler switching tasks (see Section 5.3.1). For each PEBS record, the virtual address is translated to a physical address to determine if the memory access lies in the address region of the CXL memory. For this, a manual page table walk is performed. The instruction belonging to the instruction pointer of the PEBS sample [35:§21.9.2.2.1] is then copied from user space, and Zydis [127] is used to decode the instruction to obtain its memory access size. To account for the sampling interval, i.e., that some thousands of instructions have not been sampled, the memory access size is multiplied by the sampling interval. As the sampling interval, we use the prime number 9973, as suggested by Yi et al. [122]. Using this information, the memory access counter of the corresponding task stored in its `task_struct` is updated.

5.4 CXL Bandwidth Control

The CXL bandwidth control (CBC) is responsible for limiting the total CXL bandwidth to the bandwidth threshold so that the CXL access latency stays low. For this, the CBC reduces the CPU time of tasks. We develop different CBC implementations, which can be split into two main approaches to enforce the CPU limit: Section 5.4.1 describes a custom scheduler based on the eBPF-based `sched_ext` scheduler interface of the kernel, while Section 5.4.2 explains a userspace application that uses control groups (cgroups) to set resource limits. For both implementations, we experimented with various additional features, as explained in the corresponding subsections.

5.4.1 sched_ext-based Scheduler

A `sched_ext`-based scheduler works by registering callback functions on scheduling events [70]. If a task is runnable, the Linux `sched_ext` handler calls the `select_cpu()` callback function. It selects the CPU core where the task should run, and directly dispatches it to this core if it is idling. If the selected CPU is not idling, the `enqueue()` callback function is executed. It is supposed to insert the task into a dispatch queue. Callbacks for various other parts of the scheduler can be registered within

sched_ext, but are out of scope for our implementation. We refer to the documentation of struct sched_ops on ebpf.io [21], the Linux kernel documentation about sched_ext [70], and the Linux source code documentation located in kernel/sched/ext.c for a comprehensive overview.

Our scheduler implementation is a round robin scheduler based on the FIFO example scheduler scx_simple from the SCX project [108]. In the following, we give an overview of our modifications. The main logic of our approach is placed in the enqueue() function. Here, we retrieve the FPGA and PEBS access counters, calculate the bandwidth of the task and its corresponding time slice to reduce its CPU usage if necessary. Obtaining the per-process PEBS access counters is trivial, as they are stored directly inside the task_struct passed into the callback function. Retrieving the CXL FPGA access is more complex as it involves the counter retrieval kernel module (§ 5.1.4), and is thus described in Section 5.4.1.1.

The default implementation of select_cpu() directly dispatches a task to the CPU core if it is idle, skipping the enqueue() callback function. However, we perform the time slice computation for a task in the enqueue() function. Therefore, it is crucial that the enqueue() function is always executed. Consequently, we override the select_cpu() callback function. We delegate the CPU selection to the base implementation by calling scx_bpf_select_cpu_dfl(), but do not directly dispatch the task to the CPU if it is idle. Instead, we only use the "optimization hint" [70] of returning the selected CPU core, causing the core to be woken up if idling [70].

5.4.1.1 Retrieving CXL FPGA Memory Access Counters

To retrieve the CXL FPGA memory access counters of the corresponding monitoring group for the task in the scheduler, we first need to find out what NUMA node a task is memory-bound to. For this, we can inspect the mempolicy field of the task_struct that the kernel passes to the sched_enqueue() function of the sched_ext scheduler. This struct mempolicy has a field called nodes of type nodemask_t. This is a bitmap of all the nodes the task is memory-bound to, so each bit represents if the task is bound to the NUMA node with this bit index. Internally, this bitmap is an array of longs, with a length depending on the number of specified bits. This means that if all bits fit into one long (typically 64 bits), only one long is used, but if more bits are specified (i.e., more NUMA nodes exist), multiple longs are used. In our case, we want to find out if a task is bound to exactly one NUMA node. In the kernel,

there are the functions `nodes_weight()` to find out the number of set bits/NUMA nodes in a `nodemask_t` and `first_node()` to find out the first set NUMA node in the `nodemask_t`. However, these are not exported as a BPF KFunc @[57] and can thus not be used from eBPF. Therefore, we implement our own function in eBPF that checks that only one bit is set in `nodemask_t` and returns its index.

Given the NUMA node the task is memory-bound to, the scheduler needs to retrieve the CXL FPGA memory access counter values from the FPGA counter retrieval kernel module (§ 5.1.4). Since the counter values are exposed to userspace via the debugfs file, as explained in Section 5.1.4, a userspace helper could be used to periodically retrieve the counter values. A BPF map (i.e., an array) shared between the userspace helper and the eBPF-based `sched_ext` scheduler would allow the userspace helper to communicate counter values to the scheduler [21, 100]. When the `sched_ext` scheduler is about to enqueue a task to a dispatch queue, it could retrieve the FPGA counter values from the BPF map to calculate the CXL bandwidth of the task and adjust its time slice accordingly.

However, this approach has the overhead of going from the FPGA counter retrieval kernel module (§ 5.1.4) to the debugfs file to the eBPF userspace helper program to the BPF map to the eBPF-based `sched_ext` scheduler. Retrieving the counter values directly from the counter retrieval module by calling one of its (kernel) functions would avoid this overhead, but requires some extra steps due to the eBPF safety guarantees of eBPF [18]. The KFuncs feature of the Linux kernel enables the option to declare kernel functions as callable by eBPF programs [21, 57]. Thereby, a kernel function needs to be explicitly annotated with `__bpf_kfunc` and a BPF KFunc set needs to be declared that lists this function [57]. Additionally, an instance of the `struct btf_kfunc_id_set` referencing this BPF KFunc set needs to be created and registered using `register_btf_kfunc_id_set()` with the `BPF_PROG_TYPE_STRUCT_OPS` BPF program type on module load [57]. On the side of the eBPF-based `sched_ext` scheduler, the kernel function is declared as an `extern` function prototype together with the `__ksym` suffix annotation so that it is resolved to a kernel symbol [21:KFuncs]. This way, the `sched_ext` scheduler can directly call the function of our CXL FPGA memory access counter retrieval kernel module to retrieve a counter value by passing the counter index and a boolean flag determining if the read or write should be returned.

To optimize the performance of the `enqueue()` function, we do not fetch the memory access counters from the CXL FPGA every time a task needs to be enqueued. Instead, we store an exponential running average (see Section 4.3.1.1) of the bandwidth for all NUMA nodes as well as for all tasks, and use them for computations. Only if a certain, configurable amount of time (we use 1 ms) has elapsed since the last update, we retrieve the counters and update the stored running average.

5.4.1.2 Aggregating PEBS-estimated CXL Bandwidth of a NUMA Node

Although we use Equation 4 to determine the reduction factor of each task in the `sched_ext`-based approach, and thus do not need to know the CXL bandwidth of each task, we implement the retrieval of the corresponding counters and the computation nevertheless so, that new approaches for the equation can be tried out easily.

To combine the PEBS and FPGA bandwidths as shown in Equation 3, we need to find all other tasks that are also memory-bound to the same node as the task we want to schedule to calculate the aggregate PEBS-estimated CXL bandwidth of its NUMA node. One option would be to iterate over all tasks on the system by using the eBPF functions `bpf_iter_task_new()` and `bpf_iter_task_next()` every time a task needs to be scheduled [21]. For this, we would use the approach described in Section 5.4.1.1 to find the NUMA node a task is memory-bound to and sum the bandwidth of the tasks that are memory-bound to the same NUMA node as the task we want to calculate Equation 3 for.

However, iterating all hundreds or thousands of tasks on the system each time a task needs to be enqueued to a dispatch queue in the scheduler stands in contrast with the goal that the scheduler, as a core mechanism of the OS, should be as fast as possible. To avoid this overhead, we store the process IDs (PIDs) that are memory-bound to a NUMA node in a hash set for this NUMA node. An array then contains the hash sets for all NUMA nodes of the system. To realize the hash set, we use a BPF map with type `BPF_MAP_TYPE_HASH` [21]. To store the array of hash sets, we use a BPF map of type `BPF_MAP_TYPE_ARRAY_OF_MAPS` [21]. Due to limitations of eBPF and its verifier, the user space scheduler manager needs to initialize the entries of the array by creating a map of `BPF_MAP_TYPE_HASH` and inserting it into the map for each entry before attaching the scheduler to the kernel, as also described later in Section 5.4.1.6. When a task is about to get enqueued, its memory binding is retrieved as described in Section 5.4.1.1. It is then ensured that the task is part of the

hash set of the respective NUMA node. This means that if the PID of the task is not already in the hash set because the task was just created and is enqueued for the first time, it is inserted. Due to the usage of a hash set, the lookup and insertion can be accomplished in $O(1)$ on average. Consequently, if the hash set is iterated later to sum the bandwidth of its tasks (potentially when a different task is enqueued), this task is part of the hash set and is included in the sum. If retrieving the task struct of a PID in the hash set fails when calculating the bandwidth sum, this task does not exist anymore because it exited, and its PID is removed from the hash set.

Since eBPF runs inside the kernel, it cannot use floating-point arithmetic [94]. Representing the byte count delta Δc as an integer from Equation 3 is fine, but it is problematic to store the timestamp delta Δt in integer seconds. It may work for the `top` program, which is only scheduled every 3 seconds by default, but compute-heavy applications in a decently utilized system typically experience sub-second scheduling time slice [112:§2.4.3], resulting in an integer time delta of 0, leading to a division by zero. Unfortunately, however, exactly these applications are usually bandwidth-intensive and are thus the focus of this work. As a fix, we store the timestamps t_1 and t_2 as well as Δt in nanoseconds (ns) instead (denoted as Δt_n in the following). Using an `uint64_t` is sufficient, as it does not overflow in a reasonable time period, as shown in Equation 5.

$$\begin{aligned} \text{u64\_max} &= 2^{64} - 1 \\ \text{ns\_per\_year} &= 10^9 \cdot 60 \cdot 60 \cdot 24 \cdot 365 \\ \frac{\text{u64\_max}}{\text{ns\_per\_year}} &\approx 584.9 \text{ years} \end{aligned} \tag{5}$$

The order of operations when calculating the bandwidth is also important. A naive approach to calculate the bandwidth with a nanosecond time delta would be:

$$b = \frac{\Delta c}{\Delta t_n} \cdot 10^9 \frac{\text{ns}}{\text{sec}} \tag{6}$$

However, consider a task that has a bandwidth of 900 MB/s (i.e., $\Delta c = 900 \cdot 10^6$) but is only rescheduled every second (i.e., $\Delta t_n = 10^9$ ns). Calculating the bandwidth for this task would result in 0 because $\Delta c / \Delta t_n = 0.9$, which cannot be represented using an integer and is thus rounded down to 0.

To avoid the limitations of integers, the formula can instead be rearranged to:

$$b = \frac{\Delta c \cdot 10^9 \frac{\text{nsec}}{\text{s}}}{\Delta t_n} \tag{7}$$

This circumvents intermediate values smaller than 0. In contrast, this formula tends to have large intermediate values and can thus cause integer overflows. For example, if the bandwidth is $100\frac{\text{GB}}{s} \cdot 10^9 \frac{\text{B}}{\text{GB}}$ and the program is scheduled only every second (i.e. $\Delta t_n = 10^9$ ns), the calculation of $\Delta c \cdot 10^9$ results in the value 10^{20}, which exceeds the maximum value of a `uint64_t` being $2^{64} - 1 \approx 1.8 \cdot 10^{19}$. Consequently, we do all calculations in kilobytes. Tasks with a bandwidth of under 1 KiB/s are rounded down to 0, but are negligible anyway.

5.4.1.3 Time Slice Determination

To avoid overload situations and high CXL access latencies, we control the scheduling time slice $s(t)$ of a task t. For this, we compute the CPU limit as described in Section 4.3.3.1 and scale the base time slice (s_0) accordingly, as shown in Equation 4.

$$s(t) = s_0 \cdot \text{cpu\_limit} \tag{8}$$

For the actual implementation, we again have to consider that eBPF does not support floating-point arithmetic. Thus, we scale up all numerators in fractions by multiplying them with a factor before the division, and only scale down the result at the end by dividing through the same factor. This way, all intermediate values are larger than 0, ensuring they can be computed using integers.

We do not attempt to implement the algorithm for determining the CPU limit based on the individual task CXL bandwidth(§ 4.3.3.2) in `sched_ext` because of the limitations of the eBPF verifier and the missing floating-point arithmetic support.

5.4.1.4 Virtual Runtime Scheduling

A round robin scheduler does not consider scheduling weights assigned to tasks, which is critical for timely servicing high-priority tasks, such as kernel workers. In our scenario, the reduction of scheduling time slices may also not be sufficient to reduce the CXL bandwidth of the system back to the target bandwidth threshold. To address these issues, we add virtual runtime scheduling to our `sched_ext`-based scheduler, which can be enabled via conditional compilation.

In runtime-based scheduling, the sum of the used CPU time is accounted per task. When selecting the next task to run, the scheduler selects the task that has received the least CPU time so far, thereby ensuring fairness among tasks [5:§9.7, 48]. As an extension, the time can pass at different speeds for different tasks, which is called *virtual* runtime. This is implemented by scaling the actually elapsed CPU time by the inverse of the scheduling weight of a task, which can be controlled using the `nice` value, for example [73]. As a result, a task with a higher weight has a lower virtual runtime in comparison to other tasks, causing the scheduler to prioritize this task.

To implement vtime scheduling in `sched_ext`, we extend our round robin scheduler by registering additional callback functions, similar to how it is done in the `scx_simple` example scheduler of the SCX project [108]. When a task is added to our scheduler, the `enable()` callback sets its initial vtime to the global vtime. This is the maximum vtime of all tasks and is updated by the `running()` callback each time a task starts running. When a task stops running, `stopping()` is called, which updates the vtime of this task by adding its used CPU time, scaled by the inverse of the scheduling weight of the task. By enqueuing tasks into a priority dispatch queue ordered by their vtime, tasks with a lower vtime are prioritized over tasks with a higher vtime. To avoid that tasks can accumulate a large deficit of vtime, and thus get an exceptionally high priority, we limit the vtime deficit to one time slice.

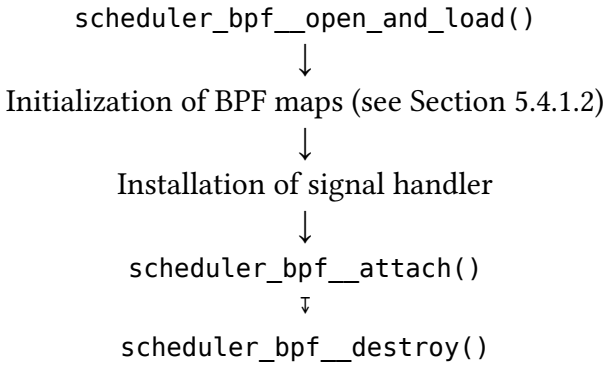
We add to the vtime concept by additionally scaling the runtime of a CXL-using task by the inverse of the CXL bandwidth. Specifically, we use the CPU limit stored in the task context, which is determined by the `enqueue()` function. By combining the vtime-based priority reduction of CXL bandwidth-intensive tasks with the reduction of their time slices, we seek to control the resulting CXL bandwidth more effectively.

5.4.1.5 Separate Thorttling Disptatch Queue

As an alternative to the vtime-based scheduling, we modify the scheduler so that it moves tasks it wants to throttle into a separate dispatch queue. We call this the *throttling dispatch queue*. When the `dispatch()` callback function is called, it first tries to consume a task from the normal dispatch queue. If this queue is empty and there is no CXL bandwidth overload situation, it also considers tasks from the throttling dispatch queue. This results in throttled tasks getting parked by the scheduler in this separate dispatch queue until the overload situation has ended.

5.4.1.6 Compilation and Userspace Scheduler Manager Lifecycle

We use the libbpf library [66] as an abstraction for the raw `bpf()` system call [68]. ebpf.io [21] and the man page about bpf-helpers [69] provide documentation for libbpf. The eBPF-based `sched_ext` scheduler can be compiled using the clang compiler, specifying `bpf` as the target. A header file is generated from the `.bpf.o` file for the userspace scheduler manager, allowing it to use libbpf to load and attach the scheduler to the kernel, as shown in Listing 8. The userspace scheduler manager, responsible for initializing BPF maps, can then be compiled by using `clang`, passing the kernel headers and the `sched_ext` specific header files while linking libbpf.



Listing 8: Workflow of the userspace scheduler manager using libbpf and `sched_ext` lifecycle. Note that the prefix `scheduler` is based on the file name of the `sched_ext` program and can thus be different.

5.4.2 cgroups-based Maximum CPU Usage Throttling

The control groups (cgroups), specifically cgroups v2, is feature of the Linux kernel that allows to group processes and set resource usage limits for this group [51, 74]. In particular, this can be used to limit the amount of CPU time a cgroup may use via the `cpu.max` attribute [46]. It consists of `max` and `period` values, specifying how many microseconds (µs) a cgroup is allowed to use the CPU in a period. If the quota is exhausted, the cgroup is not scheduled until the `period` has elapsed, which resets the used quota. The Linux CFS/EEVDF scheduler takes care of enforcing these limits [46]. Listing 9 shows the workflow of creating a cgroup, adding a PID to it, and setting CPU resource limits.

Create cgroup for PID, if not existing:
`sudo mkdir -p cbc_15011`
↓
Assign PID to cgroup:
`echo 15011 > cbc_15011/cgroup.procs`
↓
Set CPU max in a period, both in microseconds (µs):
`echo "4200 10000" | sudo tee cbc_15011/cpu.max`

Listing 9: Workflow of setting a CPU limit to 42 % via the cgroups filesystem interface for the example PID 15011.

For the implementation, we wrote a userspace program in the Rust programming language that periodically reads the FPGA and PEBS-based access counter values, determines the per-process CXL bandwidth, calculates the corresponding CPU limits accordingly (§ 4.3.3.2)), and throttles tasks using the `cpu.max` cgroups feature.

In each loop iteration, the `/proc/` directory is enumerated to find new PIDs. For each new PID, the memory binding is inspected by reading `/proc/PID/numa_maps` and parsing it for listed NUMA nodes. The PID, together with its memory binding, is then stored in a hash map to avoid the overhead of repeated lookups of the memory binding in the procfs. Subsequently, the PEBS-estimated CXL access counters are obtained for each process that has a memory binding by reading the `/proc/PID/cxl_io` file (§ 5.3.2). Similarly, the per-NUMA node access counters from the FPGA are retrieved by reading `/sys/kernel/debug/cxl-counter-driver/counter-dump`, which is exposed by our counter retrieval kernel module (§ 5.1.4). Using the old counter values and their timestamps, the bandwidth of each task is calculated as a combination of the PEBS and FPGA counters, as described in Section 4.3.1. Additionally, the CPU usage percent of the task is obtained by reading `/proc/PID/schedstat`. Based on a configuration option, the CPU limit for a task is either calculated based on the total CXL bandwidth (§ 4.3.3.1), which refer to as "simple cgroups-based CBC" or based on the individual CXL bandwidth of the task (§ 4.3.3.2). To enforce the CPU limit, a cgroup is created and the process is added to it as described in Listing 9.

When setting the CPU limit, cgroups enforces a minimum value of 1000 µs for both `max` and `period` [46]. Supporting a CPU limit of 1 % as the lowest value means that the `period` would need to be 100 000 µs. Instead of calculating the allowed `max` in a fixed

`period`, we use the reciprocal value of the allowed CPU percentage to calculate the `period`, as shown in Equation 9. This way, `max` is always set to 1000 µs and is thus as small as possible, while `period` grows for small CPU percentage limits, resulting in the best CPU limit precision possible. To avoid resetting the already used CPU time of the current period for a given process by overwriting its CPU limit, we execute the CBC logic every 150 ms only.

$$\begin{aligned} \text{max} &= 1000\ \mu\text{s} \\ \text{period} &= \text{max} \cdot \frac{1}{\text{cpu\_limit}} \end{aligned} \tag{9}$$

Chapter 6

Evaluation

Based on the implementation, we now evaluate our approach. Before starting with the main evaluation, we describe our evaluation setup (§ 6.1). Then, we first verify that our CXL FPGA and PEBS-based memory bandwidth estimation work correctly (§ 6.2). Subsequently, we compare how effective the different `sched_ext`-based and cgroups-based CXL bandwidth limitation implementations are in meeting the target bandwidth (§ 6.2). Table 1 shows an overview of how we refer to the implementations with different features enabled. To go into the detail of different CBC approaches, we inspect how they set CPU time limits (§ 6.4). We then investigate the impact of the CXL bandwidth limiting on the CXL memory access latency (§ 6.5). Further, we analyze the CPU cost of accessing CXL memory (§ 6.6) as well as the CPU throughput for tasks not using CXL on the system (§ 6.7). Finally, we measure the overhead of the CXL bandwidth implementations (§ 6.8).

For the evaluation, we use Python with the libraries matplotlib [86] to generate plots, scikit-learn [96] for statistical computations, and pandas [95] as well as numpy [93] to process, aggregate, and analyze data.

Disclaimer (in accordance with the regulations on AI usage at KIT[7], specifically § 3.2): Code for data analysis and plotting using the previously mentioned Python libraries was partially generated or inspired by ChatGPT using the GPT-4o and GPT-4.1 mini models.

[7] "Guidelines for the use of generative AI in teaching at the KIT Faculty of Computer Science (AI Guidelines for Computer Science)", 22.07.2024, https://www.informatik.kit.edu/downloads/studium/Guidelines_Generative_AI_Informatics.pdf

Name	Description
None	CFS/EEVDF with no CBC
`sched_ext`	`sched_ext`-based CBC as a round robin scheduler (§ 5.4.1) with the CPU limitation based on the total CXL bandwidth (§ 5.4.2)
vtime-`sched_ext`	`sched_ext`-based CBC with vtime extension (§ 5.4.1.4) with the CPU limitation based on the total CXL bandwidth (§ 5.4.2)
`sched_ext` with throttle DSQ	`sched_ext`-based CBC with a separate dispatch queue for tasks to be throttled (§ 5.4.1.5) with the CPU limitation based on the total CXL bandwidth (§ 5.4.2)
cgroups	cgroups-based CBC (§ 5.4.2) with the CPU limitation algorithm based on individual task bandwidth (§ 4.3.3.2)
simple cgroups	cgroups-based CBC (§ 5.4.2) with the CPU limitation based on the total CXL bandwidth (§ 5.4.2), as used in the `sched_ext`-based approaches

Table 1: List of different CBC implementations with different features enabled, as implemented in Chapter 5.

6.1 Evaluation Setup

For the evaluation, we combine a memory-bound workload (i.e., fio [6]) with a compute-bound workload (i.e., stress-ng [61]). This combination is essential for our approach: If the CXL bandwidth is saturated, we throttle CXL-intensive tasks and schedule less CXL-dependent tasks instead, thereby using the CPU time for meaningful work, as described in Chapter 4. If there were no other tasks than CXL bandwidth-intensive tasks, there is no other meaningful work to do instead of waiting for CXL memory requests, rendering our approach useless.

The *Flexible IO Tester (fio)* [6] (version 3.41) is a memory benchmark with controllable memory access patterns, number of CPU cores, and block sizes. It is focussed on

filesystem-based IO and can use different IO engines to access memory. Since we want to benchmark the performance of CXL memory accesses, we use the `mmap` engine together with a temporary file system (tmpfs) backed by CXL memory. The `mount` command [75] allows creating such a tmpfs while specifying the NUMA node for the backing memory via `--options mpol=bind:NID`. For `NID`, we use one of our virtual CXL NUMA nodes (§ 5.1.5). In addition, we use `numactl --membind=NID` to create a memory binding when launching fio. Although fio uses the CXL-backed tmpfs to access memory, the memory binding is important for our evaluation because our CXL bandwidth control implementations use it to recognize processes they should control. Unless stated otherwise, we instruct fio to perform random reads with a block size of 4 MiB.

The tool *stress-ng* [61] (version 0.18.06) can be used to stress test various components of a system. For our evaluation, only the CPU-focssed stress testing is relevant. Although stress-ng allows specifying which CPU stress test method to use, we want the load to be as general as possible. Consequently, we use the default option where stress-ng cycles through different CPU stress test methods. To provide the scheduler with non-CXL tasks to schedule, we start 39 stress-ng processes (out of 40 hyperthreads on our system) in addition to the fio processes.

Our evaluation is conducted on a host made up of the following components: The CPU is an Intel Xeon Silver 4416+ [31] from the 4th generation of Intel Xeon Scalable processors with the Sapphire Rapids microarchitecture. It features 20 cores and 40 hyperthreads with a base clock frequency of 2.0 GHz and a maximum turbo frequency of 3.9 GHz [31]. The CPU is paired with 128 GB of memory consisting of 8×16 GB Micron DDR5 DIMMs with 4000 MT/s. The host runs our patched Linux kernel based on version 6.12 (§ 5.3.1). The CXL FPGA, as described in Section 5.1, is an Intel/Altera Agilex 7 I-Series FPGA [40] and is equipped with a single 16 GB Micron DDR4 DIMM with 3200 MT/s [88].

6.2 CXL Bandwidth Estimation

In this section, we evaluate the precision of estimating the CXL bandwidth of tasks. For this, we first examine the per-monitoring group hardware memory access counters of our FPGA-based CXL device (§ 5.1) before covering the per-process PEBS-based memory access estimation (§ 6.2.2). Then, we evaluate the combination

of the two approaches (§ 6.2.3), which is used by our CXL bandwidth control implementations.

6.2.1 CXL FPGA Memory Access Counters

To verify the correctness of our FPGA memory access counters, we use a single fio [6] process to perform random reads with a block size of 4 MiB at a fixed bandwidth of 1 GiB/s. For the virtual CXL NUMA node where fio is memory-bound to, our CXL FPGA reports a read bandwidth of 0.996 GiB/s and a write bandwidth of 0.00 GiB/s. If we instruct fio to use random writes instead, still with a bandwidth limitation of 1 GiB/s, our FPGA reports a write bandwidth of 0.990 GiB/s but also a read bandwidth of 0.992 GiB/s, resulting in a bandwidth of 1.982 GiB/s in total. Although fio generates write-only memory traffic in software, the FPGA-based CXL device still reports the same amount of reads on the hardware level. This can be explained by the fact that Intel CPUs use the *write-allocate* cache allocation strategy for CXL [110, 111]. This means that if a value should be written to an address that is not present in the cache, the corresponding cache line is loaded into the cache, where the value is updated [28:B.1]. Based on the amount of random memory reads, cache lines have to be evicted from the cache at the same speed as new lines are read into the cache, causing write-backs to memory with the same bandwidth as the reads. With this information, we modify our PEBS-based bandwidth estimation from Section 5.3 so that for any sampled memory write it also accounts a read of the same size. Otherwise, the read incurred by a write would not be accounted for, causing a wrong bandwidth estimation of tasks sharing a monitoring group.

For a mixed read-write access, the FPGA reports a read bandwidth of 1.65 GiB/s and a write bandwidth of 1.03 GiB/s, resulting in 2.68 GiB/s. Although the bandwidth for reads and writes is limited to 1 GiB/s in fio, the read bandwidth on the CXL FPGA is higher because it also includes the loads incurred by the writes.

6.2.2 PEBS-based CXL Memory Access Estimation

To evaluate the precision of the PEBS-estimated bandwidth, we use two fio [6] processes with the same target bandwidth of 512 MiB/s, memory bound to the same NUMA node and thus also using the same pair of hardware read/write memory access counters. The cgroups-based approach described in Section 5.4.2 is used as a bandwidth monitoring tool by disabling the CPU time limiting of processes.

Figure 7 shows the CXL bandwidth of both fio processes as sampled by our PEBS implementation. The PEBS-based CXL bandwidth estimation reports a mean bandwidth of 3.5 MiB/s and 3.0 MiB/s, respectively. For both fio processes, PEBS heavily underestimates the used CXL bandwidth, as it is around 200 times lower than the actual bandwidth of 512 MiB/s.

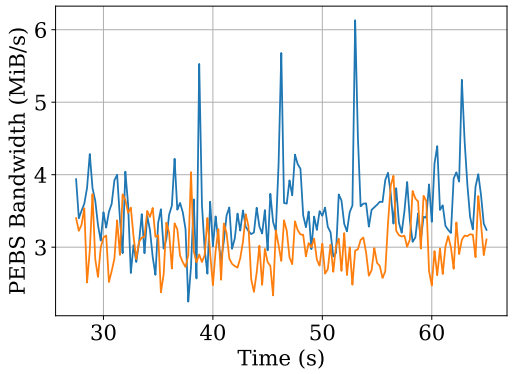


Figure 7: Bandwidth over time as reported by the PEBS-based CXL bandwidth estimation. Although both processes generate 512 MiB/s of reads, their bandwidth is only estimated to be around 3 to 4 MiB/s.

6.2.3 Combination of PEBS-based and FPGA-based Access Counters

Figure 8 shows the combination of the PEBS-based bandwidth estimation and the FPGA access counters according to Equation 3. Although the FPGA access counters do not improve the precision of the bandwidth accounting per process, they restore the correct magnitude of the bandwidth. The mean bandwidth reported by the two fio processes is 571 and 496 MiB/s, respectively. This results in a root-mean-square error (RMSE)[8] of 77.2 and 52.0 MiB/s, respectively, which is 15.1 and 10.4 % of the target bandwidth. This can be considered as the precision of our combined bandwidth estimation approach. In general, the combination of both approaches is effective, as the hardware memory access counters of the CXL FPGA are precise but only work on monitoring groups, while the PEBS-based access estimate is not precise but works per process.

[8] $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i - \overline{x})^2}$, with $\overline{x}$ being the mean of all x, as implemented by scikit-learn [96].

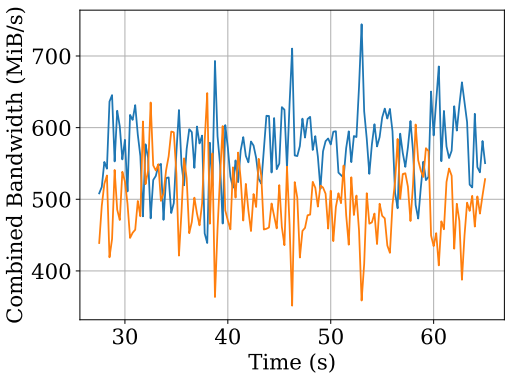


Figure 8: Combination of hardware memory access counters from the CXL FPGA with PEBS-based access estimation. The resulting bandwidth estimated for both processes is a bit different and varies over time, but is close to the actual bandwidth of 512 MiB/s.

6.3 CXL Bandwidth Limitation

Our CXL bandwidth control (CBC) implementations should keep the system at the optimal working point, i.e., at the point with the best ratio of bandwidth and latency. For this, we set the target bandwidth threshold of our CBC implementations to 4 GiB/s, which is based on the latencies measured later in Section 6.5. We determine that 5 fio worker processes are more than enough to saturate our CXL device if no CBC is used. We combine fio performing random reads of a block size of 4 MiB as a memory-bound workload with stress-ng [61] as a compute-bound workload, as explained in Section 6.3.1.

Figure 9 shows the aggregate CXL bandwidth of fio processes over time for different CBC implementations, while Figure 10 shows the bandwidth measured by individual fio processes. Table 2 provides statistical values of the bandwidth. In the aggregate bandwidth measurement in Figure 9, all plots show some kind of startup artifact where the bandwidth reported by fio oscillates between 0 and the final bandwidth. As this is also the case if no CBC is used, we consider this to be a general property. When no CBC is used, the unthrottled bandwidth is in the range from 7 to 10 GiB/s. The `sched_ext`-based approach (§ 5.4.1) reduces the CPU time of tasks such that the aggregate CXL bandwidth is at around 6 GiB/s but it does not meet the target

bandwidth of 4 GiB/s. The vtime extension (§ 5.4.1.4) merely improves upon the basic variant of the scheduler. Likely, there are not enough tasks in the dispatch queue for the vtime scaling to have an effect. Figure 10 shows that individual processes do not get any bandwidth for longer periods of time for all `sched_ext`-based approaches. Supposedly, these processes were not scheduled at all or only for a very small duration during these periods.

To verify that the two `sched_ext`-based approaches not meeting the target bandwidth is not an issue of the CXL bandwidth-based logic in our scheduler, we implement minimal variants of our schedulers: A round robin scheduler that simply assigns processes with the name "fio" a thousands of the time slices of normal processes, and a basic vtime scheduler that, in addition to reducing time slices, also sets the vtime of fio processes to one hundred times more than normal processes. However, we measured similar bandwidth results as with our schedulers including the bandwidth-dependent logic. Consequently, we attribute the insufficient bandwidth reduction to the general approach of CPU limitation in our schedulers.

The third variant of the `sched_ext`-based approach with a separate dispatch queue for tasks to be throttled (§ 5.4.1.5) is more effective in limiting the bandwidth. The root-mean-squared error is also considerably lower than without using any CBC. Although the mean bandwidth of 4.86 GiB/s is closer to the target bandwidth of 4 GiB/s, the bandwidth is heavily fluctuating, which is also indicated by the high standard deviation.

In contrast, both cgroups-based approaches (§ 5.4.2) are effective in limiting the bandwidth. However, Figure 10 shows some bandwidth spikes where individual processes suddenly report a very high bandwidth of around 4 GiB/s, which is the target bandwidth for the sum of all fio worker processes. We assume that the spikes are caused by how we enforce the CPU limit by setting a quota that a process may consume in a certain period. At the beginning of the interval, the used quota is zero, and the process starts by utilizing the CPU to its needs. Eventually, the quota is exhausted, and the process is not scheduled by CFS/EEVDF anymore until the next interval starts and the quota is reset again. If multiple processes have used up their CPU quota simultaneously, their total CXL bandwidth is almost zero. Then, a process with remaining CPU time has almost exclusive access to the CXL device and can use all its bandwidth until its CPU quota is exhausted or the interval of the other processes resets their CPU quota and they are scheduled again. Another observation

is that the standard deviation and the root-mean-squared error of the simple cgroups approach are smaller than those of the normal cgroups approach. We assume this is caused by the algorithm that sets CPU limits of tasks based on their CXL bandwidth usage, which cannot always reach the target bandwidth precisely, as discussed in Section 4.3.3.2.

In general, the cgroups-based approaches are effective in limiting the bandwidth to the target bandwidth threshold, while the `sched_ext`-based approaches have difficulties reaching the target bandwidth. Only the `sched_ext` approach with the throttle dispatch queue manages to reach the target bandwidth approximately.

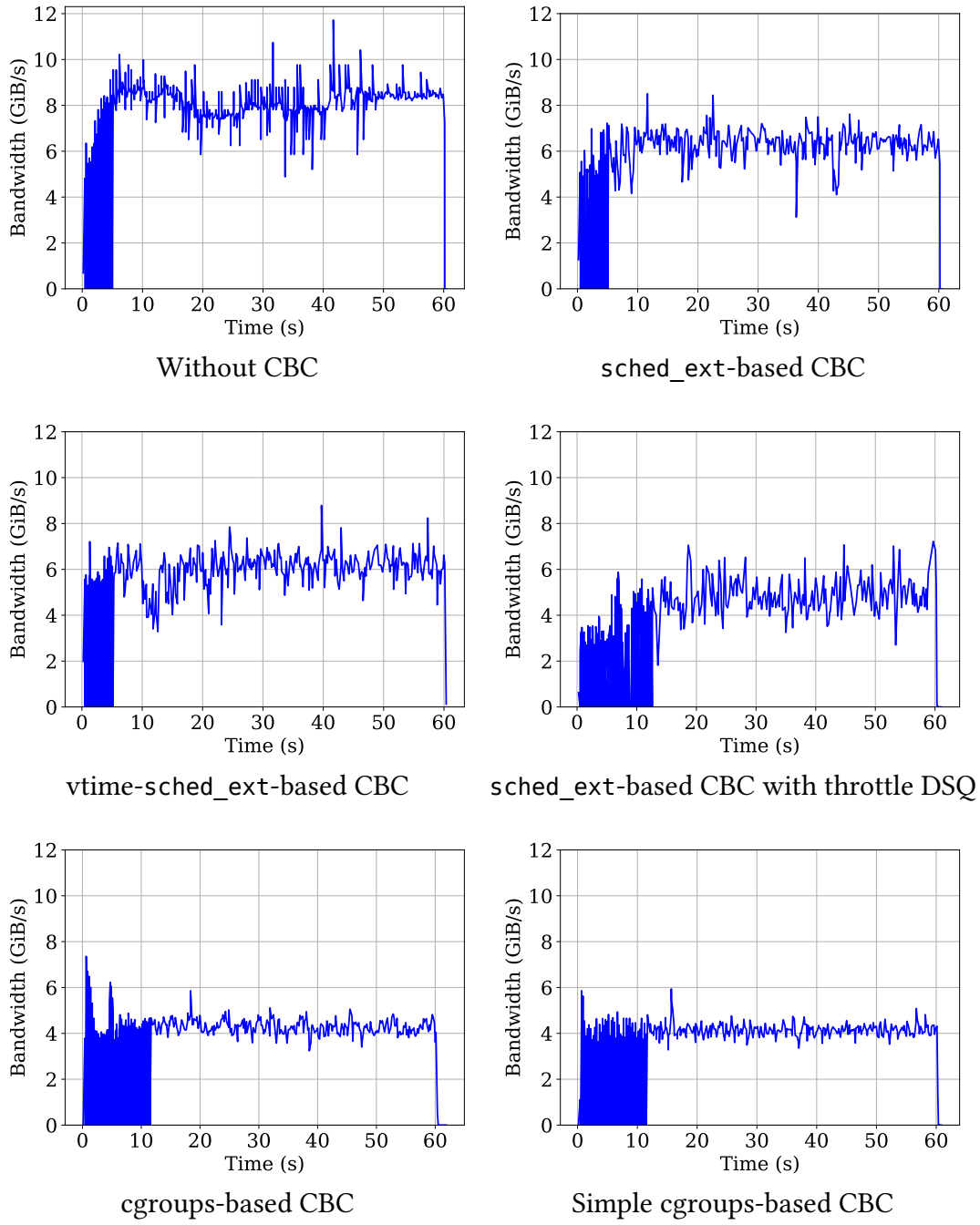


Figure 9: Aggregate CXL bandwidth using different CXL bandwidth control approaches. The cgroups-based approaches manage to limit the bandwidth to the target bandwidth of 4 GiB/s. The `sched_ext`-based meets the target bandwidth sometimes but has bigger fluctuations in its bandwidth. The other two `sched_ext`-based approaches have a bandwidth of around 6 GiB/s and do not manage to meet the target bandwidth.

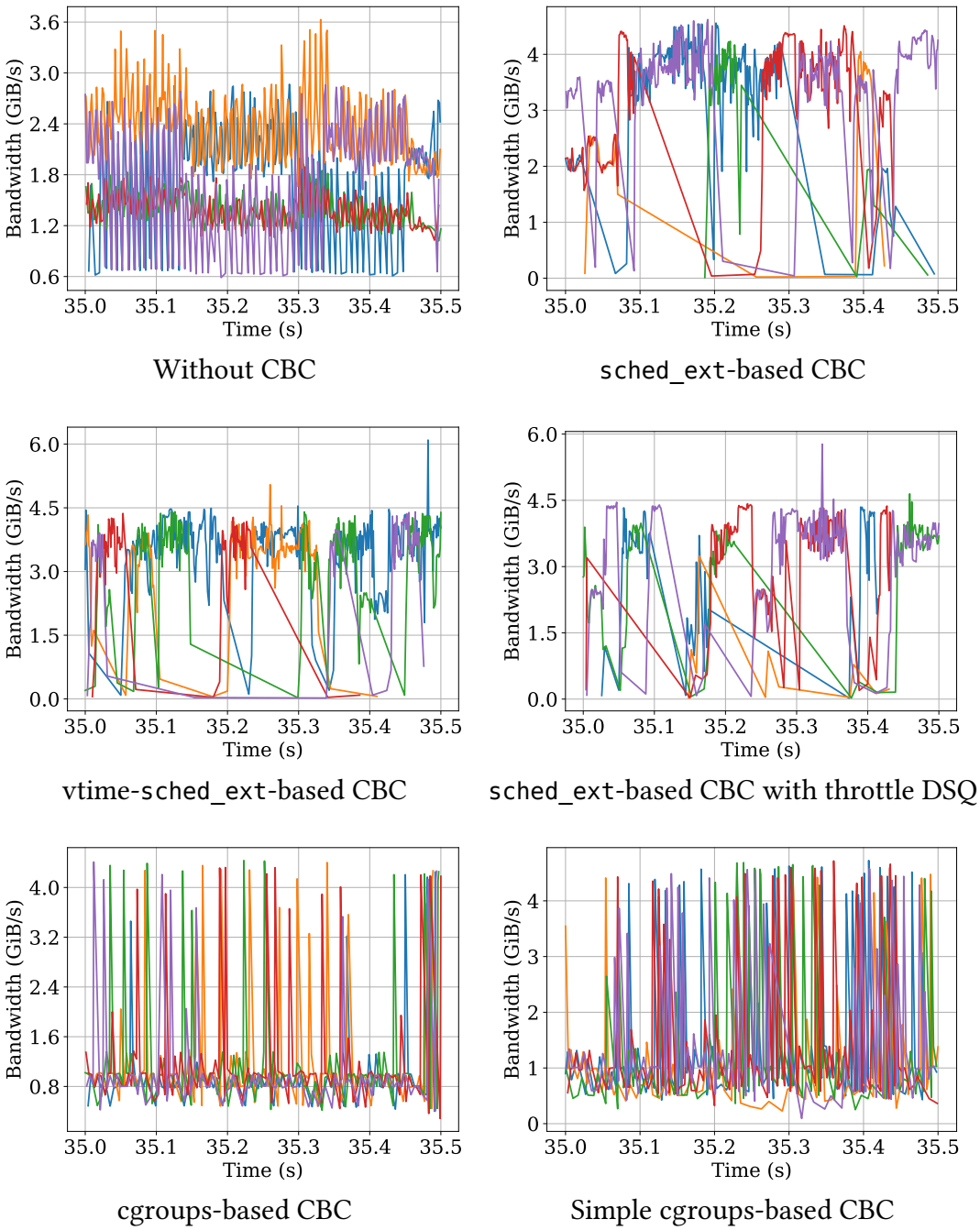


Figure 10: CXL bandwidth of individual processes using different CXL bandwidth control approaches as measured by fio. In all cases, the bandwidth fluctuates. While for `sched_ext`-based approaches individual processes do not get bandwidth at all for longer periods, processes have bandwidth spikes in cgroups-based CBC implementations.

CBC	Mean (GiB/s)	Stddev (GiB/s)	RMSE (GiB/s)	Q_1 (GiB/s)	Q_3 (GiB/s)
None	8.04	0.81	4.12	7.73	8.49
`sched_ext`	6.32	0.68	2.42	6.01	6.71
vtime-`sched_ext`	6.18	0.64	2.27	5.86	6.53
`sched_ext` with throttle DSQ	4.86	0.79	1.16	4.31	5.31
cgroups	4.27	0.31	0.41	4.05	4.45
simple cgroups	4.14	0.25	0.28	3.97	4.31

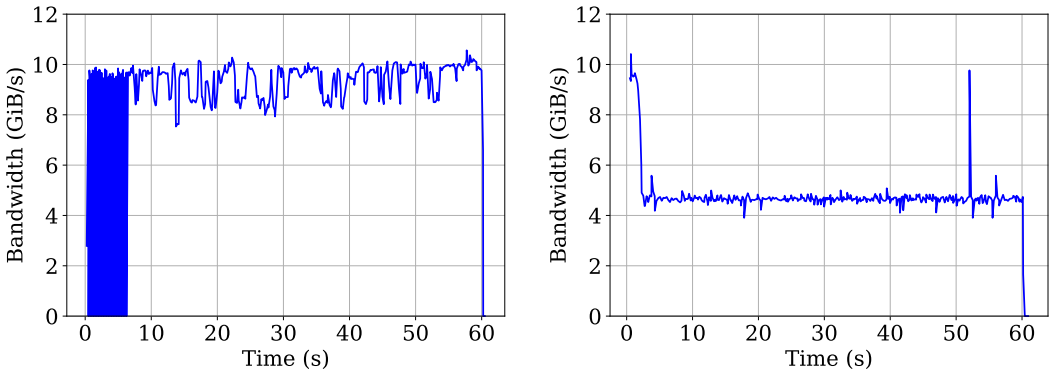
Table 2: Statistical values of different CBC implementations trying to limit the total CXL bandwidth to 4 GiB/s. The values are computed for the time period between 20 and 55 s of the benchmark to exclude startup and teardown artifacts. The root-mean-squared error describes the difference to the target bandwidth of 4 GiB/s. For all CBC approaches, the mean bandwidth is closer to the target bandwidth and the root-mean-squared error is lower. Especially the `sched_ext`-based approach with throttle dispatch queue as well as the cgroups approaches perform well.

6.3.1 CXL Bandwidth Control on an Idle System

As mentioned in Section 6.1, our approach is designed for workloads where non-CXL-using tasks can be scheduled if the CXL bandwidth is saturated. Specifically, the `sched_ext`-based CBC requires such a workload. Otherwise, reducing the time slices of tasks has no effect as there are not other tasks to schedule, leading to the same task to be scheduled again, allowing it to use the full CXL bandwidth. In contrast, the cgroups-based CBC can also limit the CPU time of tasks if the system is in idle otherwise. To show the difference in handling an idle system, we perform the same fio benchmark [6] as in Section 6.3 but without running stress-ng [61] in the background to consume remaining CPU time.

Figure 11 shows the aggregate CXL bandwidth as measured by fio. For the `sched_ext`-CBC, there is no limitation of CXL bandwidth, as it is around 10 GiB/s. The vtime extension has no influence on the resulting bandwidth, as there are no other processes to prioritize in the dispatch queue. Therefore, we have not included

its plot. In contrast, the cgroups-based CBC remains effective in limiting the bandwidth. Compared with Figure 9, the bandwidth is even more stable, likely caused by less contention and inference between processes.



`sched_ext`-based CBC cgroups-based CBC

Figure 11: Aggregate CXL bandwidth as measured by fio without stress-ng consuming the remaining CPU time while using the `sched_ext`-based CBC. The `sched_ext`-based CBC does not reduce the bandwidth at all, while the cgroups-based approach successfully limits the bandwidth to the target bandwidth of 4 GiB/s.

6.3.2 Asymmetric Bandwidth

The algorithm that sets CPU limits for tasks based on their individual CXL bandwidth (§ 4.3.3.2) implemented in the cgroups-based CBC approach is supposed to reduce the CPU limit of tasks with more bandwidth more heavily. We, therefore, inspect the imposed CPU limits for two processes with differing CXL bandwidth usage. Specifically, we create two fio jobs, one with a bandwidth limited to 1 GiB/s and the other one limited to 3 GiB/s, while setting the bandwidth threshold of our cgroups-based CBC to 3 GiB/s. The aggregate bandwidth of both fio processes exceeds this threshold, causing our bandwidth limiting to take effect. Figure 12 shows the used CPU time and the CPU limit as set by our cgroups-based CBC for both fio processes. For the process with a higher configured CXL bandwidth usage (orange plot), we also see a higher CPU utilization because submitting memory requests also takes CPU time. Its actual CPU usage of around 40 % is close to its CPU limit of around 45 %. The other process (blue plot) has a CPU usage of around 25 %, but its CPU limit is way higher at around 65 %. Since fio only performs memory requests up to the

configured limit, the process does not use its CPU time otherwise and thus yields the CPU. However, due to its low CXL bandwidth usage, our approach would allow the process to use more CPU time.

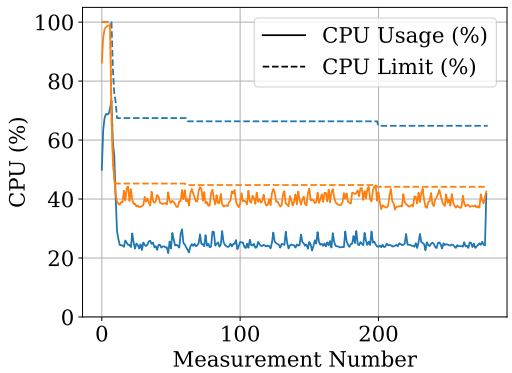
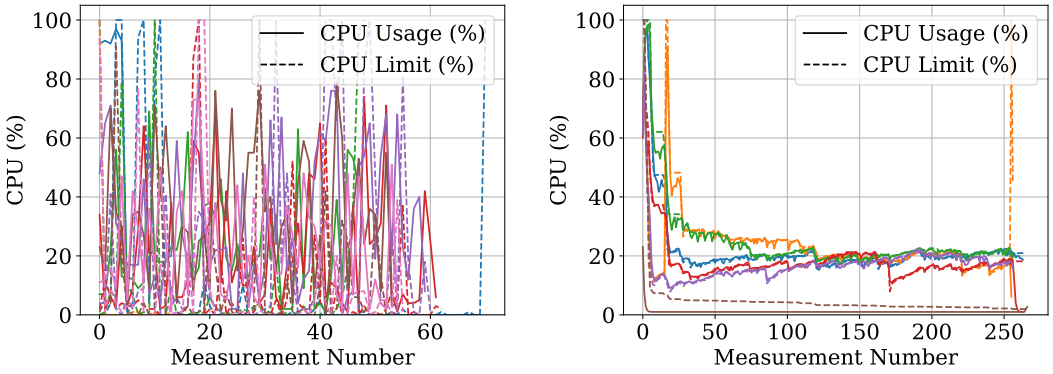


Figure 12: CPU time usage and CPU limit of two processes with a configured CXL bandwidth of 1 and 3 GB/s, respectively. The process with the higher CXL bandwidth also has the higher CPU usage and is throttled by its CPU time limit. The other process causes less CXL bandwidth and is thus allowed more CPU time.

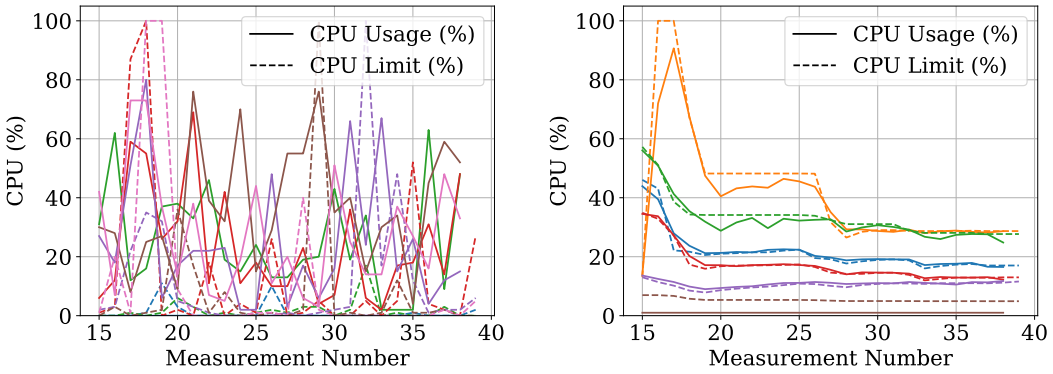
6.4 CPU Limits

To analyze the fluctuating bandwidth specifically observed for the `sched_ext`-based approaches (§ 6.3), we log the CPU usage as observed by our CBC implementations together with the CPU limits they set for individual processes. Figure 13 shows the actual CPU usage with solid lines and the limits with dotted lines, while Figure 14 provides a detail view.



vtime-`sched_ext`-based CBC cgroups-based CBC

Figure 13: CPU limit and CPU usage of fio processes using different CXL bandwidth control approaches. For the `sched_ext`-based approach, the CPU usage and limit fluctuate heavily, while they settle for the cgroups-based approach.



vtime-`sched_ext`-based CBC cgroups-based CBC

Figure 14: Detail view of CPU limit and CPU usage of fio processes using different CXL bandwidth control approaches. For the `sched_ext`-based approach, the CPU usage and limit fluctuate heavily, while they settle for the cgroups-based approach.

For the `sched_ext`-based CBC, the CPU limits do not settle after a certain amount of time but are constantly readjusted during the whole benchmark. This corresponds to the fluctuating bandwidth observed previously (§ 6.3).

In comparison, the CPU limits under the cgroups-based CBC settle after the initial phase. The bandwidth spikes of individual processes under the cgroups-based CBC

visible in Figure 9 are not visible in the CPU usage. Also, the CPU limit and the used CPU percentage have only marginal differences for most processes. Only a single process uses a small percent of CPU, and even considerably less than allowed by the CPU limit. Based on having the lowest PID number, meaning that it is the process that started first, we track this process back to the fio control process. It monitors the worker processes and periodically collects their current status, without actively generating load itself, which explains its low CPU usage. Allocating more CPU time to tasks that use less CXL bandwidth is a feature of the cgroups-based CBC, as already shown in Figure 9.

6.5 CXL Memory Access Latency

We used the Intel Memory Latency Checker (MLC) [116] to generate Figure 15 showing the latency in different bandwidth usages while using CFS/EEVDF without any CBC. The figure shows that beyond 5000 MiB/s, the latency increases exponentially while there is almost no bandwidth gain.

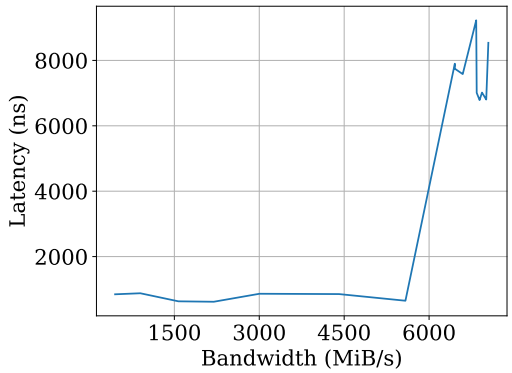


Figure 15: CXL memory access latency under different bandwidth usages generated by Intel MLC. Starting at around 5000 MiB/s, the latency increases exponentially.

We also wanted to use Intel MLC to generate the same plot for the cgroups-based and `sched_ext`-based CBC, but the PEBS sampling module (§ 5.3) fails to identify its memory accesses, although we explicitly bind memory allocations to CXL via `numactl --membind=1` and use MLC's `-j` option to specify the memory NUMA node as well. Our FPGA access counters report MLC's bandwidth successfully, indicating that actual CXL memory traffic is generated by MLC. However, the PEBS-based

bandwidth estimation returns a zero bandwidth for MLC. Consequently, our CXL bandwidth allocation does not throttle the CPU time of the corresponding process, thus having no effect on the used bandwidth and latency. Since MLC is not open source, we also cannot further analyze why PEBS fails to sample its memory accesses.

Instead of MLC, we use fio [6] to inspect the CXL memory access latency under bandwidth usage. We use 10 fio worker processes to perform random reads on the CXL device with a block size of 4 MiB/s while instructing fio to log its experienced latencies. Based on Figure 15, we set the target bandwidth threshold of our CBC implementations to 4 GiB/s. This way, we allow some imprecision in the bandwidth limitation of our CBC implementations before experiencing high CXL latencies. Figure 16 shows the completion latency over time for different CBC implementations, while histograms of the latencies can be found in Figure 17, and Table 3 contains statistical values like mean, standard deviation, and percentile quartiles.

When using no CBC, there is a high concentration of latencies around 5000 til 7000 µs, with some latencies ranging up to 20 000 µs. Considering the used block size of 4 MiB, the CXL hardware has to perform 65 536 individual cache line accesses of 64 bytes for one block. This explains why the block access latency is in the millisecond range. If the `sched_ext`-based CBC (§ 5.4.1) is used, the mean latency is 13.7 % lower than in the base case. The histogram shows that this CBC typically keeps the latencies under 2000 µs. The `sched_ext`-based CBC with the vtime extension (§ 5.4.1.4) can improve upon the basic `sched_ext` approach by causing a 30.1 % reduction of the average latency compared to the case without CBC. Although the latencies for the `sched_ext` approach with separate throttle DSQ are 27.4 % lower than without any CBC, it cannot beat the vtime-`sched_ext` CBC. However, the standard deviation for all `sched_ext`-based CBC approaches is more than 7 times higher than without a CBC, and the mean is outside the region between the 25 % (Q_1) and 75 % (Q_3) quartile, indicating that the latency has occasional spikes. This is also shown by the latency over time in Figure 16. We suspect that fio reports such high latencies when it starts reading a block but is preempted and not scheduled for a longer period of time. The cgroups-based CBC (§ 5.4.2) as well as its simpler variant are effective in keeping the latency low, as the latency is 25.9 and 25.2 % lower than if no CBC is used. Both histograms show that latencies are usually kept well under 5000 µs. Although the mean latency for the cgroups-based CBC approaches is

not lower than that of the `sched_ext`-based approaches, their standard deviation is only about half as high. This matches their property of keeping the bandwidth more stable (§ 6.3), resulting in more stable latencies as well.

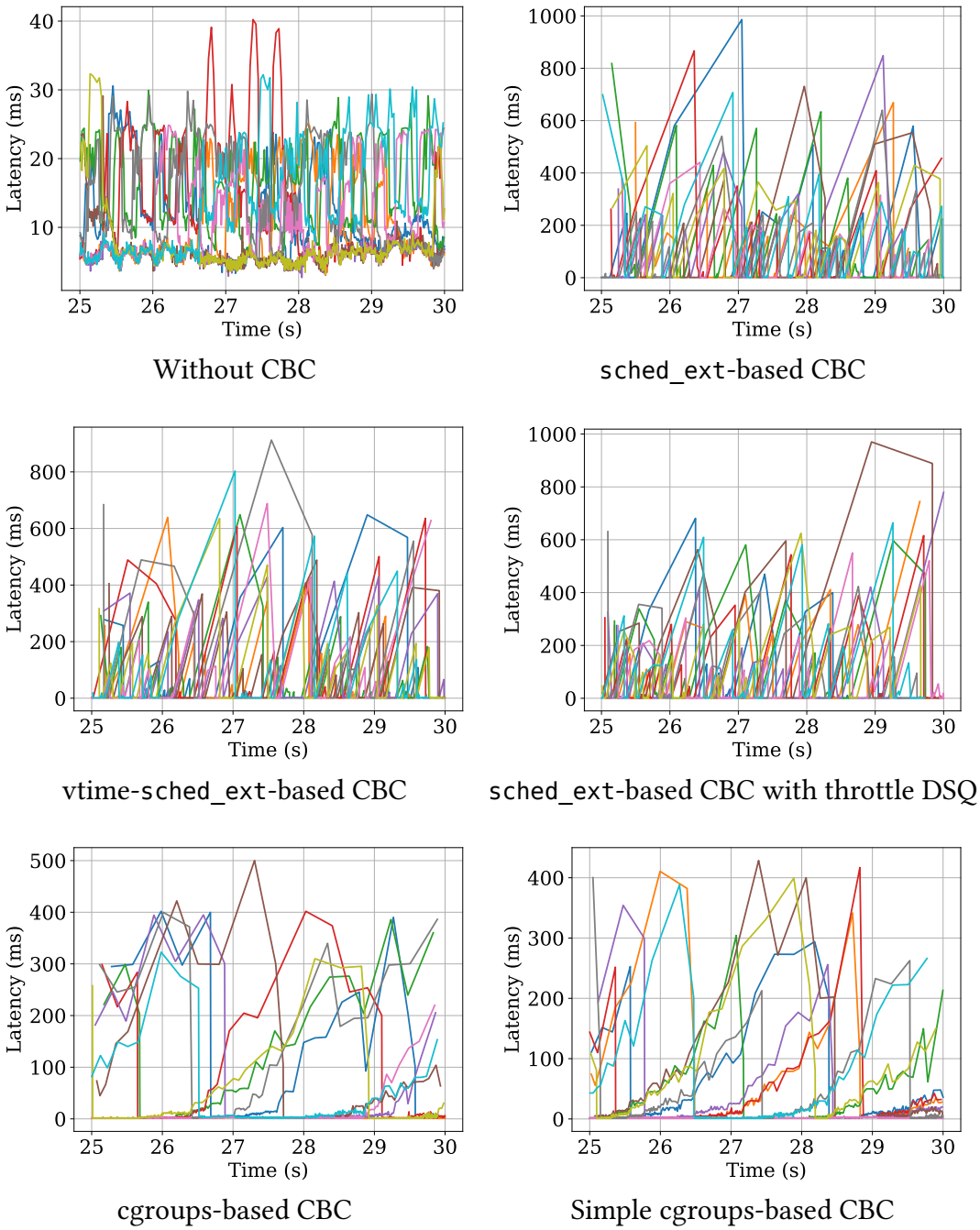


Figure 16: CXL memory access latencies for a block of 4 MiB as reported by fio processes while using different CXL bandwidth control approaches. All CBC approaches show high latency spikes compared to using no CBC.

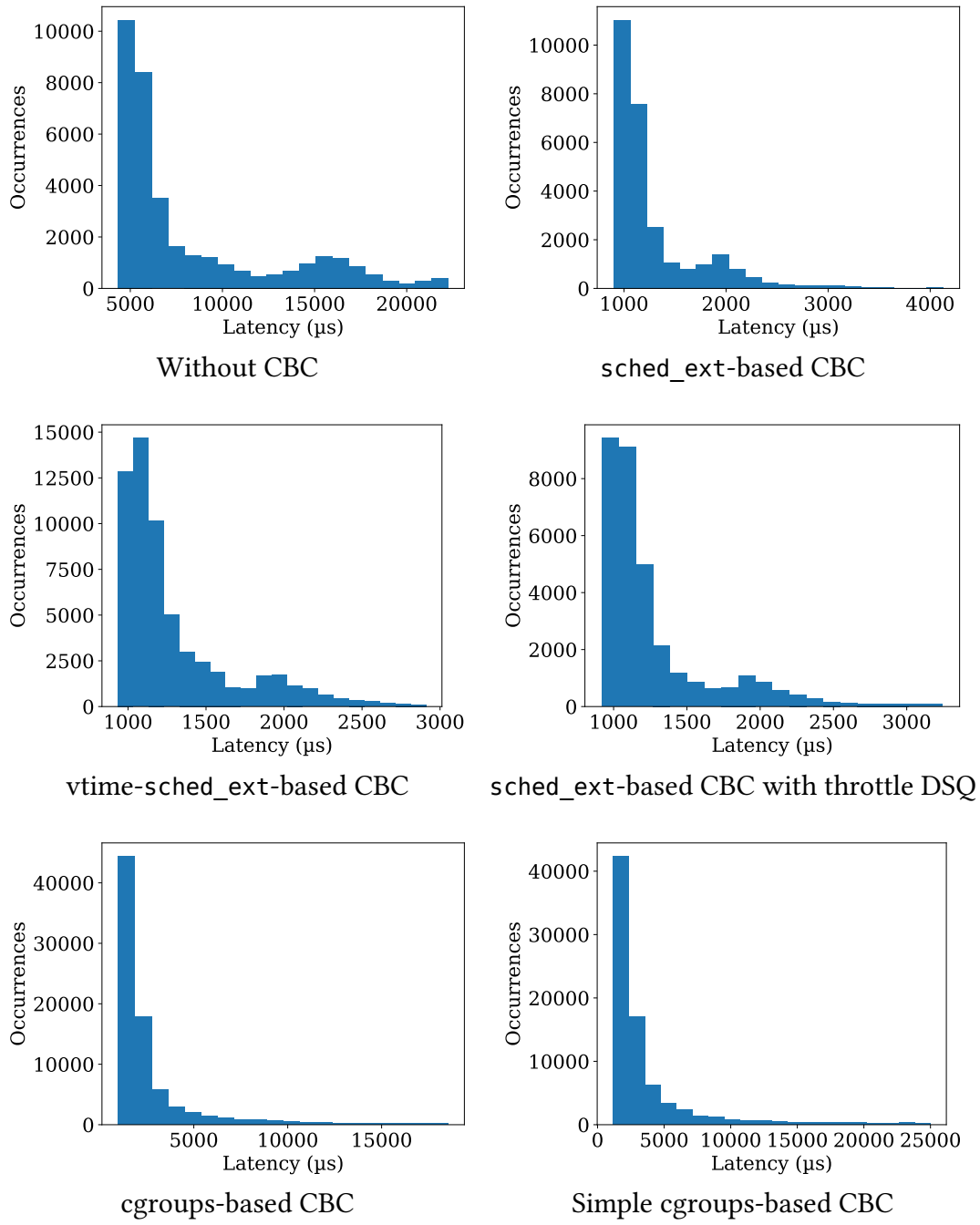


Figure 17: Histograms of the CXL memory access latencies for blocks of 4 MiB as reported by fio processes using different CBC implementations. To remove bars of non-visible size, only latencies between the 5 and 95 % percentiles are used. All CBC approaches manage to keep the majority of latencies under 5000 µs, which was the lower end of latencies without a CBC.

CBC Implementation	Mean (µs)	Stddev (µs)	Q_1 (µs)	Q_3 (µs)
None	9003	6093	5072	11 120
`sched_ext`	7772	47 229	990	1430
vtime-`sched_ext`	6292	42 953	1034	1459
`sched_ext` with throttle DSQ	6533	43 211	1015	1382
cgroups	6671	26 622	1293	2794
simple cgroups	6731	23 219	1680	3967

Table 3: Statistical values of the impact of different CBC implementations on the CXL memory access latency. The values are computed for the time period between 20 and 55 s of the benchmark to exclude startup and teardown artifacts. All CBC approaches reduce the mean latency compared to using no CBC, but also cause a higher standard deviation.

6.6 CPU Cost

The CPU cost metric by Werling et al. [118] is a measure of how much CPU time it takes to write one GiB of data. While originally designed for filesystems, we determine the CPU cost for accessing CXL memory under our CBC implementations. For this, we modify the calculation of the CPU cost by replacing the amount of data written by the amount of data read, as shown in Equation 10. Random writes would incur the same amount of reads on the CXL device because of the write-allocate cache strategy, as described in Section 6.2.1. Because this is transparent to the software, fio would not report the additional data transfers of reads caused by writes, resulting in an overestimated CPU cost that does not match the actual hardware workload.

$$\text{CPU cost} = \frac{\text{CPU time}}{\text{Amount of data read}} \tag{10}$$

Again, we use the combination of fio [6] and stress-ng [61], as described in Section 6.1. To compute the CPU time, we use the aggregate amount of data read by all fio worker processes and the sum of user and system CPU time reported by the `time` utility [76]. Figure 18 shows the CPU cost for different CBC implementations. All CBC variants improve the CPU cost with only marginal differences between the CBC implementations. They lower the CPU cost by about 40 %, reducing it from about 0.6 to about 0.35. Although the CBC implementations performed differently when limiting the bandwidth (§ 6.3), this difference is not noticeable for the CPU cost. The `sched_ext` approach with separate throttle dispatch queue (§ 5.4.1.5) is the best performing variant with only a 4 % improvement over the worst performing basic `sched_ext` approach (§ 5.4.1).

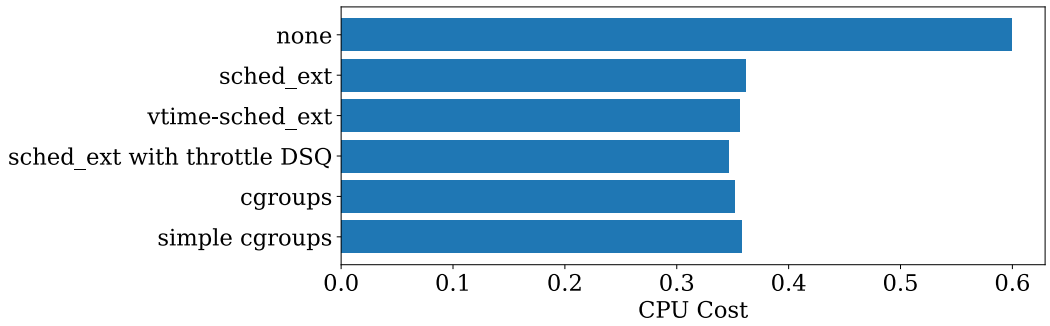


Figure 18: CPU cost for different CBC approaches. All CBC implementations improve the CPU cost over using no CBC.

6.7 CPU Throughput

We set out to increase the system throughput by reducing CXL-dependent CPU stall cycles and instead use the CPU time for meaningful work. In Section 6.6, we already showed that our approach reduces the CPU cost for accessing CXL memory. In addition, we now inspect the impact of the CXL bandwidth control on other tasks that do not use CXL memory. For this, we again use a combined memory-bound and compute-bound workload (§ 6.1). On the one hand, we use fio [6] as a CXL memory bandwidth-intensive workload. Specifically, we use 5 worker processes to conduct random reads on the CXL device until they have read 100 GiB in total. On the other hand, we use stress-ng [61] as the CPU-intensive workload. We start stress-

ng with 39 worker processes (out of 40 CPU cores on our evaluation system) and kill it when fio is finished. We use the amount of bogus operations (bogo ops) per second reported by stress-ng as a measure of the CPU throughput [61]. Figure 19 shows the number of bogus operations per second for different CBC implementations. All CBC implementations improve the system throughput compared to the baseline when no CBC is used. The cgroups-based approach with the algorithm that determines the CPU limit for tasks based on their individual CXL bandwidth (§ 4.3.3.2) improves the throughput by 6.7 %, while the `sched_ext` implementation with the separate throttle dispatch queue (§ 5.4.1.5) is the best performing variant of the `sched_ext` approaches with a throughput improvement of 4.5 % over the base case without a CBC. Thus, our CBC approaches not only lower the CPU cost of CXL memory access, as shown in Section 6.6, but also increase the CPU throughput for other tasks on the system.

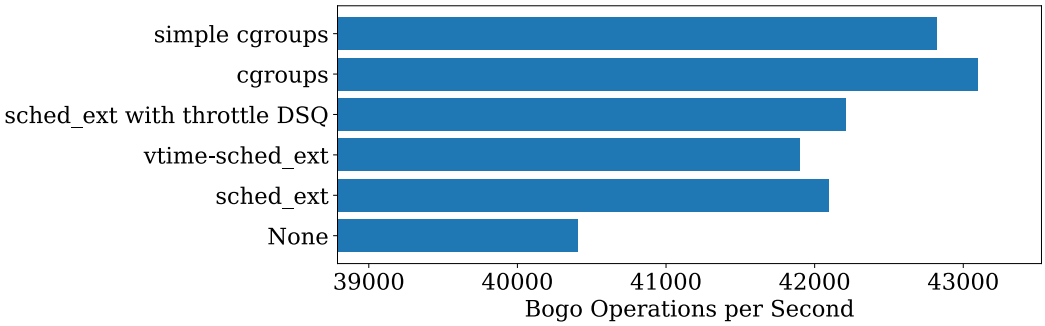


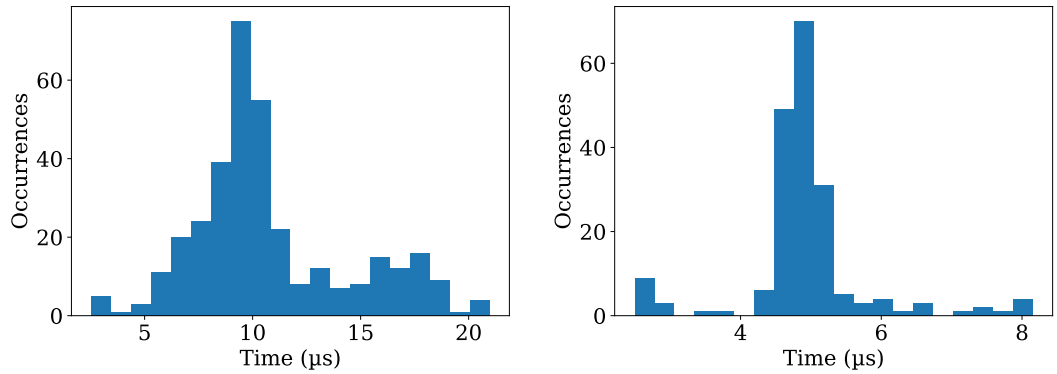
Figure 19: Bogus operations per second reported by stress-ng for different CBC implementations while fio utilizes the CXL bandwidth. All CBC implementations improve the throughput compared to when no CBC is used.

6.8 Overhead

All CBC implementations involve various processing steps, including PEBS-based sampling (§ 5.3), retrieving the memory access counters (§ 5.1.4), calculating the CXL bandwidth per task (§ 4.3.2), determining the CPU limit (§ 4.3.3) as well as enforcing it (§ 5.4). In this section, we examine the overhead of the `sched_ext`-based CBC approach (§ 6.8.1) and the cgroups-based CBC approach (§ 6.8.2).

6.8.1 sched_ext-based CBC

To investigate the overhead of our `sched_ext`-based scheduler (§ 6.8.1), we measure the time it spends in the `sched_enqueue()` callback function as well as the duration for fetching the CXL memory access counters from the FPGA. For this, we used `bpf_ktime_get_ns()` [21] to retrieve the current monotonic clock timestamp at the beginning of the function and immediately before it returns to compute the time spent in the function. The histograms of the durations are shown in Figure 20. The mean duration of `sched_enqueue()` is 10.7 µs while retrieving the FPGA counters takes 4.9 µs on average.



`sched_enqueue()` callback Fetching FPGA counters

Figure 20: Histograms of duration of `sched_ext` scheduler functions with a mean of 10.7 µs for `sched_enqueue()` and 4.9 µs for retrieving the counters from the FPGA.

To improve performance, the FPGA counters are not fetched every time a task is scheduled. Instead, they are only fetched if a certain amount of time (currently 1 ms) has elapsed since the last retrieval. Otherwise, the stored current running average is used (§ 5.4.1.1). Configuring this fetching interval is a trade-off between overhead and responsiveness. On the one hand, if the counters are retrieved more frequently, the overhead of `sched_enqueue()` increases. On the other hand, if the interval in which the counters are retrieved is too long, the scheduler cannot react quickly to changing bandwidth usage of tasks.

To inspect the general overhead of an eBPF-based `sched_ext` scheduler, we also profiled a basic round robin sched_ext scheduler. Due to the nature of a round robin scheduler, enqueuing a task and selecting the next task to be dispatched from the run

queue to the CPU is trivial. The mean duration of its enqueue operation is 137 ns. Hence, the duration of our `sched_enqueue()` function is almost exclusively caused by our CXL bandwidth control logic.

To set the duration of our `sched_enqueue()` function in contrast with a scheduler that does more complex decisions than just enqueuing tasks in a FIFO manner, we profiled the default CFS/EEVDF Linux scheduler as well. The scheduler statistics exposed via the procfs file `/proc/schedstat` [49] as well as `perf sched` [80] do not seem suitable because they report scheduling latencies per CPU and per task, respectively, but not the consumed CPU time of the scheduler itself. Instead, we use the tool bpftrace [101] to attach small eBPF programs to kprobes to measure the duration of scheduling functions. According to the Linux documentation [48:§6], `enqueue_task()` and `pick_next_function()` are the central functions of the CFS/EEVDF scheduler. We register kprobes and kretprobes to these functions, which are triggered at the entry point of the functions and when they are about to return, respectively. By saving the start timestamp per CPU core in the kprobe, we can create a histogram of the time spent in the functions using the `hist()` function built into bpftrace's scripting language [101]. It automatically scales the bin sizes of the histogram logarithmically, i.e., they are a power of 2 [101].

Figure 21 shows the histogram of the durations of the `enqueue_task()` and `pick_next_task_fair()` functions of CFS/EEVDF. For `enqueue_task()`, the mean duration is 3.16 µs, and for `pick_next_task_fair()` it is 1.84 µs. Consequently, the overhead of our scheduler is in the same magnitude as CFS/EEVDF but is a bit higher, caused by the FPGA counter retrieval. Increasing the interval in which the FPGA counter values are retrieved would lower this overhead, but would also result in slower responses to changes in the bandwidth usage of tasks.

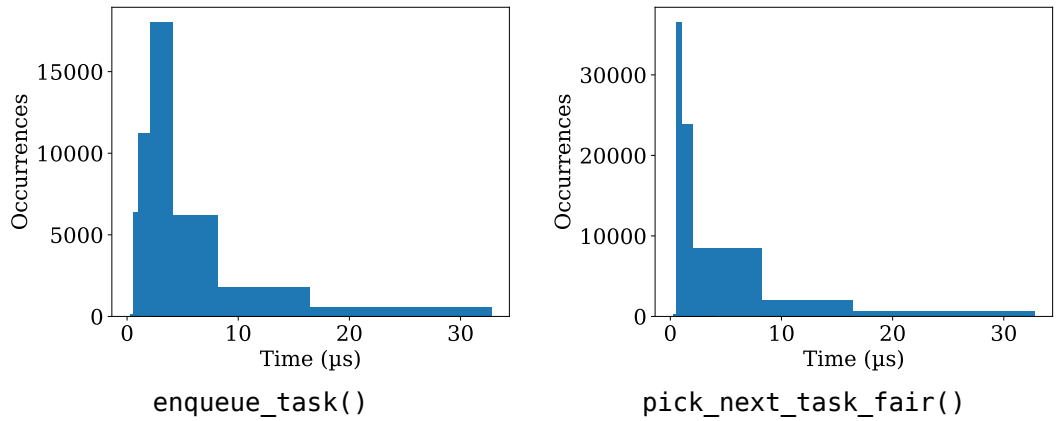


Figure 21: Histogram of the duration of kprobes attached to central EEVDF functions. The durations have the same magnitude as in our scheduler, although being a bit lower.

6.8.2 cgroups-based CBC

To determine the overhead of our cgroups-based CBC implementation (§ 5.4.2) we instrument its main loop with a duration measurement as well as start the program with the `time` utility [76] to obtain its CPU usage. Figure 22 shows a histogram of the duration of a loop iteration, which takes 30.8 ms on average. This always involves reading the per-NUMA node FPGA memory access counters, the per-process PEBS counters, determining the combined per-process bandwidth of each process, calculating the new CPU limit of each process, and setting them via the cgroups interface. The `time` utility [76] reports a CPU usage of 9 %. Note that a CPU usage of 100 % refers to fully utilizing a single core. The relatively low CPU usage is caused by the CBC logic only running every 150 ms, causing the process to sleep for the rest of the time, as described in Section 5.4.2.

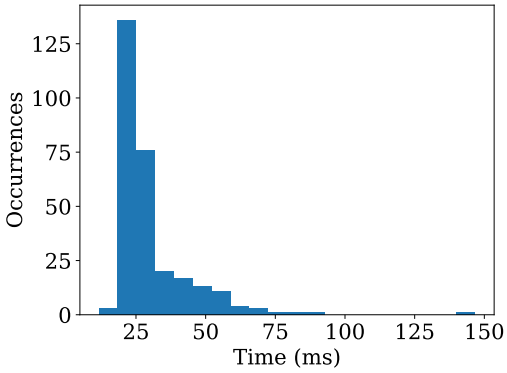


Figure 22: Histogram of the duration of a loop iteration of the cgroups-based CBC with most latencies at 25 ms.

Chapter 7

Discussion and Future Work

In this chapter, we discuss limitations of our approach, how our implementation could be different, and what the potential areas for future work are. First, we address challenges with the virtual address space copies of our CXL FPGA (§ 7.1) and its memory access counters (§ 7.2). We then discuss our PEBS-based bandwidth estimation (§ 7.3). Next, we cover the `sched_ext`-based and cgroups-based CXL bandwidth control (CBC) approaches (§ 7.4). Finally, we outline future areas for the evaluation (§ 7.5).

7.1 CXL FPGA Virtual Address Space Copies

Our initial idea was to expose multiple virtual copies of the physical address space of the CXL FPGA and use them to distinguish between accesses from different monitoring groups. Because these address spaces are aliases of each other and access the same underlying memory, they lead to memory corruptions and kernel crashes. We attempted to patch the kernel page allocator (§ 5.2) to avoid handing out the same offset inside a NUMA twice, so that pages do not alias. However, it needs further improvement to lead to a stable running system, as we have not found all paths through which pages can be allocated and freed. Currently, the address space of the FPGA is statically divided into smaller regions. Fixing the page allocator would allow to use the virtual copies and thus enable to dynamically allocate memory to tasks that are memory-bound to CXL-backed NUMA nodes. Building on that, the approach could be extended so that CXL memory can be shared between processes running on different NUMA nodes. Currently, if processes want to share CXL memory, they need to be memory-bound to the same NUMA node. This way, the processes

access the same underlying memory at the same host-physical address. The PEBS-based bandwidth estimation helps in attributing the FPGA-measured bandwidth to the processes sharing memory. We verified that page sharing still works correctly inside the same NUMA node by `mmap`-ing the same tmpfs file backed by CXL memory in two different processes and randomly reading or writing into it, respectively. In case processes want to share memory across NUMA node boundaries, the processes do not access the same host-physical address, because each NUMA node has its own host-physical address range. If the CXL FPGA exposes multiple virtual, aliased copies, accessing memory at the same offset in each NUMA node results in the same device-physical address on the CXL device. However, due to the different host-physical addresses being used, the CPU does not perform cache invalidation between them, as it considers accesses to different host-physical addresses to be independent of each other. A memory coherence protocol like Modified-Shared-Invalid (MSI) [20, 28:§5.2], usually implemented in the CPU to manage coherence between CPU cores, would need to be manually re-implemented on the OS level for this scenario. For this, the kernel could make use of "read" and write-protecting pages. If a page is shared between NUMA nodes, all aliased pages are write-protected by clearing the read-write flag of the page table entries (PTEs) [10:§2]. If a task attempts to write to that page, the OS receives a write page fault, and could then apply "read protection" to all other aliased pages by removing the present flag. Also, the write protection is removed for the page of the NUMA node where the task wanted to write to. If a task that is memory-bound to a different NUMA node subsequently tries to read from the same offset in the NUMA node's address space, the OS again receives a page fault and can then flush the cache of the task that previously wrote data to that page, set up write protection for the previous owning process of the page and remove the "read" protection for tasks of all monitoring groups. We estimate a substantial overhead for the coherence management because the setup and removal of each page protection requires a TLB flush. For example, Calciu et al. [11] have measured around 13 % lower throughput and 15 % higher latency under write page faults. Therefore, it might be more efficient to omit sharing pages across NUMA nodes and instead migrate processes to the same NUMA node, where page sharing can make use of the native coherence mechanism of the CPU. However, manual in-software cache coherence between different virtual CXL copies would not be necessary anymore in CXL 3.0, as it supports multiple logical devices to access the same CXL-physical page. By using *back-invalidation* introduced in CXL 3.0 [15, 63], the CXL device manages coherence

itself by snooping the cache of connected hosts as well as invalidating their cache lines if it snoops an invalidate message from one of the hosts.

7.2 CXL FPGA Memory Access Counters

For each access, our CXL FPGA accounts a full cache line of 64 byte to the respective monitoring group. However, for non-temporal (NT) stores, the access size may be different [37:§10.4.6]. By using special dedicated Streaming SIMD Extensions (SSE) instructions [37:§5.5], the CPU is instructed to directly write to memory and not load the corresponding cache line into memory first, because the next access to this data is temporally far away. If an application uses lots of non-temporal stores, the calculated bandwidth accounted by our hardware memory access counters (§ 5.1) might be more than the application actually uses. Related work also does not correctly account non-temporal stores, relying on the statement that they are rarely used [23]. Analyzing the actual usage of non-temporal stores in applications would give insights into whether future work should consider adding correct accounting of non-temporal stores.

Our approach was limited to CXL 1.1, because it is the newest CXL version supported by current CPUs at the time of writing [1, 33] and the existing CXL FPGA implementation [60]. Newer versions of the CXL standard enable various methods to obtain performance monitoring information from a CXL device. For example, CXL 2.0 introduces Multi Logical Devices (MLDs), which allows a CXL device to be connected to multiple connected hosts [15:§2.4]. This CXL version also adds QoS telemetry in the form of a `DevLoad` level that a CXL device can report to indicate light load, optimal load, moderate overload, or severe overload [15:§3.3.4.1]. The CXL device can determine these levels from the length of its request queue, which correlates to queueing delay [15]. Upon receiving a moderate or severe overload signal from the CXL device, the OS is supposed to increase its request-rate throttling, while it should decrease it on a light load signal, as described in table 3-29 of the CXL standard [15]. If a CXL device would use the MLD mechanism to expose multiple logical devices to the *same* host, different tasks could use different logical devices and thus perform memory requests with different Logical Device Identifiers (LD-IDs) [15]. The `DevLoad` level reported by each logical CXL device [15:§3.3.4.1] could then replace the hardware access counters per monitoring group on the CXL device.

Alternatively, CXL 3.0 defines a CXL Performance Monitoring Unit (CPMU) [15:§13.2], which is similar to the PMU of a CPU. The CXL PMU can be used to count events on the CXL device like loads and stores (i.e., `M2S RwD` and `M2S Req` of CXL.mem, respectively) [15:Table 13-5]. The Linux kernel and the tool `perf` already support CPMUs [58], thereby preceding supporting hardware [12]. In combination with MLDs, CPMUs could also be used to measure the CXL utilization of individual monitoring groups.

7.3 PEBS-based CXL Bandwidth Estimation

As already mentioned in Section 5.1.1, our FPGA is able to support 512 address space copies with an individual read/write counter pair for each copy. This means that we can precisely attribute the bandwidth of 512 monitoring groups. The maximum number of NUMA nodes in Linux is 1024, as declared in `arch/x86/Kconfig` [59], so this is not a limitation. Consequently, we question the necessity of PEBS sampling with its suboptimal accuracy and overhead. Additionally, the PEBS-based bandwidth estimation does not work for all workloads, for example, it reports a zero CXL bandwidth for Intel MLC, as mentioned in Section 6.5. For other workloads like fio in our evaluation in Section 6.2.2, it underestimates the CXL bandwidth by orders of magnitude. The PEBS-based bandwidth estimation could therefore see improvements in precision and general applicability.

In general, our implementation of the PEBS-based memory access estimation also requires direct access to the PMU. On the one hand, this means that the PMU is not available to perf, which limits profiling capabilities on a system using our approach. On the other hand, our implementation requires a CPU with at least performance monitoring version 3 [35:§21.3.1] and PEBS format version 1 [35:§21.3.1.1.1]. This is needed for the memory access address to be included in the PEBS sample (see “Data Linear Address” in [35:Table 21-4]). Without the address, our approach cannot distinguish between accesses to DRAM and CXL memory. This means that our approach cannot be used inside virtual machines based on KVM, be it for debugging or for cloud computing. Although we found a patchset that adds performance monitoring version 5 to KVM [125], the PEBS format version inside a KVM-based virtual machine is still 0[35:§21.8], thus missing the accessed memory address. Consequently, future work could try to implement PEBS format version 1 for KVM. This

would likely also help other programs running in a KVM-virtualized environment requiring or benefiting from a higher PEBS version. We refer to a discussion about the challenges of PMU feature support in KVM for further insights [107].

To reduce the overhead of the approach when the system is not in a CXL bandwidth overload situation, the PEBS-based bandwidth estimation of individual tasks could also be dynamically activated only when the CXL FPGA memory access counters report an overload. The idea is similar to the approach of Fried et al. [23:§5.2.2], who only sample LLC misses of individual tasks when the DRAM controller reports high bandwidth usage.

The PEBS-based bandwidth estimation, as well as the CXL FPGA memory access counters, could be avoided completely if PMUs in new CPUs would offer dedicated, per-core counters for loads and stores to CXL memory. Then, the bandwidth control could be implemented without hardware modifications and would require less overhead due to avoiding the PEBS-based sampling. Specifically, the third generation of Intel's Resource Directory Technology (RDT) seems promising [41:§3.2.4.3]. Its Memory Bandwidth Allocation (MBA) feature now allows per-core [41:§3.2.4.3] and region-aware bandwidth limitation, which can be used to treat DRAM and CXL accesses differently [41:§3.2.4.4]. However, we assume that throttling of tasks is still based on injecting delays, as in the previous versions [23]. This stands in contrast to our approach of using the CPU time for meaningful work instead. Another feature, the Memory Bandwidth Monitoring (MBM) [41:§3.1.4], could be useful if combined with our CXL bandwidth limiting approach. It allows the per-core and per-region bandwidth monitoring [41:§3.1.4.2], which could be used in combination with our bandwidth limitation approach (§ 5.4).

7.4 CXL Bandwidth Control

We evaluated that our CBC approaches indeed reduce the latency of CXL memory accesses (§ 6.5) as well as the CPU cost (§ 6.6). However, the overhead of the CBC implementations (§ 6.8) could be reduced further. Closely linked to this are the reaction times to bandwidth changes. Currently, the reaction times are not on a microsecond timescale, which stands in contrast to Caladan [23]. However, its implementation requires applications to explicitly use green threads provided by their runtime and have a suitable concurrency architecture. Otherwise, only limited

CPU time multiplexing is supported, which was not benchmarked [23]. In contrast, our approach does not require any modifications to applications.

Profiling our CBC implementations further could give insights where improvement is possible. The code of the cgroups-based CBC approach could be manually instrumented with timing measurements. For the `sched_ext`-based approach, bpftool [77] supports using perf to collect the metrics `cycles`, `instructions`, `l1d_loads`, `llc_misses`, `itlb_misses`, and `dtlb_misses` over a specified amount of time [44, 78]. However, our approach prevents this capability because all hardware event counters are disabled by our kernel patch (§ 5.3.1), so it can exclusively control them for the PEBS-based CXL access counter sampling. Consequently, trying to use the example from the bpftool-prog man page [78], adapted to our eBPF program, fails with the error "failed to create event cycles on cpu 0".

As shown in Section 6.3.1, the time slice adjustments of the `sched_ext`-based CBC are not effective if the system is in idle otherwise and has thus no other tasks to schedule. On the one hand, this can be considered sensible as there is no other usage for the CPU time anyway, so it can be used to maximize the CXL bandwidth. On the other hand, if the CPU time were limited, the CPU could go into power-saving modes instead, thus saving energy. The cgroups-based CBC approach enforces the CPU limit even if the system is in idle otherwise, so it has the opposite advantages and disadvantages.

7.4.1 sched_ext-based CBC

The reduction of time slices in our scheduler (§ 5.4.1) as well as the vtime-based priority reduction (§ 5.4.1.4) are not sufficient to reduce the CPU time of CXL bandwidth-intensive tasks so that the bandwidth threshold is met. The approach using a separate dispatch queue (§ 5.4.1.5) to throttle tasks is more effective but the resulting CXL bandwidth is not as stable as in the cgroups-based approach (§ 5.4.2). We assume that if the scheduler detects that the CXL bandwidth is no longer overloaded, it immediately schedules all tasks from the throttling dispatch queue, resulting in an overload again. Using a more gradual approach of dispatching the tasks from this queue to the CPU could improve the stability of the resulting bandwidth. A statistical approach could be used to determine if a task should be enqueued into the normal or into the throttle queue. For example, if a randomly drawn number is smaller than a CXL bandwidth-dependent threshold, the task is enqueued into the throttle

queue. This could enable more control than the previously binary decision whether to always move a task into the throttle queue, thereby smoothing the resulting CXL bandwidth. We consider the `sched_ext` approach to be promising if the problem with the fluctuating bandwidth can be solved because it has full control over scheduling decisions, allowing faster reactions to bandwidth changes than the cgroups-based approach.

In general, the control stability could be improved for all `sched_ext`-based approaches (§ 6.3). Currently, the control logic uses exponentially weighted moving averages for smoothing (§ 4.3.1.1). Setting the parameter that controls how much a new value influences the running average is a trade-off between a smooth resulting bandwidth and reaction speed (§ 4.3.1.1). We settled with weighting the new value by 30 % and the average by 70 %. Further tuning the parameter or using different techniques from the area of control engineering could improve the response behaviour, for example, by using a proportional-integral-derivative (PID) controller [92].

Future work could investigate retrieving the counters periodically outside the hot enqueuing function. For this, the eBPF timers feature would be a perfect fit, but it is not available for eBPF `sched_ext` programs, i.e., for type `BPF_PROG_TYPE_STRUCT_OPS`, unfortunately [21]. Alternatively, a user space helper program could be used to periodically load the counters and insert them into a BPF map shared between user space and the kernel space eBPF-based `sched_ext` program, as already mentioned in Section 5.4.1.1. This, however, would generate the additional overhead of passing the counters from kernel space to the user space helper only to pass them back to user space. Instead, the kernel module responsible for retrieving the counter values from the FPGA (§ 5.1.4) could directly write into the BPF map by obtaining a handle to the map using `bpf_map_get_fd_by_id()` and then `bpf_map_update_elem()`[21].

For more fairness in CPU throttling regarding tasks that only contribute negligibly to the CXL bandwidth overload, we designed an algorithm for setting the CPU limit of a task based on its individual CXL bandwidth (§ 4.3.3.2). However, we only implemented this in the cgroups-based CBC. Implementing it for the `sched_ext`-based approach may become a challenge due the eBPF verifier limitations, especially regarding recursion. Consequently, future work could explore different algorithms for the CXL bandwidth-dependent CPU time limitation of tasks that are simpler but deliver the same fairness.

In general, the `sched_ext`-based CBC approach misses all the engineering that went into advanced schedulers like CFS/EEVDF. Although our scheduler could be improved with inspiration from the schedulers provided in the scx repository [108], it will likely still lack behind CFS/EEVDF in universality. Furthermore, the eBPF development experience is not on par with typical application development used in the cgroups-based CBC. As an example, if the kernel eBPF verifier finds a violation (e.g., a missing NULL pointer check), it dumps the BPF bytecode together with an internal BPF error, leaving the developer with finding the problem in the source code. This is a contrast to the Rust programming language we used for the cgroups-based CXL bandwidth control. Here, the Rust compiler helps with finding and fixing errors by explaining compilation errors and suggesting solutions, while also enforcing strict safety guarantees. Using Rust instead of C to generate eBPF code may therefore be worth considering [7], as the Rust compiler should already catch some kernel eBPF verifier violations. Furthermore, the lack of floating-point arithmetic in eBPF leads to error-prone equation rearrangements and conversions to integer arithmetic.

7.4.2 cgroups-based CBC

The cgroups-based approach (§ 5.4.2) has the advantage that it avoids the scheduler's hot path by setting CPU limits from userspace, which are then enforced by CFS/EEVDF [46]. However, this approach has the overhead of context switches to the kernel space, as both the PEBS and the FPGA counters are exposed via procfs/debugfs files by the kernel (§ 5.1.4 and § 5.3.1). To improve the time spent per loop iteration as well as the reaction speed, asynchronous IO could be used to read counter-related files and write cgroup-related files concurrently.

However, the main limiting factor for the reaction speed of the cgroups-based CBC approach is the CPU limitation feature of cgroups itself. As described in Section 5.4.2, for a CPU limit of 1 %, the allowed CPU time has to be set to 1 ms in a given period of 100 ms, because 1 ms is the minimum value supported by cgroups [46]. Hence, updating the CPU limits faster than every 100 ms may reset the already used CPU time of the current period for a given process. This is why we set the interval of executing the CBC logic to 150 ms for the cgroups-based approach (§ 5.4.2). Lowering the minimum amount of time accepted by cgroups likely requires modifications to cgroups and to the CFS/EEVDF scheduler in the form of a kernel patch, which involves considerable effort. We, therefore, see only limited future optimization potential of the cgroups-based approach.

7.5 Evaluation

Our evaluation did not measure energy consumption, which is supposed to be reduced by avoiding CPU stalls. Although the momentary power consumption with CXL bandwidth control is likely to be higher during a benchmark than without it because of the better CPU utilization, a given workload should finish faster. By avoiding static energy consumption during CPU stalls, we would expect an overall reduction in energy consumption. Future work could measure the consumed energy using the power information provided by modern Intel CPUs via the Running Average Power Limit (RAPL) [35:§16.10.3, 98].

In our evaluation, we only used 5 or 10 fio processes to generate load. Because of the low bandwidth of our CXL FPGA of about 10 GiB/s, using more processes would require each process to have an even smaller CPU limit to reach the target bandwidth. Measuring and controlling the single-digit CPU usage percentages of programs becomes harder, as small inaccuracies can result in big changes in the aggregate bandwidth. Therefore, a more powerful FPGA or even an ASIC, as well as an improved design resulting in higher maximum CXL bandwidth, would allow to evaluate the approach with more concurrency on the system.

In the evaluation regarding CXL memory access latencies (§ 6.5), latencies were in the range of 5 til 20 ms. This is caused by the block size of 4 MiB/s, as the CXL hardware has to perform 65 536 individual cache line accesses of 64 bytes for one such block. Future work could experiment with different block sizes to study their effect on the CBC approaches. Also, the applicability of our approach regarding CXL hybrid SSDs could be studied, as they may have different performance characteristics.

To analyze the reaction speed of the individual CBC approaches, it needs to be investigated why the bandwidth reported by fio oscillates between 0 and the final bandwidth at the start of each measurement, as mentioned in Section 6.3 and shown in Figure 9. Otherwise, the effect of bandwidth changes on fio is not clear, which could distort such measurements.

In general, the real-world applicability and usability of our approach could be examined. A successful use of the CBC approach in the wild requires sufficient system

load and a combination of memory-bound and compute-bound processes, so that the scheduler has a selection which process to schedule next.

Chapter 8

Conclusion

In this thesis, we implemented memory access counters on a CXL FPGA in combination with PEBS-based access sampling to estimate the CXL bandwidth used by each process. To keep CXL memory access latencies low and prevent queuing delay, we throttled the CPU time of individual tasks to reduce the CXL bandwidth. We implemented two approaches for enforcing the CPU limit, both with their own advantages and disadvantages. In our evaluation, we showed that they improve the CXL memory access latency, lower the CPU cost by up to 40 %, and increase the CPU throughput by 6.7 %. While the resulting CXL bandwidth is rather stable for the cgroups-based approach, this approach is limited in its reaction speed. The `sched_ext`-based approach has difficulties in meeting the target bandwidth stably, but has more room for future optimization, as it has more control over scheduling decisions. Nevertheless, it still reduces the CXL access latency by around 30 %. In general, the CXL bandwidth control can improve the CXL access characteristics of a system but requires a sufficient workload combination of memory-bound and compute-bound applications.

Bibliography

[1] Advanced Micro Devices (AMD). 2023. 4TH GEN AMD EPYC PROCESSOR ARCHITECTURE. Retrieved from https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/221704010-B_en_4th-Gen-AMD-EPYC-Processor-Architecture%E2%80%94White-Paper_pdf.pdf

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. *SIGPLAN Not.* 52, 4 (April 2017), 631–644. Retrieved from https://doi.org/10.1145/3093336.3037706

[3] Soramichi Akiyama and Takahiro Hirofuchi. 2017. Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017 (ROSS '17)*, 2017. Association for Computing Machinery, Washingon, DC, USA. Retrieved from https://doi.org/10.1145/3095770.3095773

[4] Yuda An, Shushu Yi, Bo Mao, Qiao Li, Mingzhe Zhang, Ke Zhou, Nong Xiao, Guangyu Sun, Xiaolin Wang, Yingwei Luo, and Jie Zhang. 2024. A Novel Extensible Simulation Framework for CXL-Enabled Systems. Retrieved from https://arxiv.org/abs/2411.08312

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2023. *Operating Systems: Three Easy Pieces* (1.10 ed.). Arpaci-Dusseau Books.

[6] Jens Axboe and Vincent Fu. 2017. fio - Flexible I/O tester. Retrieved from https://fio.readthedocs.io/

[7] The Aya Contribtutors. 2022. Aya. Retrieved from https://aya-rs.dev/

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *(DAC '12)*, 2012. Association for Computing Machinery, San Francisco, California, 1216–1225. Retrieved from https://doi.org/10.1145/2228360.2228584

[9] Georgios Bitzes and Andrzej Nowak. 2014. The overhead of profiling using PMU hardware counters. *CERN openlab report* (2014), 1–16. Retrieved from https://core.ac.uk/download/pdf/30440122.pdf

[10] Daniel P. Bovet and Marco Cesati. 2006. *Understanding the Linux kernel* (3rd ed.). O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472. Retrieved from https://www.oreilly.com/library/view/understanding-the-linux/0596005652/

[11] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. 2019. Project PBerry: FPGA Acceleration for Remote Memory. In *(HotOS '19)*, 2019. Association for Computing Machinery, Bertinoro, Italy, 127–135. Retrieved from https://doi.org/10.1145/3317550.3321424

[12] Jonathan Cameron. 2023. perf: CXL 3.0 Performance Monitoring Unit support. Retrieved from https://lwn.net/Articles/928792/

[13] ChipsAlliance. Chisel: Software-defined hardware. Retrieved from https://www.chisel-lang.org/

[14] Albert Cho, Anish Saxena, Moinuddin Qureshi, and Alexandros Daglis. 2023. A Case for CXL-Centric Server Processors. Retrieved from https://arxiv.org/abs/2305.05033

[15] Compute Express Link Consortium, Inc. 2023. Compute Express Link (CXL): Specification. Revision 3.1. *CXL Consortium White Paper* (2023). Retrieved from https://computeexpresslink.org/cxl-specification/

[16] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux device drivers* (3rd ed.). O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472. Retrieved from https://lwn.net/Kernel/LDD3/

[17] Jonathan Corbet. 2009. DebugFS. Retrieved from https://docs.kernel.org/filesystems/debugfs.html

[18] Jonathan Corbet. 2021. Calling kernel functions from BPF. (2021). Retrieved from https://lwn.net/Articles/856005/

[19] Intel Corporation. 2024. PerfMon Events. Retrieved from https://perfmon-events.intel.com/

[20] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. 56, 11 (July 2024). Retrieved from https://doi.org/10.1145/3669900

[21] eBPF.io authors. 2025. eBPF - Introduction, Tutorials & Community Resources. Retrieved from https://ebpf.io/

[22] Eunomia. eBPF Tutorial by Example: Learning CO-RE eBPF Step by Step. Retrieved from https://eunomia.dev/en/tutorials

[23] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020. USENIX Association, 281–297. Retrieved from https://www.usenix.org/conference/osdi20/presentation/fried

[24] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: a tool to instrument x86-64 TLB misses. *SIGARCH Comput. Archit. News* 42, 2 (September 2014), 20–23. Retrieved from https://doi.org/10.1145/2669594.2669599

[25] Joseph Greathouse. 2019. AMD Research Instruction Based Sampling Toolkit. Retrieved from https://github.com/jlgreathouse/AMD_IBS_Toolkit

[26] Daniel Habicht. 2025. Efficient PMEM Accounting.

[27] Barry Hembree, Jack Prins, Pat Spagon, Paul Tobias, and Chelli Zey. 2003. NIST/SEMATECH e-Handbook of Statistical Methods. Retrieved from https://doi.org/10.18434/M32189

[28] John L Hennessy and David A Patterson. 2017. *Computer Architecture: A Quantitative Approach* (6th ed.). Elsevier. Retrieved from https://archive.org/details/computerarchitectureaquantitativeapproach6thedition

[29] Andrew J Herdrich, Marcel David Cornu, and Khawar Munir Abbasi. 2019. Introduction to Memory Bandwidth Allocation. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html

[30] Intel Corporation. 2020. MBM MBA how to guide. Retrieved from https://github.com/intel/intel-cmt-cat/wiki/MBM-MBA-how-to-guide

[31] Intel Corporation. 2023. Intel Xeon Silver 4416+ Processor. Retrieved from https://www.intel.com/content/www/us/en/products/sku/232378/intel-xeon-silver-4416-processor-37-5m-cache-2-00-ghz/specifications.html

[32] Intel Corporation. 2024. Which Generation of Intel Xeon Scalable Processors Support Compute Express Link (CXL) and PCIe 5.0?. Retrieved from https://www.intel.com/content/www/us/en/support/articles/000059219/processors.html

[33] Intel Corporation. 2024. What Is the Compute Express Link* (CXL) Support on the Intel® Server M50FCP and D50DNP Families?. Retrieved from https://www.intel.com/content/www/us/en/support/articles/000098171/server-products.html

[34] Intel Corporation. 2024. Intel 64 and IA-32 Architectures Optimization Reference Manual Volume 1 (Version 49). Retrieved from https://www.intel.com/content/www/us/en/content-details/814198/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html

[35] Intel Corporation. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[36] Intel Corporation. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[37] Intel Corporation. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[38] Intel Corporation. 2025. Quartus® Prime Design Software. Retrieved from https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

[39] Intel Corporation. 2025. Intel Performance Counter Monitor (Intel PCM). Retrieved from https://github.com/intel/pcm

[40] Intel Corporation. 2025. Agilex™ 7 FPGA I-Series Development Kit (2x R-Tile and 1x F-Tile). Retrieved from https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html

[41] Intel Corporation. 2025. Intel Resource Director Technology (Intel RDT) Architecture Specification. Retrieved from https://cdrdv2-public.intel.com/789566/356688-intel-rdt-arch-spec.pdf

[42] IO Visor Project. 2025. BPF Compiler Collection~(BCC). Retrieved from https://github.com/iovisor/bcc

[43] Myoungsoo Jung. 2022. 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 (HotStorage '22)*, 2022. Association for Computing Machinery, Virtual Event, 45–51. Retrieved from https://doi.org/10.1145/3538643.3539745

[44] Bryce Kahle. 2020. How and When You Should Measure CPU Overhead of eBPF Programs. Retrieved from https://ebpf.io/summit-2020-slides/eBPF_Summit_2020-Lightning-Bryce_Kahle-How_and_When_You_Should_Measure_CPU_Overhead_of_eBPF_Programs.pdf

[45] The kernel development community. 2022. linux/arch/x86/events/intel.h. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/events/intel?h=v5.4.209

[46] The kernel development community. 2023. CFS Bandwidth Control. Retrieved from https://docs.kernel.org/scheduler/sched-bwc.html

[47] The kernel development community. 2024. Kernel modules. Retrieved from https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html

[48] The kernel development community. 2024. CFS Scheduler. *Linux Kernel Documentation* (2024). Retrieved from https://docs.kernel.org/scheduler/sched-design-CFS.html

[49] The kernel development community. 2024. Scheduler Statistics. *Linux Kernel Documentation* (2024). Retrieved from https://docs.kernel.org/scheduler/sched-stats.html

[50] The kernel development community. 2025. DAMON: Data Access MONitor — The Linux Kernel documentation. *Linux Kernel documentation* (2025). Retrieved from https://docs.kernel.org/mm/damon/index.html

[51] The kernel development community. 2025. Control Group v2 — The Linux Kernel documentation. *Linux Kernel documentation* (2025). Retrieved from https://docs.kernel.org/admin-guide/cgroup-v2.html

[52] The kernel development community. 2025. BPF Documentation. *Linux Kernel documentation* (2025). Retrieved from https://docs.kernel.org/bpf/index.html

[53] The kernel development community. 2025. libbpf Overview. Retrieved from https://www.kernel.org/doc/html/latest/bpf/libbpf/libbpf_overview.html

[54] The kernel development community. 2025. The kernel's command-line parameters. Retrieved from https://docs.kernel.org/admin-guide/kernel-parameters.html

[55] The kernel development community. 2025. Upgrading ACPI tables via initrd. Retrieved from https://docs.kernel.org/admin-guide/acpi/initrd_table_override.html

[56] The kernel development community. 2025. Kernel Build System. Retrieved from https://docs.kernel.org/kbuild/index.html

[57] The kernel development community. 2025. BPF Kernel Functions (kfuncs). *Linux Kernel documentation* (2025). Retrieved from https://docs.kernel.org/bpf/kfuncs.html

[58] The kernel development community. 2025. CXL Performance Monitoring Unit (CPMU). *Linux Kernel documentation* (2025). Retrieved from https://docs.kernel.org/admin-guide/perf/cxl.html

[59] The kernel development community. 2025. kernel/git/torvalds/linux.git - Linux kernel source tree. Retrieved from https://web.git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v6.12

[60] Yussuf Khalil. 2025. Chisel FPGA implementation of a CXL memory expander.

[61] Colin Ian King. 2020. man 1 stress-ng: stress "next generation", a tool to load and stress a computer system. Retrieved from https://manpages.ubuntu.com/manpages/focal/man1/stress-ng.1.html

[62] Andi Kleen. 2025. pmu-tools. Retrieved from https://github.com/andikleen/pmu-tools

[63] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In *(HotStorage '23)*, 2023. Association for Computing Machinery, Boston, MA, USA, 24–30. Retrieved from https://doi.org/10.1145/3599691.3603406

[64] Yunjae Lee, Yoonhee Kim, and Heon Y Yeom. 2020. Lightweight memory tracing for hot data identification. *Cluster Computing* 23, (2020), 2273–2285. Retrieved from https://doi.org/10.1007/s10586-020-03130-1

[65] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*, 2023. Association for Computing Machinery, Vancouver, BC, Canada, 574–587. https://doi.org/10.1145/3575693.3578835

[66] The libbpf authors. 2025. libbpf/libbpf: Automated upstream mirror for libbpf stand-alone build. Retrieved from https://github.com/libbpf/libbpf

[67] The Linux Kernel Documentation Team. 2023. man 1 cpio: copy files to and from archives. Retrieved from https://linux.die.net/man/1/cpio

[68] The Linux Kernel Documentation Team. 2024. man 2 bpf: perform a command on an extended BPF map or program. Retrieved from https://man7.org/linux/man-pages/man2/bpf.2.html

[69] The Linux Kernel Documentation Team. 2024. man 7 bpf-helpers: list of eBPF helper functions. Retrieved from https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[70] The Linux Kernel Documentation Team. 2025. Extensible Scheduler Class. Retrieved from https://docs.kernel.org/scheduler/sched-ext.html

[71] The Linux Kernel Documentation Team. 2025. man 1 perf-mem: Profile memory accesses. Retrieved from https://man7.org/linux/man-pages/man1/perf-mem.1.html

[72] The Linux Kernel Documentation Team. 2025. man 5 proc_pid_io: /proc/pid/io - I/O statistics. Retrieved from https://man7.org/linux/man-pages/man5/proc_pid_io.5.html

[73] The Linux Kernel Documentation Team. 2025. man 1 nice: run a program with modified scheduling priority. Retrieved from https://man7.org/linux/man-pages/man1/nice.1.html

[74] The Linux Kernel Documentation Team. 2025. man 7 cgroups: Linux control groups. Retrieved from https://man7.org/linux/man-pages/man7/cgroups.7.html

[75] The Linux Kernel Documentation Team. 2025. man 8 mount: mount a filesystem. Retrieved from https://man7.org/linux/man-pages/man8/mount.8.html

[76] The Linux Kernel Documentation Team. 2025. man 1 time: run programs and summarize system resource usage. Retrieved from https://man7.org/linux/man-pages/man1/time.1.html

[77] The Linux Kernel Documentation Team. 2025. man 8 bpftool: tool for inspection and simple manipulation of eBPF progs. Retrieved from https://manpages.debian.org/experimental/bpftool/bpftool.8.en.html

[78] The Linux Kernel Documentation Team. 2025. man 8 bpftool-prog: tool for inspection and simple manipulation of eBPF progs. Retrieved from https://manpages.debian.org/experimental/bpftool/bpftool-prog.8.en.html

[79] Linux perf wiki Contributors. 2024. perf: Linux profiling with performance counters. Retrieved from https://perfwiki.github.io/

[80] Linux perf wiki Contributors. 2024. man 1 perf-sched: Tool to trace/measure scheduler properties (latencies). Retrieved from https://man7.org/linux/man-pages/man1/perf-stat.1.html

[81] The LLVM Project. CIRCT: Circuit IR Compilers and Tools. Retrieved from https://circt.llvm.org/

[82] Hasan Al Maruf and Mosharaf Chowdhury. 2023. Memory Disaggregation: Advances and Open Challenges. *SIGOPS Oper. Syst. Rev.* 57, 1 (June 2023), 29–37. Retrieved from https://doi.org/10.1145/3606557.3606562

[83] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*, 2023. Association for Computing Machinery, Vancouver, BC, Canada, 742–755. Retrieved from https://doi.org/10.1145/3582016.3582063

[84] Marvell. 2024. Marvell Introduces Breakthrough Structera CXL Product Line to Address Server Memory Bandwidth and Capacity Challenges in Cloud Data Centers. Retrieved from https://www.marvell.com/company/newsroom/marvell-introduces-breakthrough-structera-cxl-product-line-to-address-server-memory-bandwidth-and-capacity-challenges-in-cloud-data-centers.html

[85] Marvell. 2025. CXL Near-Memory Compute and Expansion. Retrieved from https://www.marvell.com/products/cxl.html

[86] The Matplotlib development team. 2025. Matplotlib: Visualization with Python. Retrieved from https://matplotlib.org/

[87] Micron Technology, Inc. 2023. Micron's Perspective on Impact of CXL on DRAM Bit Growth Rate. Retrieved from https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-impact-dram-bit-growth-white-paper.pdf

[88] Micron Technology. 2021. MTA18ASF2G72PZ-3G2J3 part detail. Retrieved from https://www.micron.com/products/memory/dram-modules/rdimm/part-catalog/part-detail/mta18asf2g72pz-3g2j3

[89] Robert Moore and Mattia Dongili. 2013. man 1 iasl: iasl - ACPI Source Language compiler/decompiler. Retrieved from https://linux.die.net/man/1/iasl

[90] Robert Moore and Al Stone. 2013. man 1 acpixtract: ACPICA source code conversion utility. Retrieved from https://manpages.debian.org/unstable/acpica-tools/acpixtract-acpica.1.en.html

[91] Robert Moore, Chao Guan, and Al Stone. 2013. man 1 acpidump: ACPI table dump utility. Retrieved from https://manpages.ubuntu.com/manpages/jammy/man1/acpidump.1.html

[92] Norman S. Nise. 2011. *Control Systems Engineering* (7th ed.). John Wiley & Sons, Inc.

[93] NumPy team. 2025. NumPy: The fundamental package for scientific computing with Python. Retrieved from https://numpy.org/

[94] Atsuya Osaki, Manuel Poisson, Seiki Makino, Ryusei Shiiba, Kensuke Fukuda, Tadashi Okoshi, and Jin Nakazawa. 2024. Dynamic Fixed-point Values in eBPF: a Case for Fully In-kernel Anomaly Detection. In *Proceedings of the Asian Internet Engineering Conference 2024 (AINTEC '24)*, 2024. Association for Computing Machinery, Sydney, NSW, Australia, 46–54. Retrieved from https://doi.org/10.1145/3674213.3674219

[95] The pandas development team. 2025. pandas: Python Data Analysis Library. Retrieved from https://pandas.pydata.org/

[96] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, (2011), 2825–2830.

[97] Vinicius Petrucci, Felippe Zacarias, and David Roberts. 2025. A Limits Study of Memory-side Tiering Telemetry. *arXiv preprint arXiv:2508.09351* (2025). Retrieved from https://doi.org/10.48550/arXiv.2508.09351

[98] Guillaume Raffin and Denis Trystram. 2025. Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis. *IEEE*

Transactions on Parallel and Distributed Systems 36, 1 (2025), 96–107. https://doi.org/10.1109/TPDS.2024.3492336

[99] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021. Association for Computing Machinery, Virtual Event, Germany, 392–407. Retrieved from https://doi.org/10.1145/3477132.3483550

[100] Liz Rise. 2023. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media, Inc., Sebastopol, CA 95472. Retrieved from https://cilium.isovalent.com/hubfs/Learning-eBPF%20-%20Full%20book.pdf

[101] Alastair Robertson. 2025. bpftrace: Dynamic Tracing for Linux. Retrieved from https://bpftrace.org/

[102] Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, and Jim Huang. 2025. *The Linux Kernel Module Programming Guide*. Retrieved from https://sysprog21.github.io/lkmpg/

[103] Samsung Electronics. 2024. Samsung Demonstrates New CXL Capabilities and Introduces New Memory Module for Scalable, Composable Disaggregated Infrastructure at Memcon 2024. Retrieved from https://semiconductor.samsung.com/news-events/news/samsung-demonstrates-new-cxl-capabilities-and-introduces-new-memory-module-for-scalable-composable-disaggregated-infrastructure-at-memcon-2024/

[104] Samsung Semiconductor EMEA. 2024. Samsung CXL Solutions – CMM-H. Retrieved from https://semiconductor.samsung.com/emea/news-events/tech-blog/samsung-cxl-solutions-cmm-h/

[105] Thomas Schmidt. 2021. Achieving Optimal Throughput for Persistent Memory with Per-Process Accounting. (2021). Retrieved from https://os.itec.kit.edu/downloads/2022_MA_Schmidt_PMEM_Accounting.pdf

[106] Martin Schoeberl. 2019. *Digital Design with Chisel*. Kindle Direct Publishing. Retrieved from https://github.com/schoeberl/chisel-book

[107] Schol-R-LEA. 2018. Re: Get the performance monitoring interrupt on Qemu-Kvm. Retrieved from https://f.osdev.org/viewtopic.php?t=32793&sid=63d062d947a48af54a7fa52ef2b9e594

[108] scx developers. 2025. scx: sched_ext schedulers and tools. Retrieved from https://github.com/sched-ext/scx

[109] Debendra Das Sharma. 2022. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2022. 5–12. Retrieved from https://doi.org/10.1109/HOTI55740.2022.00017

[110] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*, 2025. Association for Computing Machinery, Rotterdam, Netherlands, 604–621. https://doi.org/10.1145/3676641.3711999

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

[112] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall Press, USA.

[113] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory.

In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*, 2024. Association for Computing Machinery, Athens, Greece, 818–833. Retrieved from https://doi.org/10.1145/3627703.3650061

[114] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, April 2018. USENIX Association, Renton, WA, 283–297. Retrieved from https://www.usenix.org/conference/nsdi18/presentation/tootoonchian

[115] UEFI Forum, Inc. 2021. ACPI Software Programming Model. Retrieved from https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/05_ACPI_Software_Programming_Model/ACPI_Software_Programming_Model.html

[116] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Sri Sakthivelu, and Sharanyan Srikanthan. 2024. Intel® Memory Latency Checker v3.11b. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[117] Lukas Werling, Daniel Habicht, and Frank Bellosa. 2023. Per-Process Memory Bandwidth Management for Heterogeneous Memory Systems. (2023). Retrieved from https://www.betriebssysteme.org/wp-content/uploads/2023/09/FGBS_2023_Bamberg_Folien_Werling.pdf

[118] Lukas Werling, Yussuf Khalil, Peter Maucher, Thorsten Gröninger, and Frank Bellosa. 2023. Analyzing and Improving CPU and Energy Efficiency of PM File Systems. In *(DIMES '23)*, 2023. Association for Computing Machinery, Koblenz, Germany, 31–37. Retrieved from https://doi.org/10.1145/3609308.3625265

[119] Matthew Wilcox and Alan Cox. 2025. Bus-Independent Device Accesses. Retrieved from https://www.kernel.org/doc/html/v6.6/driver-api/device-io.html

[120] Thomas Willhalm and Roman Dementiev. 2022. Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html

[121] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, July 2023. USENIX Association, Boston, MA, 601–617. Retrieved from https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[122] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. 2020. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20)*, 2020. Association for Computing Machinery, Tsukuba, Japan, 98–105. Retrieved from https://doi.org/10.1145/3409963.3410490

[123] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. 2022. MT²: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, February 2022. USENIX Association, Santa Clara, CA, 199–216. Retrieved from https://www.usenix.org/conference/fast22/presentation/yi-mt2

[124] Seung Won Yoo, Joontaek Oh, Myeongin Cheon, Bonmoo Koo, Wonseb Jeong, Hyunsub Song, Hyeonho Song, Donghun Lee, and Youjip Won. 2025. DJFS : Directory-Granularity Filesystem Journaling for CMM-H SSDs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, February 2025. USENIX Association, Santa Clara, CA, 35–51. Retrieved from https://www.usenix.org/conference/fast25/presentation/yoo

[125] Xiong Zhang. 2023. [PATCH v2 0/9] Upgrade intel vPMU version to 5. Retrieved from https://lore.kernel.org/all/20230921082957.44628-1-xiong.y.zhang@intel.com/

[126] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. 2024. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024. 1518–1531. https://doi.org/10.1109/MICRO61859.2024.00111

[127] zyantific. 2025. Zydis: The ultimate, open-source X86 & X86-64 decoder/disassembler library. Retrieved from https://zydis.re/