Karlsruhe Institute of Technology

Fine-Grained Bandwidth Monitoring for CXL Memory Devices

Bachelor's Thesis of

Gero Beckmann

at the
Department of Informatics
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.

25. March 2026 – 27. July 2026

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, 27.07.2026

Abstract

Todays high performance computing requires large memory capacities with low latency. These requirements can be fullfilled by using memory tiering solutions to optimize the latency of heterogenous memory systems, that solve the need for more memory capacity. Memory management strategies like memory tiering and Transparent DAX Mappings (TDM), which make direct access (DAX) mappings accessable for unaltered applications, rely on monitoring access frequency of data to determine whether to promote or demote said data to memory with less or more latency. Previous work that monitored the access frequency of data based on features of the operating system or the CPU show a trade-of between accuracy and performance impact, which we eliminate by using hardware-assisted bandwidth measurements of a custom FPGA-based Compute Express Link memory expander. Our implementation extends the Linux kernel to monitor the bandwidth for memory mappings and provide an interface of bandwidth measurements for monitored memory mapping to the userspace. To evaluate our approach, we implement a memory tiering application which promotes memory mappings to a fitting memory tier based on the measured bandwidth. We measured that our bandwidth measurement closely matches the actual bandwidth. Our memory tiering application uses these measurements to select the most fitting memory tier, which optimizes the overall bandwidth to beat just using the hot tier by 21% and interleaving each memory mapping across multiple memory tiers by 8%. We show that our bandwidth monitoring for memory mapping can be of advantage for applications like memory tiering.

Contents

Abstract **v**

Contents **vii**

1 Introduction **1**

2 Background **5**

2.1 Compute Express Link 5
2.2 Direct Access Mapping 7
2.3 Memory Tiering 8
2.4 Transparent DAX Mapping 8
2.5 NUMA 9
2.6 Monitoring Groups 9
2.7 PEBS 10

3 Related Work **11**

4 Design **15**

4.1 CXL Access Counter Device 16
4.2 Monitoring Group Allocator 18
4.3 Report Measurements 19

5 Implementation **21**

5.1 Configuration of Monitoring 21
5.2 NUMA node ID allocator 22
5.3 Migration to another NUMA node 23
5.4 Expose Bandwidth Information 24
5.5 Memory Tiering 29

6 Evaluation **35**

6.1 Evaluation Setup and Methodology 35
6.2 Bandwidth Measurement 38

6.3 Memory Tiering 40
6.4 Transparent DAX Mappings 43

7 Discussion and Future Work **45**
7.1 Shortcomings of the Allocator 45
7.2 Alternative Allocators 47
7.3 Shared Memory 49
7.4 Userspace Implementation 50

8 Conclusion **51**

Bibliography **53**

Chapter 1

Introduction

The rise of high performance computing, with requirements for large capacity memory with low latency, such as artificial intelligence and machine learning, made the demand for more memory capacity apparent by the general public through the ongoing high of memory prices [24]. Memory capacity grows slower than computing power, and DDR memory channels have not been keeping up with demand for high performance computing, which is known as the memory wall [36]. Compute Express Link (CXL) is a new high bandwidth and low latency cache-coherent open interconnect protocol which addresses the demand for more memory capacity amongst further benefits (Chapter 2.1) [18, 33]. CXL utilizes the same physical interface as Peripheral Component Interconnect Express (PCIe), which has a higher per-pin bandwidth than DIMM. Thus, devices such as CXL memory expanders and CXL-based hybrid SSDs can be used to extend the memory capacity and bandwidth over the physical PCIe interface beyond the limits set by the DRAM slots. However, due to the longer distance between the CPU and PCIe physical interface, in comparison to DRAM slots, CXL devices suffer from about double the latency. Memory latency is important since the CPU waits for the completion of the memory request by stalling, thus the throughput of the memory becomes a bottleneck of how fast a processor can operate, which is known as the Von-Neumann bottleneck [15]. The bottleneck can be mitigated through caching, prefetching and out-of-order execution, but falls short for applications doing many random memory accesses. Improving the overall memory latency improves the overall performance, especially for memory-bound applications.

The development of CXL has sparked research into memory management strategies that explore properties of CXL for its use in heterogeneous memory systems or different ways of addressing its memory. These memory management strategies rely on metrics like access frequency to promote or demote data so that it can be accessed with less or more latency or overhead. For this work, we will look at two memory management strategies. The first strategy, memory tiering, tries to optimize the overall latency of heterogeneous memory systems, in which memory has different latency, like DRAM together with CXL devices. Infrequently accessed pages should be put on high latency memory to keep space for frequently accessed pages on low latency memory. As a metric to determine whether to promote or demote a page to a memory device with lower or higher latency, memory tiering applications rely on detecting the access frequency of pages. The second strategy, Transparent DAX Mapping, tries to reduce operating system overhead when accessing files on a CXL-based Hybrid SSD [27]. CXL-based Hybrid SSDs offer block-addressable storage with a byte-addressable cache. Using direct access (DAX) mappings, files can be accessed without traditional read/write syscalls and circumventing the page cache and operating system, and instead access the contents of the file directly over the byte-addressable cache [27]. Transparent DAX Mappings make it possible to use DAX mappings on unmodified applications by linking to modified read/write functions that internally use DAX mappings to access the file [27]. The usage of Transparent DAX Mappings has to be managed by promoting and demoting files as to not overwhelm the storage device with memory accesses and thus increasing the latency [27].

The challenge of both memory tiering applications and Transparent DAX Mappings is monitoring the access frequency of data as a metric on promoting or demoting said data. Previous methods of the CPU or the operating system of detecting access frequency have a trade-of between accuracy and performance impact (Chapter 3).

In this work, we extend the Linux kernel to monitor the bandwidth per memory mapping using hardware-assisted measurements of a custom CXL memory expander and provide and evaluate these measurements as a metric for memory management strategies like memory tiering or Transparent DAX Mappings. To limit the performance impact while keeping great accuracy, we take hardware assisted bandwidth measurements from monitoring groups proposed by Kämmerer, 26. Monitoring groups are seen as virtual memory-only NUMA nodes by the operating system, and

each shows a view of the device's memory while counting all memory accesses and exposing the counters back to the operating system. We extend the Linux kernel such that whenever an application asks the operating system for a direct access mapping, the operating system assigns that mapping to a free monitoring group. By having one monitoring group per mapping, each monitoring group can accurately count all memory accesses that are done through that mapping. Because the memory accesses are counted on the custom FPGA-based CXL memory expander, we do not suffer from much CPU overhead.

By monitoring the bandwidth for memory mappings, we provide an accurate metric for the access frequency of data by which memory tiering applications and the TDM Manager can decide on promoting or demoting a mapping. This improves bandwidth and reduces latency for heterogeneous memory systems. To evaluate this, we extend the QEMU virtual machine to emulate a heterogeneous tiered memory system and implement our own memory tiering application.

In the following, we give more detailed information about the technologies and methods at stake in Chapter 2. Chapter 3 compares related work. In Chapter 4, we explain our general approach of how a monitoring group is allocated for a monitored memory mapping and how the bandwidth is reported. Then, we show the implementation of our approach in Chapter 5, which introduces new flags to syscalls, the NUMA Node ID allocator, reports the bandwidth to userspace and a memory tiering application. After the implementation, we evaluate our implementation in in Chapter 6 by comparing the measurement against a generated bandwidth and use the measurements for our memory tiering application, which we compare against interleaving a memory mapping across multiple memory tiers. Finally, we conclude our findings in Chapter 8.

Chapter 2

Background

In this chapter we explain the technology we build up on. First, we summary the specification of Compute Express Link in Chapter 2.1. Second, we explain the advantages of direct access mappings in Chapter 2.2. Third, we explain memory tiering in Chapter 2.3. Four, we build up upon DAX mappings to explain transparent DAX mapping in Chapter 2.4. Five, we explain the Non-uniform memory access system in Chapter 2.5. Six, we summarize monitoring groups in Chapter 2.6. And seven, we give a short definition of process event based sampling in Chapter 2.7.

2.1 Compute Express Link

Compute Express Link (CXL) is a new high bandwidth and low latency cache-coherent open interconnect standard [18]. It is defined by the CXL Consortium consisting of many major industry corporations like Alibaba, Cisco, Dell, Google, Huawei, Meta, Microsoft and HPE. It is an afford create a united standard that replaces proprietary and solutions that were not cross-compatible like “Intel’s Quick Path Interconnect (QPI) and Ultra Path Interconnect (UPI) , AMD’s Infinity Fabric, IBM’s Bluelink, and Nvidia’s NVLink” [33:§2]. CXL uses the same physical interface as PCIe and builds upon the PCIe specification and is as such backwards-compatible with PCIe devices [33:§2]. Support for CXL has to be integrated in the CPU as is the case with Intel Sapphire Rapids and AMD Genoa and Bergamo CPUs [33:§6].

It allows to plug in memory, storage and expansion devices, like DRAM, GPUs, NICs or persistent memory for High Performance Computing (HPC). The CXL 3 specification also adds the possibility to share access to CXL devices across multiple

CPUs and multiple Hosts called pooling, even thou as of this writing there is no hardware support. This allows a server with low utilization to share its resources with a server with a higher utilization, which is useful in applications like "web search, social network content composition, and advertisement selection" [33].

The usage of CXL memory expanders is especially interesting as memory capacity and bandwidth of DDR channels hardly can keep with the demands of High Performance Computing [29, 33, 36]. This problem is known as the memory wall [36]. CXL memory expanders solve this problem by adding memory into the coherent address space and having a higher bandwidth per physical connection than the DDR interface while only having double the latency compared to DDR [33, 36]. These characteristics of CXL memory expanders make them a great target for memory tiering solutions [26, 29, 32].

To allow the range of memory, storage and expansion devices CXL implements three protocols.

CXL.io extends PCIe and is used for Accelerators such as GPUs or FPGAs. This protocol is also used for device discovery, status reporting, virtual to physical address translation and direct memory access [33:§3.1].

CXL.mem allows for cache coherent memory access by the CPU/host to memory on the CXL device. This allows to extend the RAM with CXL. Memory access is cached by the CPU such that the latency of getting the value can be decreased if the value is cached. CXL.mem devices are mapped in the address space and accessed with `load`/`store` instructions/semantics by the CPU.

CXL.cache allows the CXL device to access data from the CPUs cache and RAM.

These three protocols can be combined to form the types of CXL devices.

Type 1 devices use the CXL.io and CXL.cache protocols. They access and cache host memory. This is useful for devices like smart network interface controllers (smart NIC).

Type 2 devices use the CXL.io, CXL.cache, and CXL.mem protocols. They access and cache host memory and make their memory available to the host. This is useful for devices like GPUs or FPGAs.

Type 3 devices use the CXL.io and CXL.mem protocols. These devices are used to expand the bandwidth and capacity with different types of memory. An example would be hybrid SSD which combine byte addressable DRAM and block addressable memory. (used for memory tiering) [33]

The custom Type 3 memory expander is implemented on an Altera Agilex FPGA [14]. The implementation is based on the design by Khalil, 28 and is expanded with Monitoring Groups by Kämmerer, 26 (Chapter 2.6).

2.2 Direct Access Mapping

Traditionally when accessing a file from storage, the file contents get dynamically loaded into the page cache and written back to storage later (when the file is closed or flush). This improves latency since the latency for random access on block memory is high. So loading the file in the page cache in RAM once and working on the cached contents is faster. The block storage is only accessed when the next page is read or the file is closed and needs to be stored back to non-volatile block storage. The page cache is also used for memory mapped filed using mmap.

Memory that is byte-addressable by the CPU does not need to be copied to the page cached since we can read and write directly on the storage device. To use such byte-addressable memory Linux provides two interfaces.

The devdax interface is used to map arbitrary portions of storage device into address space.

The fsdax interface is used to memory map files from a file system. As of now only a few file systems are supported, like ext2, ext4, xfs, virtiofs and erofs. Support for DAX mapping a file can be enabled per inode, file or directory [2, 22].

2.3 Memory Tiering

Memory tiering puts different byte-addressable memory devices in different levels based on latency. Then data can be allocated or moved to the device that best matched the required characteristics for the application. Frequently used "hot" pages are best allocated on memory with low latency. To keep space on low latency memory infrequently used "cold" pages are best allocated on memory with higher latency [19].

Generally there are two types how an application might use memory tiering. With explicit memory tiering the application has some control over where the data is allocated. The developer might know beforehand that the application needs fast access to data and thus can communicate that. With transparent memory tiering the application is not aware that there are multiple memory devices. A memory tiering system decides where to allocate and move data in the background without the application knowing or any configuration needed [32].

2.4 Transparent DAX Mapping

Transparent DAX Mapping is an approach to use byte-addressable storage, like CXL-based Hybrid SSDs, for unaltered applications, that use traditional POSIX I/O interface. This sidesteps the kernel together with the page cache and directly uses the storage device through userspace DAX mappings. Since page caches are sidestepped, the usage of DRAM memory is reduced, and instead the cheaper Hybrid SSDs are utilized. To balance latency and utilization of the storage device, the TDM manager promotes and demotes files to use TDMs or revert back to POSIX I/O [27].

The DAX mapping is done transparently such that the application is not aware of this happening and does not have to be rewritten in a way to be able to use this mechanism. The mechanism is implemented as two components. The first component, `libtdm`, is loaded into a process using `LD_PRELOAD` and is responsible to transparently hook into the `open`, `read` and `write` functions of the `libc` to use DAX mappings instead. The second component is the `ktdm` kernel module, which is responsible to write back dirty contents when the process exists and resizes the file. A TDM manager in the kernel is responsible for deciding on promoting and demoting files [27].

2.5 NUMA

Non-uniform memory access system consists of cells which contain memory. Cells are accessible by any CPU. The point of NUMA is to model memory with different amounts of latency. Access to local memory of a CPU core has a lower latency than access to remote memory of another CPU core. The information might be used to have allocated memory close to a task scheduled on a CPU core so that the latency is minimal.

Linux abstracts the cells of the NUMA system to nodes to hide architecture details. NUMA nodes can be split into many emulated nodes that each only see a part of the cells physical memory [3].

The monitoring groups of the custom CXL memory expander are represented as virtual-memory only NUMA nodes.

2.6 Monitoring Groups

Monitoring groups have been proposed by Kämmerer as an extension of a custom CXL memory expander design to monitor bandwidth per process [26:§4]. His approach is implemented on an Altera Agilex 7 I-Series FPGA [14] and extends the CXL memory expander design by Khalil, 28. The proposed method gives each process its own view of the memory, called monitoring group, which each count every memory access done by a process.

The bandwidth monitoring per process is implemented by binding each process to their own virtual-memory only NUMA node. Each NUMA node represents a monitoring group, which is a chunk of the exposed host-physical address space of the custom CXL memory expander [26:§4]. The CXL memory expander then attributes every memory address access to a monitoring group by its upper bits and counts every read and write. The lower bits get used to access the device-physical address space of the physical RAM of the memory expander [26:§5.1.2]. The read and write counters of each monitoring group are exposed as a memory-mapped IO area, which then gets periodically read by a custom kernel module.

The proposed method of mapping every monitoring groups to all of the device-physical address space resulted in access conflicts because of corrupt data as the kernel page allocator is not aware that an access from different host-physical addresses can

change the same device-physical address. To solve this the work proposes mapping each monitoring group to a smaller region of the device-physical address space that does not overlap with another mapping, thereby shrinking the usable memory per NUMA node and making the memory of one process inaccessable to a process that uses an other NUMA node [26:§5.2].

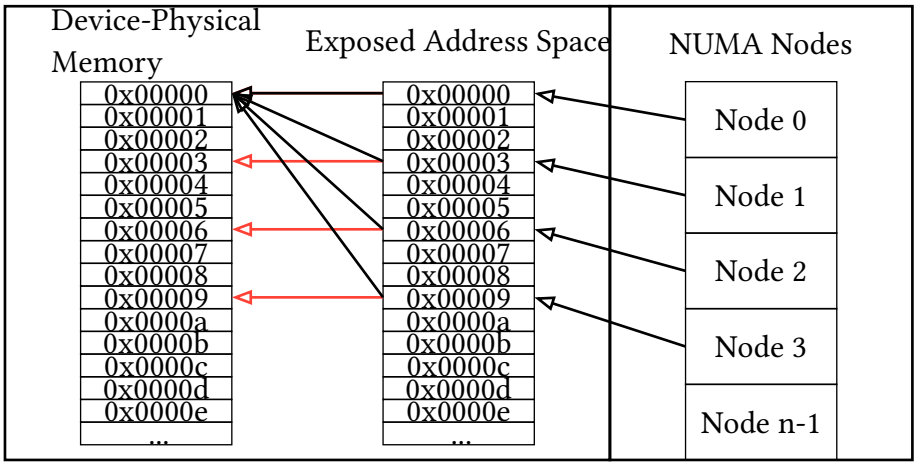


Figure 1: Overview of monitoring groups

2.7 PEBS

Process Event Based Sampling is a technique that uses the Performance Counter Monitor (PMU) of Intel CPUs to count events such as how often a read/write instruction is executed. When the counter reaches an overflow, an interrupt is triggered, allowing to disassemble the instruction and finding the address that is being written to/read from. This address than can be looked up in the page table to determine whether the address belongs to DRAM or a CXL device.

Chapter 3

Related Work

Previous work to monitor the access frequency of memory regions can be categorized in four methods, namely scanning page table entries (PTE), monitoring hinting page fault, sampling accessed memory addresses using the Performance Monitoring Unit (PMU) and hardware-assisted monitoring. Many proposed solutions of these methods suffer from various issues like performance overhead and limited precision. In this chapter we will explore related work and how it compares to our proposed solution.

The method of PTE-scanning relies on the access bit that is present in each page table entry and gets set once the page gets accessed. The access frequency of each page can then by detected by periodically walking the page table and checking if the access bit is set and clearing it. Here the resolution is limited by the page size and the frequency of the scans, which also has to be evaluated against the performance impact of walking the page table. The Linux kernel already implements this through Data Access MONitor (DAMON) [1], which builds of the work from Park et al., 30.

The method of hint-fault monitoring poisons page table entries it wants to monitor to trigger protection faults when the page gets accessed. The page protection fault handler is then used to count the page access, thus removing the need for walking the page table. The performance impact of page protection fault interrupts gets worse than page table walks for more page accesses. This is implemented in the tool BadgerTrap [21].

The method of PMU sampling relies on the Performance Monitoring Unit of the CPU, such as Intel's Processor Event-Based Sampling (PEBS) and AMD's Instruction Based Sampling (IBS). These can be used to track last level cache (LLC) misses

or the number of load/store instructions executed. The frequency of sampling the counters of these events also has to be evaluated against performance impact. PEBS-based sampling is explored in the work of transparent page placement (TPP) [29] by Maruf et al.. In their work they also look at memory tiering involving CXL-based memory expanders and present their tool Chameleon for hot-page detection. Chameleon uses PEBS-based sampling to generate a heat-map of memory usage [29]. Kämmerer uses PEBS-based sampling in extension to monitoring groups. As the number of processes can far exceed the static number of monitoring groups, many processes can share one monitoring group. The approximate PEBS-based bandwidth measurement per process is used to partition the accurate measurement of the monitoring group back to each individual process of said monitoring group [26]. Another feature that is integrated in Intel processors is Memory Bandwidth Monitoring (MBM) as part of Resource Director Technologies [23]. It provides L3 Total and Local External Memory Monitoring for threads/apps/VMs/containers. The operating system can assign a Resource Monitoring ID (RMID) to a thread it wants to monitor using MBM. Using Region Aware MBM [23:§3.1.4.2] the monitoring of a RMID can simultaneously measure the bandwidth utilization for each memory device. This limits the precision to be only on level of individual devices and threads, which makes it improbable to be used as a metric for memory tiering applications.

The method of hardware-assisted monitoring extends the memory device itself to be able to monitor the memory requests and expose the measurements to the host. The FPGA-based CXL memory device from Sun et al., 34 implements Page Access Counters (PAC) and Word Access Counters (WAC) for use as metrics for memory tiering applications. The limiting factor is that the number of PAC/WAC counters is static. A similar FPGA-based CXL memory device from [37, 37] analyzes each physical page access and exposes information about page hotness, memory bandwidth utilization, read/write ratio and access frequency. HeteroMem [17] is an approach that is similar to monitoring groups by Kämmerer, in that it also does a translation between host physical addresses to device-physical addresses and emulating memory devices that count memory requests. But in comparison to monitoring groups it works the other way around, exposing a heterogeneous two-tier memory system as a singular memory device to the host. The measurements of each memory tier are used for an device internal page migration strategy. Thus the measurements are not exposed to the host, but the host does not have to do any page migration themselves either. This

has the obvious limitation that host-attached DDR, with lower latency compared the CXL device, is not part of the page migration strategy by HeteroMem. The consensus amongst hardware-assisted monitoring seems to be that it should be the preferred method for bandwidth monitoring as no computation of the CPU is required to count the access frequencies, but careful consideration needs to be applied in utilizing the static amount of access counters.

Chapter 4

Design

In this chapter, we provide the overall design of our work and explain conceptionally how each component works and how they interact with each other. For a short overview, we first explain broadly how all components fit together and then go on explaining each component in more detail.

Memory mappings can be selected for monitoring and then get bound to a memory device, that implements hardware-assisted measurement of load and store instructions, effectively measuring how often a userspace application accesses memory through the memory mapping. Since only one mapping is allowed on one memory device, so that the hardware-assisted measurements of that device only account for the selected mapping, an allocator is needed that selects a device that is not in use by any memory mapping. The memory devices may have different access latencies, that results in CPU stalls. To minimize this the memory devices are categorised in different tiers based on their access latency. The allocator allows selecting devices from a specified tier, such that a userspace application may decide to promote or demote a memory tiering based on the bandwidth it is accessing the memory with. Once a free memory device is selected by the allocator, the mapping needs to be bound to the memory device and all data of the memory mapping needs to be migrated over. The hardware-assisted measurements are periodically polled to calculate the bandwidth that the userspace application is accessing the memory with. The tier, total memory accesses and current bandwidth utilization of each monitored memory mapping is reported for each application to userspace. We now go into a detailed explanation of the design of each component.

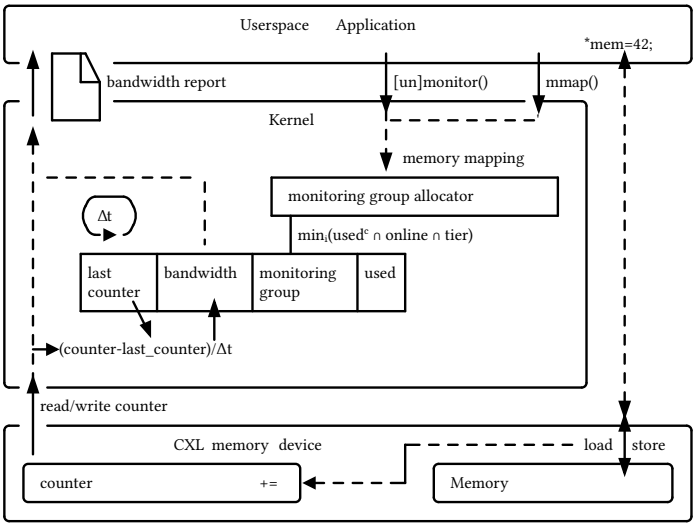


Figure 2: Component diagram showing the interfaces and data flow of the userspace application, kernel and CXL memory device. The monitoring group allocator selects an unused monitoring group, when a memory mapping is selected to be monitored. The bandwidth gets periodically calculated using the counters of the monitoring group from the CXL memory device, which increase on memory access through the monitored memory mapping.

4.1 CXL Access Counter Device

Our approach uses a custom CXL memory expander design [26] implemented on a Altera Agilex FPGA [14] and as a virtual device for the QEMU virtual machine, that provides hardware-assisted bandwidth measurement. We use a patched version of the QEMU virtual machine, that includes an implementation for the CXL-access-counter device, for easier access to that device. Using a virtual machine instead of using real hardware, allows for use of debugging tools and is easier to use and restart from a working remotely and to add a latency feature to the device, that is not yet implemented on the FPGA implementation.

The custom CXL memory expander exposes multiple memory devices, called monitoring groups, to the operating system. The monitoring groups are internally mapped to physical memory by the custom CXL memory expander. The operating system can register these monitoring groups as virtual NUMA nodes, that do not

have a CPU but only memory. Each monitoring group counts the load and store memory accesses and exposes the counters via a memory-mapped IO area. These counters are frequently read by a kernel module to determine the bandwidth (see Chapter 2.6). This custom CXL memory expander has previously been used to measure the bandwidth per process [26].

In this work we use the custom memory expander to measure the bandwidth for memory mappings. To measure the bandwidth for memory mappings, each monitored memory mapping is assigned to their own monitoring group. For this one to one relation between monitored memory mapping and monitoring group, the monitoring groups which are already in use need to be tracked. Whenever a new memory mapping is selected to be monitored, a free monitoring group is allocated for it.

The CXL memory expander design is intended to map the exposed monitoring groups the same device physical memory. This makes it possible to measure monitoring groups that each read or write to the same memory, which can be used to measure the bandwidth of different mappings of shared memory. To support this feature, the kernel page allocator needs to be patched, because its possible to write on the same device physical address from different addresses from registered monitoring groups and the operating system does not know how to handle this. However, we noticed that the bandwidth of a mapping that is bound to a monitoring group gets split to surrounding monitoring groups. This made it unfeasible to use the same physical memory for all monitoring groups in our case. Due to time constraints, we were not able to find out why the bandwidth gets split to other monitoring groups. We suspect that some pages get mapped to other monitoring groups by the page allocator patch, that results in the wrong accounting of read and write accesses by the monitoring groups. As a result, we resort to each monitoring group having its own device physical memory and thus do not need the kernel page allocator patch. Each monitoring group having its own device physical memory limits the use-cases to monitor memory mappings without shared memory, but still demonstrates the capability of monitoring memory mappings.

4.2 Monitoring Group Allocator

Whenever a memory mapping is selected to be monitored, a monitoring groups has to be allocated that measures all memory accesses that happen through the memory mapping. In order to select a memory mapping to be monitored we provide additional flags for syscalls handling memory mappings. These flags create memory mappings that are immediately monitored, or monitor existing memory mappings or stop a memory mapping from being monitored.

Every time a memory mapping is selected to be monitored, our allocator in the kernel finds a monitoring group that is not already used by an existing monitored memory mapping. Monitoring groups measure all reads and writes, that happen through pages, that are allocated on the monitoring group. By having a one-to-one relation between a monitored memory mapping and monitoring group, we guaranty that the measured reads and writes only come through the corresponding monitored memory mapping. When the memory mapping is undone, the allocator frees the monitoring group for use by a new memory mapping. The strategy of the allocator selects a free monitoring group from the intersection of the monitoring group that are online, not in use and the memory tier that is specified. Of the all the monitoring groups that are in the intersection, the allocator chooses the first monitoring group, meaning the one with lowest id.

$$\begin{gathered} \text{online}, \text{in_use}, \text{tiers}_i \subseteq \{\text{monitoring_group}_0, ..., \text{monitoring_group}_{n-1}\} \\ \text{free} = \text{online} \cap \text{in_use}^{\complement} \cap \text{tiers}_i \\ \text{selected} = \min_i\{\text{monitoring_group}_i \mid \text{monitoring_group}_i \in \text{free}\} \end{gathered} \tag{1}$$

We thought of choosing a fitting monitoring group at random, but chose not to, to be easier to debug. Choosing at random might distribute the utilization and wear of the memory hardware across all devices within a memory tier equally, once bandwidth monitoring is available in more memory devices.

After a monitoring group is determined by the allocator, all pages of the memory mapping get migrated to that monitoring group. As an example, when memory mapping a file, the file content gets put in the page cache of the unmonitored host RAM. Once we select the memory mapping to be monitored using one of the flags, the allocator selects a free monitoring group. Then all pages of the page cache of the file on the unmonitored host RAM get copied to newly allocated pages on the

monitoring group on the custom CXL memory device and the pages on the unmonitored host RAM get deallocated. The virtual address returned by mmap() stays the same. Just the translation from the virtual address changes from the physical address of the unmonitored host RAM to the physical address of the monitoring group on the custom CXL memory device.

During testing we observerd some allocated pages on monitoring group that are not part of any memory mapping that we made. These pages generate a small amount of reads and writes to the monitoring, thus putting some noise in our bandwidth measurement for the memory mapping that we want to monitor. This far we did not do much research to find the source of these pages. We suspect that the operating system might put some metadata on the monitoring group or performs tests that generate small bandwidth.

4.3 Report Measurements

The measurements for the monitored memory mappings are exposed for other applications that want to use them, for example memory tiering applications. The bandwidth is calculated for each monitoring group. Bandwidth is defined as the difference between two measurements of bytes read or bytes written over the difference in time between these to measurements and reported as a units of data per second, e.g. $\frac{\text{MB}}{s}$ (Megabytes per second). The periodical timer accesses the counters of each monitoring group in a fixed time interval to report the read and written bytes and calculate the read and write bandwidth. The difference between the measurements of the current iteration counter_t and last iteration counter_{t-1} equal the bytes that were read and written since the last measurement Δb. These differences each get divided by the fixed time interval between the measurements Δt to get the bandwidth since the last measurement.

$$\text{bandwidth}_t = \frac{\Delta b_t}{\Delta t} = \frac{\text{counter}_t - \text{counter}_{t-1}}{\Delta t} \tag{2}$$

The calculation of the bandwidth in a periodical fixed time interval results in predictable behavior for applications that make use of the measurements. We deem this characteristic as important, even though this solution has some performance overhead on the CPU. A first approach only calculated the bandwidth whenever an

application requested it. This makes the accuracy of the bandwidth dependent on the polling interval of the application, which results in unpredictable behavior. If the polling interval was to long, then the application could not differentiate between the bandwidth rising gradually or having a spike. The bandwidth is calculated with the difference between the last measurement, meaning the last time an application polled. This means that an application might miss bandwidth information between two polls if another application polled in between. We chose to have periodically timed bandwidth calculation to have predictable behavior for applications and to have no application breaking the measurements for other applications.

Chapter 5

Implementation

Now, that we described our Design in Chapter 4, we describe our implementation for the Linux kernel and usespace memory tiering application. We begin in the order in which an application would interact with our interfaces. Starting with modifications to different syscalls that an application can use to configure the monitoring of memory mappings in Chapter 5.1. After which, we describe the inner workings of the NUMA node ID (nid) allocator and migration of the memory mapping to another NUMA node in Chapter 5.2. We explain the bandwidth calculation and output of all measurements through the `proc` file system in Chapter 5.4. At last, we show the usage of the provided interfaces in a sample memory tiering application in Chapter 5.5.

5.1 Configuration of Monitoring

We extend existing syscalls that already handle memory mappings to support configuring the monitoring of memory mappings. The syscall `mmap()` [4] creates a new virtual address space mapping for the process and returns the virtual address of this area. By adding a new flag `MAP_MONITOR` to `mmap()` [4], an application can create a memory mapping that is being monitored starting with its creation. The new flag `MAP_MONITOR` can be combined with the regular other flags of `mmap()` [4] using the bitwise or operation. The syscall `madvise()` [5] is usally used to give the kernel advice on a memory mapping based on which the kernel might do some optimizations. New flags added to `madvise()` [5] are used to configure the monitoring of existing memory mappings. Calling `madvise()` [5] with the new `MADV_MONITOR` flag monitores a previously unmonitored mapping. Calling `madvise()` [5] with the

new `MADV_UNMONITOR` flag stops a memory mapping from being monitored. The new `MADV_PROMOTE` and `MADV_DEMOTE` flags promote or demote a mapping to a higher or lower memory tier. The values of the flags are chosen to not collide with existing flags.

The removal of memory mappings happens in two ways: either the application can remove a memory mapping explicitly by using the `munmap()` [4] syscall or the removal can happen implicitly by the operating system when the application quits, for example using the `exit()` [6] syscall. We found that both paths lead to calling the function `void remove_vma(struct vm_area_struct *vma)`, which removes the virtual memory mapping. We extend this function to also mark that mapping as no longer monitored. This way the NUMA node gets marked as free for use by a new monitored memory mapping.

5.2 NUMA node ID allocator

Every time a memory mapping is selected to be monitored, a NUMA node has to be allocated to put the memory mapping onto. One NUMA node is only allowed to contain one memory mapping such that the mapping is the only way to generate reads and writes for the corresponding monitong group counters to track. As such whenever a memory mapping is removed or should be unmonitored, the NUMA node the memory mapping is bound to should be marked as free for a new monitored memory mapping to be used.

The kernel uses `nodemask` structures to keep track of the availability of NUMA nodes [7]. These `nodemask` structures are large bitmasks where the index of the bit, starting at the least significant bit, corresponds to the NUMA node ID (nid) the bit tracks. For example, the kernel tracks whether a node is online, has memory, or a CPU using `nodemask` structures. The NUMA node ID allocator stores the NUMA nodes that are already in use by a monitored memory mapping in a `nodemask` structure.

Whenever a memory mapping should be monitored using `mmap()` [4] or `madvise()` [5], a new NUMA node ID (nid) to bind the memory mapping to must be allocated. To get all nids that are free for use by monitored memory mappings, the allocator combines three `nodemask` structures using the bitwise and operation. One `nodemask` structure is `node_states[N_ONLINE]`, which the kernel uses to keep track which NUMA nodes are online. In the second `nodemask` structure, the allocator keeps

track of nodes that are already in use by monitored memory mappings. Inverting this `nodemask` using a bitwise not operation gives all nodes that are not in use by any monitored memory mappings, but which also contain nodes which might not even be online. The last `nodemask` structure exists for each memory tier and tracks which nodes are part of that tier. Combining all three nodemasks using the bitwise and operation yields a `nodemask` that contains all free NUMA node IDs. The allocator then selects the first free node, meaning the first set bit starting from the least significant bit.

$$\text{nid}(\text{tier}) = \text{first}(\text{online } \mathbf{AND\ NOT} \text{ in_use } \mathbf{AND} \text{ tiers}_{\text{tier}}) \tag{3}$$

In order to make debugging easier, we chose not to select a fitting NUMA node at random. Choosing at random might distribute the utilization and wear of the memory hardware across all devices equally.

After finding a free nid for the memory mapping, it has to be marked as in use, so that the nid is not selected again by another call to the allocator while a mapping is still bound to that node. This is done by setting the bit at the index of the nid in the `nodemask` structure that keeps track of the nodes in use.

Deallocation using `munmap()` [4], `exit()` [6] or `madvise()` [5] with `MADV_UNMONITOR` works by getting the nid from the `vm_policy` attribute in the `vm_area_struct` which represents the memory mapping and clearing the bit in the in_use nodemask at the index of the nid. The attribute `vm_policy` gets set when migrating to a new node (see Chapter 5.3).

5.3 Migration to another NUMA node

In Chapter 5.2 we explained how we keep track of NUMA nodes that are in use by monitored memory mappings. But the memory mappings itself still need to be migrated to the NUMA node that was allocated to them by the NUMA node ID allocator. For this we use a function that is already implemented in the kernel, namely the function `do_mbind` in the kernel, which the syscall function for `mbind()` [8] is just a wrapper for. Calling a syscall function directly is not possible because the arguments of syscalls are stored in different registers than for normal function calls. The syscall `mbind()` [8] binds a memory range to NUMA nodes using a memory policy. By calling `mbind()` [8] with the range of the memory mapping that needs to

be migrated and a `nodemask` structure with a set bit at the index of the allocated nid as the memory policy, the memory mapping effectively gets migrated to the allocated NUMA node. The memory policy that is provided to `mbind()` [8] gets stored in the `vm_policy` attribute of the `vm_area_struct` which represents the memory mapping and can then be used to find out which NUMA node the memory mapping is bound to. Importantly `mbind()` [8] already handles all the memory management that is needed for our purposes, so we do not need to reimplement them, which reduces code duplication. The memory management done by `mbind()` [8] consists of migrating all pages of the memory mapping from the old NUMA node the NUMA nodes that are set in the memory policy. Another part of the memory management done by `mbind()` [8] is merging memory mappings together when multiple mappings with the same attributes line up right after another or splitting a memory mapping when only a part of it changes attributes.

5.4 Expose Bandwidth Information

The memory mapping is bound to a NUMA node and the corresponding monitoring group counts all reads and writes that are made through the memory mapping. These measurements of the monitoring groups need to be accounted to the corresponding memory mapping. Further, this information about the read and written bytes, as well as the read and write bandwidth, of the memory mapping needs to be made available to the userspace, so that memory tiering applications can promote or demote memory mappings based on this data.

Whenever a memory mapping gets monitored and bound to a NUMA node, the values of the corresponding monitoring group counters get stored in the `vm_area_struct` as a baseline for the read and written bytes. The monitoring group counters count all load and store memory accesses of the NUMA node since the initialization of the memory device. This includes all memory mappings that have been bound and unbound to the NUMA node since. Storing the values of the monitoring group counters when the memory mapping is first bound to the NUMA node as a baseline makes it possible to retrieve the read and written bytes since the memory mapping is monitored. Every time we want to measure the read and written bytes since the memory mapping is monitored, the values of the monitoring group counters get subtracted by the baseline.

The bandwidth is calculated for all NUMA nodes in a periodical fixed time interval using a high-resolution timer (`hrtimer`) and fixed-point arithmetic, since the use of floating point numbers is discouraged in the kernel [9]. Bandwidth is not dependent on any baseline since it is the value of change of read or written bytes over time. Since the bandwidth is not dependent on when any memory mapping started to being monitored, the bandwidth is calculated for each NUMA node and can then be referenced when polled for measurements about a memory mapping. The kernel provides two different timers.

The `timer_list` timer is used for cases where timing precision is not important, like timeouts or watchdogs [10]. The timeout of the `timer_list` timer is given as jiffies, which as a global variable that gets counts ticks since the system booted and is incremented every 1 to 10 ms by an interrupt [25].

The high resolution `hrtimer` operates in the unit of nanoseconds and is used for high precision applications like multimedia [10]. Userspace sleep and timers are implemented using the `hrtimer` timer [10].

We chose to use the `hrtimer` timer to calculate the bandwidth in a periodical fixed time interval. We do this to be able to use the same fixed time difference we give the timer to also use when calculating the bandwidth. Jiffies might be too imprecise and would lead to imprecisions in the bandwidth calculation.

The bandwidth calculation has to be done using fixed-point arithmetic. The switch between kernel and userspace does not store the state of floating point registers to improve performance. Floating point arithmetic can be enabled for code blocks, but the use of floating point arithmetic is generally discouraged in the kernel [9]. We define bandwidth bw as the difference in read or written bytes between to points in time Δb over the difference of the points in time Δt. The differences in data read or written Δb and the time differences Δt are each scaled by their own constant factor scale_B and scale_T. This controls the scale of the output unit. In our case, we opted for Megabytes per second ($\frac{\text{MB}}{s}$) as in 1,000,000 bytes per second. This output scale fits the measured bandwidth well. The fraction of fractions is simplified to a single fraction to reduce rounding errors from integer division. Integer division cuts of the decimal part of the solution, which reduces precision. To get some decimal part for the solution of the bandwidth calculation, that otherwise would be lost in the integer division, the numerator gets multiplied by scale_{FP}.

$$\begin{aligned} &\text{scale}_{\text{FP}} = 10^c; c, \text{scale}_B, \text{scale}_T, \Delta b, \Delta t \in \mathbb{N}_+ \\ \text{bw}_t &= \frac{\frac{\Delta b}{\text{scale}_B}}{\frac{\Delta t}{\text{scale}_T}} \cdot \text{scale}_{\text{FP}} = \frac{\Delta b \cdot \text{scale}_T}{\Delta t \cdot \text{scale}_B} \cdot \text{scale}_{\text{FP}} \\ &= \frac{(\text{bytes_written}_t - \text{bytes_written}_{t-1}) \cdot \text{scale}_T \cdot \text{scale}_{\text{FP}}}{\Delta t \cdot \text{scale}_B} \end{aligned} \tag{4}$$

To retrieve the integer part, the solution has to be divided by scale_{FP}, using the fact that integer division cuts of the decimal part. This is effectively the solution as if the scale_{FP} would not be multiplied to the numerator since the scale_{FP} cancel each other out. The decimal part can be retrieved by getting the remainder of that division between the bandwidth and scale_{FP} using the modulo operator. For this to work, scale_{FP} has to be a power of ten with the exponent representing the precision of the fixed-point number with the number of digits behind the decimal point.

$$\begin{aligned} \text{integer part} &= \left\lfloor \frac{\text{bw}_t}{\text{scale}_{\text{FP}}} \right\rfloor \\ \text{decimal part} &= \text{bw}_t \bmod \text{scale}_{\text{FP}} \end{aligned} \tag{5}$$

The scaling factors should be chosen carefully, such that when multiplying all variables in the numerator and denominator do not overflow the integer data type for. With $\text{scale}_T = 1000000000$, $\text{scale}_B = 1000000$ and $\text{scale}_{\text{FP}} = 1000$ the data type for a `uint64` is not overflown.

$$\begin{aligned} \Delta b \cdot \text{scale}_T \cdot \text{scale}_{\text{FP}} < 2^{64} &\Rightarrow \Delta b_{\max} = 18446744 \text{ bytes} \approx 18.45 \text{ MB} \\ \Delta t \cdot \text{scale}_B < 2^{64} &\Rightarrow \Delta t_{\max} = 18446744073709 \text{ ns} \approx 18446 \text{ s} \end{aligned} \tag{6}$$

To improve $\Delta b_{\max}$ and $\Delta t_{\max}$ the fraction of the scaling factors $\frac{\text{scale}_T \cdot \text{scale}_{\text{FP}}}{\text{scale}_B}$ may be precomputed or shortend, since each scaling factor is a power of ten. This can be further improved as the Δt might also be considered a constant. These optimizations have not been done so far since the limits where not reached when testing and to change the values more easily.

All measurements of monitored memory mappings for a process are exposed in two virtual files in the `proc` file system. The file `/proc/pid/maps` [11] shows a line for each memory mapping of a process. Each line contains the virtual start and end

address of the memory mapping, as well as the permissions, the offset into a mapped file, the device, inode, and pathname if a file is mapped. We extend this layout with an extra column to the very right, to be backwards compatible with any application that reads these information. The new column contains the read and written bytes by the memory mapping and the read and write bandwidth. The read and written bytes get calculated by getting the nid the memory mapping is bound to using the `vm_policy` attribute of the `vm_area_struct`, which represents the memory mapping, getting the values of the counters of the corresponding monitoring group to the nid and subtracting the baseline, which is also stored in the `vm_area_struct`. The bandwidth is read from an array containing the bandwidth information for each NUMA node, as calculated in the periodical fixed time interval `hrtimer`, and indexed using the nid from the `vm_policy` in the `vm_area_struct`. The same information is also exposed through the file `/proc/pid/smaps` [12]. This file lists all attributes of each memory mapping in the form of `/proc/pid/maps` [11] and as key-value pairs, which is easier to read by people since the key provides a more descriptive name.

```
$ ./access-cxl m &
$ cat /proc/$!/maps
root@pc45-vm:~/host-share# cat /proc/578/maps
55b1bbf15000-55b1bbf16000 r--p 00 00:2a 41609491 /root/host-share/access_cxl
55b1ee1d7000-55b1ee1f8000 rw-p 00 00:00 0        [heap]
7f7f4a800000-7f7f6a800000 rw-p 00 00:00 0        1 247806096 683173507 0.26 0.89
7f7f6a910000-7f7f6a913000 rw-p 00 00:00 0
7f7f6a913000-7f7f6a93b000 r--p 00 08:01 5174     /usr/lib/linux-gnu/libc.so.6
7f7f6ab1a000-7f7f6ab1c000 rw-p 00 00:00 0
7f7f6ab1c000-7f7f6ab20000 r--p 00 00:00 0        [vvar]
7f7f6ab20000-7f7f6ab22000 r--p 00 00:00 0        [vvar_vclock]
7f7f6ab22000-7f7f6ab24000 r-xp 00 00:00 0        [vdso]
7f7f6ab5a000-7f7f6ab5b000 rw-p 36 08:01 5171     /usr/lib/linux-gnu/ld-linux.so.2
7f7f6ab5b000-7f7f6ab5c000 rw-p 00 00:00 0
7ffd396e7000-7ffd39708000 rw-p 00 00:00 0        [stack]
```

Listing 1: Sample output of `/proc/<pid>/maps`. The third line of the output shows the statistics of the monitored memory mapping. The numbers in the right most column show the memory tier, read counter, write counter, read bandwidth and write bandwidth. The output is stripped and shortend to fit on this page.

```
$ ./access-cxl m &
$ grep Monitor /proc/$!/smaps
MonitorTier:           1
MonitorRead:   322972716  B
MonitorWrite: 1642184172  B
MonitorReadBW:      0.00 MB/s
MonitorWriteBW:     1.41 MB/s
```

Listing 2: Sample output of `/proc/<pid>/smaps` only showing the statistics of the monitored memory mapping. The key-value pairs and the units are descriptive and easy to read for humans.

The function `seq_printf` is used to output the fixed-point numbers of the bandwidth calculations to the files in the `proc` file system. The output of the fixed-point numbers using `printf` is made simple through the use of macros. One macro converts the fixed-point number into the two arguments, being the integer part and the decimal part. Since the fixed-point number gets split into an integer part and decimal part, the format string consists of two conversion specifiers. The format string for a fixed-point number is put in a macro definition. This takes advantage of the fact that in the c language string literals can be written one after another but semantically form one long string.

```
#define FP_FMT_STR "%5llu.%02llu"
#define FP_FMT(fp) ((fp) / SCALE_FP), ((fp) % SCALE_FP)

seq_printf(m, "MonitorWrite: %llu B\nMonitorWriteBW: " FP_FMT_STR " MB/s\n",
       bytes_written, FP_FMT(bandwidth));
// expands to
seq_printf(m, "MonitorWrite: %llu B\nMonitorWriteBW: " "%5llu.%02llu" " MB/s\n",
       bytes_written, ((bandwidth) / SCALE_FP), ((bandwidth) % SCALE_FP));
```

Listing 3: Code snippet that demonstrates the use of macros to output fixed-point numbers using `printf`. The last line shows the expanded macros that insert the format string and integer and decimal part of the fixed-point number.

5.5 Memory Tiering

One use-case for bandwidth monitoring is memory tiering applications. As such, we chose memory tiering as an application to test and evaluate our bandwidth monitoring for memory mappings (see Chapter 6).

Memory with different latencies create a hierarchy of memory tiers. Based an these tiers we build a memory tiering application that puts memory mappings with high bandwidth utilization on tiers with low latency and memory mapping with low bandwidth utilization on tiers with high latency. This has the goal of increasing the overall bandwidth while decreasing the overall latency.

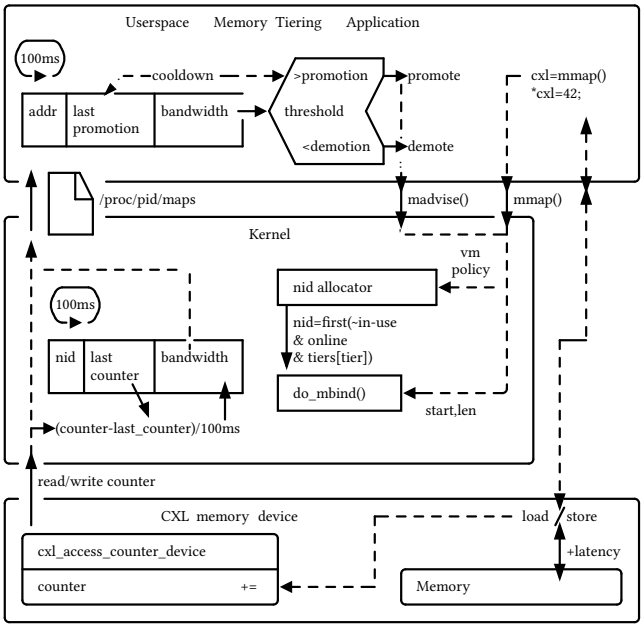


Figure 3: Component diagram showing the interfaces and data flow of the userspace memory-tiering application, kernel and CXL memory device. The memory-tiering application chooses to promote/demote with a cooldown based on the periodically measured bandwidth. The tiers are implemented in the CXL memory device using a latency option at the read/write functions.

All of our implementation is tested and executed on a QEMU virtual machine with dynamic binary translation instead of KVM, because KVM would throw an internal error when registering a CXL fixed memory window as a NUMA node [16]. We use a patched version of QEMU that introduces the *CXL access counter device* (Chapter 4.1). We further patch QEMU to optionally add latency to load and store operations on individual memory devices, to create memory tiers with different latencies. This is done by adding a command line parameter value for the latency to the *CXL access counter device*, which gets written to a new latency attribute of the structure that defines the cxl access counter device. In the functions handling the load and store operations of CXL devices, namely `cxl_type3_read()` and `cxl_type3_write()` that also handle incrementing the monitoring group counters, the latency attribute is read, which is used to iterate a busy loop. The busy loop takes time to compute in the thread that handles the virtual CPU from which the load or store operation originated. This should mimic the blocking characteristic of a real load or store operation with latency on the memory device. We first tried using `nanosleep()` to implement the latency, but this resulted in the kernel running into deadlocks. This also happens for too many (>1000) iterations in the busy loop. Due to time constraints, we did not look further into the source of the deadlock. Using `CONFIG_LOCKDEP` we detected that the deadlock is not a result of any of our code.

We also started implementing a latency parameter for `HostMemoryBackend`, which is an abstract class in the QEMU Object Model (QOM). `MemoryBackendRAM` is one implementation of this class, which our QEMU configuration uses as a memory backend for the CXL memory devices and where the latency parameter can be appended. These abstractions made it hard to implement and retrieve the latency value in the functions handling the reads and writes, which is why we opted for our problem specific solution of storing the latency in the *CXL access counter device*. But implementing the latency for `HostMemoryBackend` opens up use-cases besides our specific case with CXL memory devices.

Our NUMA node ID allocator supports the memory tiers via our newly added the two `madvise()` [5] flags `MADV_PROMOTE` and `MADV_DEMOTE`, and `nodemask` structures for each tier. On the initialization of the kernel module, an array of `nodemask` structures representing each tier gets initialized. Each `nodemask` structure of this array represents a tier and stores which NUMA nodes are part that tier. Depending on the provided tier argument, the allocator allocates a node from that tier or returns with

an error code, if all nodes from the requested tier are already in use. The `nodemask` structure of the requested tier gets combined using the bitwise AND operation with the online `nodemask` and the inverse `nodemask` of the nodes in use to get all free nodes on that tier. Our `madvise()` [5] flags `MADV_PROMOTE` and `MADV_DEMOTE` get the nid the memory mapping is bound to using the `nodemask` `vm_policy` attribute of the `vm_area_struct` that represents the memory mapping. The new tier is calculated by adding or subtracting one from the current tier, depending on whether to promote or demote. The current tier is retrieved by reading the nid the memory mapping is bound to and looking up which tier the nid belongs to. The memory mapping is then bound and migrated to the allocated nid and the old nid is deallocated. The memory tier of a monitored memory mapping is exposed in the `/proc/pid/maps` and `/proc/pid/smaps` files of the proc file system.

To decide which memory mapping to promote and which to demote, we use a userspace application that periodically reads and parses the `/proc/pid/maps` file and compares the bandwidth of each of its monitored memory mappings to a promotion and demotion threshold. We implement promotion and demotion to occur on two different thresholds. If promotion and demotion would occur on the same threshold, then the noisy measured bandwidth would oscillate above and below the threshold, leading to unwanted rapid changes of the memory tier. The promotion and demotion is done with a cooldown and exponential moving average of the bandwidth. Changing the tier of the memory mapping migrates all pages to the allocated node on the new tier, which results in a large increase of bandwidth of 3 to 6 $\frac{\text{MB}}{s}$ for a about 3 seconds. If a memory mapping gets demoted to a cold tier, then this bandwidth spike would cross the promotion threshold, which would trigger an unintendet promotion. As a solution to this, we introduce a cooldown. If a memory mapping get promoted or demoted, a cooldown of 10 seconds starts, during which the memory mapping is blocked for further promotions or demotions. We use the exponential moving average of the bandwidth using a weigth of 0.1 to detect if the bandwidth crossed a promotion or demotion threshold. The exponential moving average smoothes out spikes in the bandwidth measurements, which otherwise would cross one threshold and start a promotion or demotion. We also considered using a moving median of the last five measurements. But the moving median includes spikes, that last for a couple of measurements, while the exponential moving average is able to smoothen them out.

Each monitored memory mapping is tracked together with the time of its last promotion or demotion in a dynamic array. A mark and sweep algorithm periodically keeps our dynamic array of monitored memory mappings in sync with the data from the `/proc/self/maps` file. The algorithm runs periodically in a separate thread and marks all entries in the dynamic array as dirty and reads and parses the `/proc/self/maps` file. For each line in the `/proc/self/maps` file, it finds the mapping in the dynamic array and marks it as clean or creates a new entry if it is not found, which means the mapping must have been newly created. After reading the file, the algorithm iterates over all entries in the dynamic array and removes all entries that are still marked as dirty by swapping with the last element of the dynamic array and decreasing the length variable until the swapped element is clean or the dynamic array shrunk to the current entry. Further, the time, tier, and read and write bandwidth are reported to the console. The userspace application creates a new thread that runs besides the normal execution that generates bandwidth on memory mappings. In another thread the promotion manager compares the difference between the current time and the time of the last promotion or demotion of the tracked mapping to the cooldown threshold for each monitored memory mapping in the dynamic array. If the difference is above the threshold and the parsed bandwidth is larger than the promotion threshold or lower than the demotion threshold and the tier of the tracked mapping is not already the tier target tier, then the tier is changed. The tier is changed by calling `madvise()` [5] with the corresponding `MADV_PROMOTE` or `MADV_DEMOTE` flag and the attribute of the tracked mapping storing the last time the tier was changed is updated to the current time. The promotion manager is executed in a thread separate from the mark and sweep algorithm because `madvise()` [5] takes a couple of seconds to migrate all pages, which would block the algorithm and the bandwidth report to the console.

Other data structures could also be used to track the memory mappings. In a linked list the elements can be deleted in constant time, but iterating the list takes longer since the entries are scattered in memory. If we assert that the `/proc/pid/maps` file is sorted by the start address of the memory mapping, then we do not need to iterate over the whole dynamic array for every line in the file and instead only need to increase the index of the dynamic array. Then for each entry that still exists in the file and the dynamic array or is new and that exists in the file we append that entry to a new dynamic array. After the file is read, the old dynamic array can be freed

or used as the new dynamic array for the next time the file is read. This follows a similar approach to the mark and compact algorithm. However, we did not want to assert that the `/proc/pid/maps` file is sorted and take the little overhead of iterating over the dynamic array for each line in the file and worse deletion performance as we argue that selecting a memory mapping for monitoring or unmonitoring does not happen very often.

Chapter 6

Evaluation

After the implementation, we now evaluate our approach. First, we describe our hardware setup and configurations of the software environment and explain our methodology for conducting our measurements in Chapter 6.1. Then we show measurements of our bandwidth monitoring implementation in Chapter 6.2. Further, we evaluate how our approach can be used in the context of memory tiering application in Chapter 6.3. At the end, we discuss the use of bandwidth monitoring for file mappings in transparent DAX mappings in Chapter 6.4.

6.1 Evaluation Setup and Methodology

The evaluation environment consists of the hardware and operating system of the host system, as well as settings and operating system of the guest system on the virtual machine. The host system is build with a CPU containing 16 cores and 32 threads, which allows for multiple virtual cores of the virtual-machine. Additionally the host system is equipped with 96 GB of host memory, which is used to configure multiple memory regions for the CXL memory expanders for the virtual machine. Table 1 shows the specifications of the hardware components in more detail. On the software side, the host system is running the operating system Fedora Linux 41 (Server Edition) with Linux kernel version 6.15.7. All measurements are done inside of the virtual machine QEMU version v10.1.0 with additional changes, that include the *cxl-access-counter-device* and optional latency to memory accesses. The virtual-machine is running the operating system Debian 13 Genericcloud with Linux kernel version 6.15.0 with additional changes, that include a driver for the *cxl-fpga-counter-device* and our memory monitoring implementation.

Component	Specification
Mainboard	Supermicro X11SPi-TF
Processor	Intel(R) Xeon(R) Gold 6130 (32) @ 3.70 GHz
Memory	2 × Samsung M393A4K40CB2-CTD 32GiB DIMM DDR4 2666 MHz 2 × Samsung M393A2K40BB2-CTD 16GiB DIMM DDR4 2666 MHz
Storage	KINGSTON SA2000M8250G NVMe 250 GB

Table 1: Hardware specifications of the host system for evaluation

We configure QEMU to emulate four cores. Further, we configure four CXL memory devices, each with 4 GB of their own memory and their own monitoring group counters. Half of the memory devices are configured to be the hot tier with no added latency and the other half is configured as the cold tier with an added latency, implemented as 1000 iterations of a busy loop. The latency is set to 1000 iterations of a busy loop. More iterations or sleeping results in a deadlock in the kernel that does not occur in our code and we did not investigate due to time constraints. We configure QEMU to use dynamic binary translation instead of KVM, because KVM would throw an internal error when registering a CXL fixed memory window as a NUMA node. This was only fixed in a more recent version of QEMU [16], which we did not try to apply our patches to due to time constraints. We use a virtual machine for faster iteration when implementing and easier change and extensibility of memory configurations. The FPGA implementation does currently not implement different latencies for monitoring groups, which we need to configure different memory tiers. The use of a virtual machine allowed us to extend the emulation of memory to include configurable latencies.

When compiling the Linux kernel we enable the `CXL_FPGA_COUNTER` option and our `MM_MONITOR` option. We enable `CONFIG_LOCKDEP`, `HARDLOCKUP_DETECTOR` and `SOFTLOCKUP_DETECTOR` during development to detect when the kernel runs into a deadlock, but disable them for evaluation, because `CONFIG_LOCKDEP` has some runtime overhead, that introduce noise in the measurements. Lastly, we disable `NUMA_BALANCING`, which is the kernel's own implementation of memory tiering [35]. This config would otherwise interfere with our own implementation. Disabling `NUMA_BALANCING` also partly solves the distribution of bandwidth between surrounding NUMA nodes that we experienced when using the page allocator patch.

To evaluate the bandwidth measurements we need to create a controlled load on a monitored memory mapping, which we can compare against the measurements. For that, we periodically generate an actual target bandwidth using `memset`, which writes a number of bytes to get target bandwidth. The bandwidth generation is done in a separate thread for each monitored memory mapping. We do this so that each thread can be on a separate core and thus the memory accesses from different memory mappings do not interfere with each other and slow each other down because of the CPU stalls from different memory latencies. The bandwidth generation is also separate from the bandwidth monitoring to measure the bandwidth parallel to the bandwidth generation. The number of bytes that `memset` writes is determined by multiplying the target bandwidth with the fixed time interval of 50 ms for the periodical timer. The interval of 50 ms for the bandwidth generation is exactly half of the interval for monitoring. We do this to eliminate the risk of measuring the bandwidth while the bandwidth generation is mostly sleeping, which otherwise adds noise to the measured bandwidth. See Equation 7 for a calculation of the number of bytes that need to be written during 50 ms to reach the example target bandwidth of $1\frac{\text{MB}}{s}$.

$$50\ \text{ms} \cdot 1\frac{\text{MB}}{s} = 50\ \text{ms} \cdot \frac{10^6 B}{10^3\ \text{ms}} = 50\ \text{ms} \cdot \frac{10^3 B}{\text{ms}} = 50\ \text{kB} \tag{7}$$

The time `memset` takes to write the bytes is measured and then subtracted from the fixed time interval of 50 ms. The difference is used to sleep to fill the remaining time of the fixed time interval in case `memset` is done faster than 50 ms. The bandwidth generation can underrun when `memset` takes longer than the fixed time interval and thus misses the target bandwidth. We try to reduce underruns by choosing a target bandwidth that is low enough, so that `memset` can write the number of bytes within the fixed time interval. Parsing, processing and reporting of the bandwidth measurements for each monitored memory mapping from the `proc` file system happens in another thread to not be effected by and not effect bandwidth generation.

6.2 Bandwidth Measurement

To evaluate the bandwidth measurement for a monitored memory mapping, we generate a target bandwidth that follows a triangle wave. The bandwidth rises linearly from zero to a maximal bandwidth and then decreases back to zero. We do this to observe the bandwidth measurement across multiple target bandwidths and change of bandwidth. We decided to use a maximum bandwidth of $1\frac{\text{MB}}{s}$ to minimize underruns described in Chapter 6.1, while still having a visible change of the target bandwidth following the triangle wave compared to the noise.

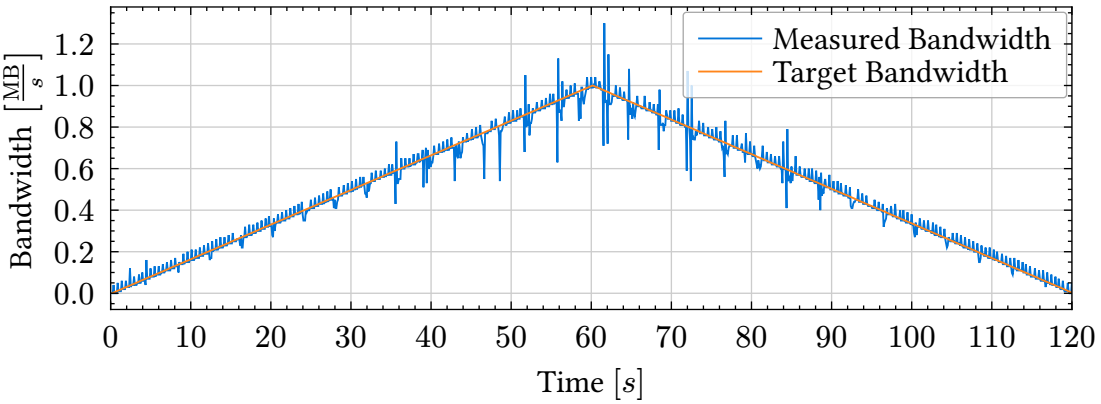


Figure 4: Plot showing the target bandwidth that follows a triangle wave (orange line) and the measured bandwidth as calculated using the monitoring group counters (blue line). The measured bandwidth follows the target bandwidth with some noise.

The noise originates from four sources: First, pages allocated on the monitoring group that are not part of the monitored memory mapping write to memory and distort the counted memory accesses. Second, on high target bandwidths, the bandwidth generation might not be able to keep up and underrun, resulting in the actual bandwidth being lower than the target bandwidth. This is then represented in the measured bandwidth. Third, the phase of the periodical bandwidth monitoring might sometimes not align with phase of the periodical bandwidth generation. This shows in the bandwidth being monitored while the bandwidth generation sleeps for most of the measurement time and results in less reported bandwidth. We try to mitigate this by the bandwidth generation having double the frequency of the bandwidth monitoring. Fourth, some noise stems from the fact, that the measurements take place in a virtual machine. The performance of the virtual machine is dependent on the utilization of the host system. We made sure not to run demanding applications

on the host system during measurements. Some background tasks on the host system may still spike in resource usage and effect performance of the virtual machine, resulting in noise.

In Figure 4, the measured bandwidth is a little greater than the target bandwidth most of the time. We suspect that allocated pages that are not part of the monitored memory mapping generate a small constant bandwidth, resulting in more bandwidth than we expect. The noise increases for larger target bandwidths. We suspect that the bandwidth generation almost underruns and `memset` takes at lot of time during the interval. The bandwidth generation increasingly underuns with target bandwidths above $1\frac{\text{MB}}{s}$.

The measured bandwidth and the target bandwidth are in the range of a couple megabytes per second. These bandwidths are very small in comparison to bandwidths that we commonly see on real hardware, which is in the range of a couple gigabytes per second. We believe reason behind the low bandwidth is the execution on a virtual machine with dynamic binary translation instead of KVM. Software-based dynamic binary translation is a lot slower than hardware virtualisation with KVM. Thus the virtual CPU is slower and cannot issue as many memory accesses per second as with hardware virtualisation.

While conducting measurements, we observed high bandwidth with up to 15 $\frac{\text{MB}}{s}$ when writing bytes with a distance of one page. We suspect the kernel mapping the page only when it is first accesses using demand paging. Then the kernel zero-fills the page if it was not zeroed in the first place, so the userspace application cannot read the previous data. This results in the high bandwidth measurement.

When creating a monitored memory mapping, the bandwidth spikes for a couple seconds. This happens because the pages of the memory mapping get migrated to the monitoring group, which generates bandwidth of 3 to 6 $\frac{\text{MB}}{s}$ for about 3 seconds. We do not measure while monitored memory mapping is initialized, to focus on the intentional target bandwidth, which comes from the bandwidth generation. However, these spikes can be seen in Figure 5, when the pages of the memory mapping get migrated to a new memory tier.

The average absolute difference between the measured bandwidth and the target bandwidth is around 20 kB.

6.3 Memory Tiering

Our memory tiering application is modeled to resemble real world memory tiering applications but focuses on the relevant aspect of migrating a memory mapping to a fitting memory tier, that best matches the utilized bandwidth of the memory mapping. We implemented a memory tiering application because we did not find an existing suitable application that could easily be modified to take advantage of measuring bandwidth per memory mapping. At the time we started our work, the implementation of TDMs was not suitable to integrate our bandwidth monitoring approach. This would introduce measuring the bandwidth of memory mapped files, based on which the mapping could be promoted to a DAX mapping. Other applications that are used to benchmark memory performance, like in-memory databases, do not make sense to be measured as fine-grained as for memory mapping. It is suffice to measure such applications on a process granularity [26].

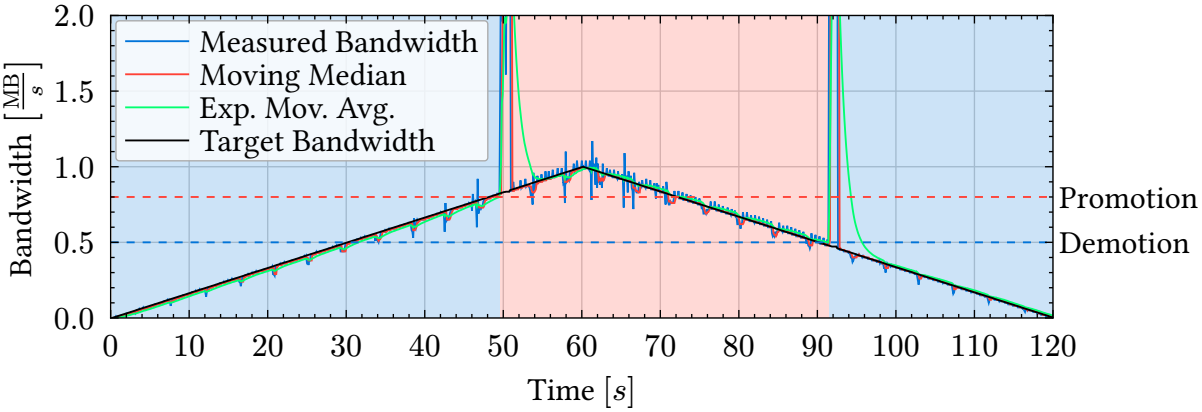


Figure 5: Plot showing the target bandwidth that follows a triangle wave (black line) and the measured bandwidth as calculated using the monitoring group counters (blue line). The background color represents the change of memory tiers, with the hot tier shown in red and the cold tier shown in blue. Dashed horizontal lines mark the promotion and demotion thresholds.

In Figure 5, the monitored memory mapping starts in the cold tier with the target bandwidth increasing linearly. The spike of the measured bandwidth at second 47 would trigger a promotion if no moving average was used. The memory mapping only get promoted to the hot tier when the exponential moving average of the measured bandwidth reaches the promotion threshold at the 50th second. Because of

the promotion, all pages of the memory mapping get migrated to the newly allocated NUMA node of the hot tier, resulting in a large bandwidth increase between 3 to 6 $\frac{\text{MB}}{s}$ for two seconds. The hot tier has no added latency and the bandwidth performs similar to Figure 4. The noise increases for target bandwidths in the range of 0.9 to 1.0 $\frac{\text{MB}}{s}$, because the bandwidth generation is close to underrun. After half the time, the target bandwidth reached the maximum and decreases. The spike of the measured bandwidth at second 89 would trigger a demotion if no moving average was used. The spike lasts for a couple of measurements and thus is included in the moving median, while the exponential moving average smoothes out the spike. If we chose the moving median for demotion, we would change tiers too early. The memory mapping gets demoted to the cold tier once the exponential moving average of the measured bandwidth drops below the demotion threshold. Because of the demotion, all pages of the memory mapping get migrated to the newly allocated NUMA node of the hot tier, resulting in a large increase of bandwidth. The spike crosses the promotion threshold. But any change of tiers is blocked by the 10 second cooldown, that started with the demotion.

We evaluate how well our memory tiering application select the most fitting memory tier based on the monitored bandwidth of a memory mapping. We create one hot memory mapping and one cold memory mapping with a constant bandwidth generation of $1\frac{\text{MB}}{s}$ and $0.25\frac{\text{MB}}{s}$ respectively. Our memory mapping application puts the hot memory mapping on the hot memory tier and the cold memory mapping on the cold memory tier. We compare our memory tiering application with configuration where all memory mapping are put on the hot tier and one configuration where all memory mapping are put on the cold tier. Additionally, we compare against interleaving both memory mappings on both memory tiers using `mbind()` [8], which is a common way to utilize multiple NUMA nodes.

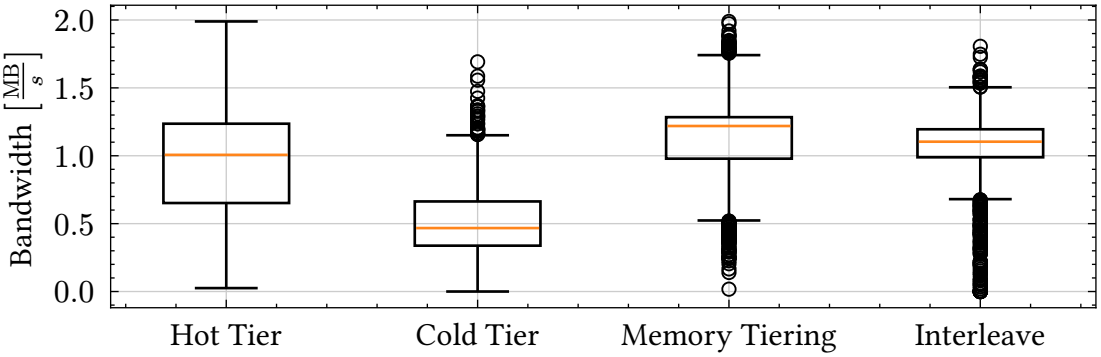


Figure 6: Comparison of a hot and a cold memory mapping being bound only to the hot tier, bound only to the cold tier, bound to the hot and cold tier respectively and each mapping being interleaved across the hot and cold tier.

The measurements in Figure 6 are taken with a kernel, which has `CONFIG_LOCKDEP` enabled and introduces runtime overhead. The runtime overhead results in more noise compared to the previous plots and a larger difference of the measured bandwidth on the cold and hot memory tier. We chose to show these measurements instead of ones with `CONFIG_LOCKDEP` disabled, to better visualize the different bandwidths of the memory tiers.

In Figure 6, both memory mapping are bound to the hot tier, which has no added latency. Both memory mapping are mostly able to reach their target bandwidth, which the average of the combined bandwidth hovering around $1\frac{\text{MB}}{s}$. The bandwidth is very noise, which might be the case, because each the memory mapping is close to underrun. The cold tier cannot provide enough bandwidth for the hot memory mapping, which is why the average bandwidth is only close to $0.5\frac{\text{MB}}{s}$. With memory tiering, each memory mapping gets put on a fitting memory tier. The hot memory mapping is put on the hot memory tier, which allow the memory mapping to reach its target bandwidth. The cold memory mapping is put on the cold memory tier, which is enough for the memory mapping to also reach its target bandwidth. This is the only configuration, which averages on the expected total bandwidth of $1.25\frac{\text{MB}}{s}$. The interleaving of memory mappings across both memory tiers results in each second page being bound to the cold tier and every other page being bound to the hot tier. Each second page of the hot memory mapping is bound to the cold tier with limited bandwidth. This results in a slightly worse average bandwidth, compared to the memory tiering configuration. Thus, the memory tiering achieves a

total measured bandwidth across both memory mappings, which is 21% better than the measurements from just using the hot tier and 8% better than interleaving both memory mapping across both memory tiers.

6.4 Transparent DAX Mappings

At the time we started our work, the implementation of TDMs was not suitable to integrate our bandwidth monitoring approach. We argue that our implementation would benefit TDMs. TDMs work on unmodified applications by overwriting the functions for file access using `LD_PRELOAD`. With this, TDM can measure the bandwidth with a sliding window in the `read()` and `write()` functions when copying the data to the DAX mapping or page cache. Based on the bandwidth, the TDM manager decides if the file should be promoted to use DAX mappings, to bypass the kernel and get strong access latency guarantees, or demoted to use POSIX I/O to free space of the DRAM cache on the hybrid SSD for files that utilize more bandwidth. Getting the bandwidth like this is not possible with memory mapped files, since there is no function that reads or writes data, that could be intercept, because the application directly accesses the DAX mapping or page cache using pointers. The TDM manager would need to be able to monitor the bandwidth for DAX mappings and the page cache to decide to promote or demote a file mapping. By introducing our approach on bandwidth monitoring for memory mapping, TDMs can monitor the bandwidth for file mappings. This requires hybrid SSDs and host RAM that implement monitoring group counters for monitoring DAX mappings and the page cache. With such memory, the `mmap()` function could be overwritten to create a monitored memory mapping, which migrates between a monitored DAX mapping and monitored page cache. The TDM manager might even only rely on the bandwidth measurements from the monitored DAX mapping and monitored page cache. Not only memory mapped files, but also for the overwritten `read()` and `write()` functions access the DAX mapping or page cache. This would make the bandwidth measurement using the sliding window in the `read()` and `write()` functions obsolete and might simplify the implementation.

With the hardware-assisted bandwidth measurement, the TDM manager could measure the total bandwidth that is utilized on the Hybrid SSD. The TDM manager could act before a hybrid SSD is fully saturated with memory access requests, when the

total bandwidth increases above a threshold. If the hybrid SSD is fully saturated with memory access requests, it has to buffer the requests. This results in an increase in latency, which breaks our strong access latency guarantees for DAX mappings. If the total bandwidth on the hybrid SSD increases above the threshold, the TDM manager could implement different strategies which DAX mapping to demote. The simplest strategy to immediately reduce the most utilization would be to select the DAX mapping with the highest bandwidth utilization on the hybrid SSD and demote it to use POSIX I/O instead. The file mappings that benefit most from DAX mapping with strong latency guarantees are exactly the ones with high bandwidth utilization. So this strategy might only be used as a last resort to desaturate the hybrid SSD. Another strategy might try to prioritize DAX mappings with high bandwidth utilization to maximize their benefit. Therefore, the TDM manager would demote multiple DAX mappings with minimal bandwidth utilization to desaturate the hybrid SSDs. An alternative promotion strategy of the TDM manager could to promote file mapping in such a way, that their utilized bandwidth gets optimally packed such that the total sum of the utilized bandwidth is just under the threshold. This strategy tries to get the most overall bandwidth benefit from the hybrid SSD.

Chapter 7

Discussion and Future Work

In this chapter, we discuss the limitations of our nid allocator implementation in Chapter 7.1 and provide other implementation ideas for the nid allocator in Chapter 7.2, which might solve the limitations. Further, we show a possible solution to monitor shared memory mapping in Chapter 7.3 and argue whether our approach could also be implemented just in userspace in Chapter 7.4.

7.1 Shortcomings of the Allocator

The nid allocator has some shortcomings and assumes that the userspace application uses the syscalls only with the start address and length of memory mappings it selected for monitoring. A malicious application can mark used monitoring group as free, allocate monitoring groups that cannot be freed again, and distort the measured bandwidth using crafted arguments to syscalls.

Memory mappings can be bound to NUMA nodes that are already in use by a monitored memory mapping using the `mbind()` [8] syscall. The allocator does not forbid binding to NUMA nodes that are already in use. Binding multiple memory mappings to the same NUMA node can be used to measure memory mappings that make sense to measure together or to deal with the limited amount of monitoring groups. This can also be done maliciously, by binding a memory mapping to a NUMA node that is already in use and generate reads and writes through the memory mapping to distort the measurements of the memory mapping that was selected to be monitored.

Further, the nid allocator does not work correctly on some edge cases when an intersection with an already monitored memory mapping is selected. Selecting only a subrange of a monitored memory mapping to be unmonitored will unmonitor the complete memory mapping and free the monitoring group. Selecting an intersection, a monitored memory mapping will bind the intersection to a newly allocated monitoring group while the rest from the already monitored memory mapping stays at its monitoring group. This breaks when the newly selected memory range to be monitored is a superset of the already monitored memory mapping. In this case the already monitored memory mapping also gets bound to the newly allocated monitoring group, which leaves the old monitoring group without only memory mapping and no way to free it. Ideally the allocator would keep the same monitoring group and just extends the monitored memory mapping to the selected superset. Selecting a subsection of an already monitored memory region splits that memory region, resulting in two `vm_area_struct` being bound to a monitoring group. This breaks the assumption that one `vm_area_struct` is bound to one monitoring group. Unmonitoring a memory region that only includes one `vm_area_struct` would mark the monitoring group as free, even though the other `vm_area_struct` is still bound to it and might generate bandwidth.

In the next chapter we provide ideas to solve the discussed shortcomings using alternative nid allocators. One solution might be to special case every possible combination of intersections. We demonstrated in our evaluation that our approach works even with the discussed shortcomings. We argue that the shortcomings are unlikely to show up in real world applications and would have to be fabricated by a malicious application. Because of this and due to time constraints, we decided to not invest time to implement a solution.

7.2 Alternative Allocators

We considered some alternative implementations for the nid allocator that might solve some or all shortcomings as discussed in Chapter 7.1.

One solution might be to iterate over all memory mappings to see if there is a NUMA node that is not used by any memory mapping. This way we ensure that the allocated NUMA node definitely has no other memory mapping bound to it except the one that the NUMA node was allocated for. For our implementation we only had to look at the memory mappings of the current process. But iterating over the memory mapping of the current process is not enough for this strategy as memory mappings of another process might be bound to NUMA nodes that no memory mapping of the current process is using. Because of this, we would have to iterate over all memory mappings of all processes. As a result, we discarded this strategy because of the expected performance overhead, even though the iterations would only happen whenever a memory region is selected for monitoring, which might not happen often. To improve this strategy we thought of adding a data structure that contains the monitored memory mappings of every process. This improves the number of elements the allocator has to iterate over.

Another solutions would be to do changes directly to the memory management code in the kernel and prevent memory mappings to be bound to already in use NUMA nodes in the first place. The kernel allocates pages on NUMA nodes that are online, have enough capacity left, and have a close distance to the CPU on which the process was scheduled [3]. Each one of these constrains would allow us to disable a NUMA node for future allocations while the memory that already was allocated stays the same. This would make it impossible to maliciously bind memory mappings to NUMA nodes that are already in use by monitored memory mappings. Further, this would make added flags to `mmap()` [4] and `madvise()` [5] obsolete and it would suffice to call `mbind()` [8] with a nodemask selecting all nodes. The call to `mbind()` [8] would select one NUMA node from the provided nodemask and then disable it, such that another call of `mbind()` [8] would select a different active NUMA node.

We now look into how we can use the constraints to disable a NUMA node for new allocations. The kernel keeps track which NUMA nodes are online in nodemask `node_states[N_ONLINE]`. When the nid allocator selects a free node, then the nid could be marked as offline in the `node_states` nodemask and thus no new allocations

to that node would be possible. It might be that the kernel has some assumptions about online NUMA nodes, that might break when setting the memory as offline even though the NUMA node is still actively in use. This would need to be investigated. All NUMA nodes being accessible by default with places in the kernel allocating on each available node without any way to limit the access is a known problem [20]. An allocation can only limit itself from the full access of all NUMA nodes, like we do with `mbind()` [8]. One proposed kernel patch is the introduction of private memory nodes, which are inaccessible by default and an allocation has to first request access [31]. Using this patch, we could register the monitoring groups as private memory nodes and request access to one node when a memory mapping is selected to be monitored. This would make it impossible to have bandwidth from pages that have not explicitly requested access and thus are not part of the monitored memory mapping. The nid allocator would select a private memory node that does not contain any memory mappings or does not have any allocated pages, which could come only from the one monitored memory mapping that requested access to that node.

The capacity of a NUMA node can be manipulated by removing all entries from the freelists of the NUMA node after the monitored memory mapping already got migrated to the node. When pages get allocated on a NUMA node, the page allocator allocates the physical memory from ranges that are defined in the entries of the freelists. New entries get appended to the appropriate freelist when pages get deallocated. This is a change that deeply interacts with the memory management logic and thus might be hard to implement correctly.

The distance between two NUMA nodes is defined by the latency between the nodes. The kernel tries to allocate pages on NUMA nodes that are closest to the NUMA node of the CPU that executes the process. In our setup NUMA node zero contains all CPUs and the host RAM while all NUMA nodes greater than zero do not contain a CPU and only contain memory. Setting the distance between a used NUMA node and all other nodes to the highest possible distance would make the allocator select a node with a lower distance. This might break when all NUMA nodes are in use or the last free NUMA node does not have enough capacity. Then the allocator would select a NUMA node that is in use, even when it has the highest possible distance. This is because it still can fulfill the allocation because it still has enough capacity.

7.3 Shared Memory

The custom FPGA-based memory expander only implements a predetermined amount of monitoring groups. When allocating one monitoring group per memory mapping, this might exceed the amount of monitoring groups available on the custom CXL memory expander. One solution to this would be to share multiple monitoring groups per memory mapping. PEBS-based sampling would allow to approximately portion the bandwidth per monitoring group back to each memory mapping. Every n-th load/store instruction is captured using the CPU's PMU. The used memory address is looked up in the page table to see if it accesses the CXL device. The approximate measurements are used to portion the measurements from the FPGA to the actively scheduled process. We probably need to extend the PEBS-based approximate measurements to further look what memory mapping the address belongs to in the actively scheduled process. Then we use these measurements to portion the measurements from the FPGA to the memory mappings to get the overall bandwidth approximation per memory mapping. The nid allocator would need to support binding multiple memory mappings to one monitoring group. For this we might implement reference counting of memory mappings for monitoring groups. When the nid allocator selects a monitoring group for a memory mapping, the reference counter increases. When a memory mapping should not be monitoring anymore, the reference counter decreases. The reference counter might be used by the allocator to select a monitoring group that has the least amount of references to balance the load on monitoring groups, which might improve the accuracy of the PEBS-based portioning. A NUMA node is free if the reference counter is zero.

To implement measuring the bandwidth of shared memory mappings, we put all memory mappings that map the same shared memory on one monitoring group. The bandwidth for each memory mapping is calculated by portioning the bandwidth measurements of the monitoring group using the approximate measurements from PEBS.

7.4 Userspace Implementation

Everything in this work which we implemented in the kernel might also be possible in userspace. This would result in an easier setup for the user since the kernel would not require patches.

All functionality could be implemented in a patched standard library, which would allow keeping the added flags for `mmap()` [4] and `madvise()` [5]. The patched standard library can be used which any program using the standard library. This is done by setting `LD_PRELOAD` environment variable to the shared object of the patched shared library. The patched standard library wraps the `mmap()` [4] and `madvise()` [5] functions to add the flags. The nid allocator would work using files to communicate and keep track of used NUMA nodes across multiple processes. This might work similar to `.pid` files through which applications keep track if another instance is already running. The memory mapping can then be bound to the selected NUMA node by calling the `mbind()` [8] syscall directly from userspace. The monitoring group counters would be retrieved by reading from the PCI BAR using the `/sys/bus/pci/devices/<id>/resource<0..N>` file [13]. The counters can be read periodically with a timed loop using `usleep`. The bandwidth is calculated using the difference between the current and last counter value and dividing by the time interval in-between the measurements. Userspace can use floating point numbers, which would make our use of fixed-point arithmetic obsolete. Swapping the implementation of the standard library with `LD_PRELOAD` only works for dynamically linked programs that link with the standard library.

Chapter 8

Conclusion

In this thesis, we implemented bandwidth monitoring for memory mappings in the Linux kernel. Our implementation uses hardware-assisted monitoring of a custom CXL memory device. We introduced an interface to select a memory mapping to be monitored, which allocates and binds the memory mapping on a free monitoring group. Further, we provide an interface for the bandwidth measurements of monitored memory mappings to userspace. To evaluate our implementation, we build a memory tiering application that promotes memory mapping to a fitting memory tier based on our bandwidth measurements. We extended the QEMU virtual machine to be able to emulate a heterogeneous tiered memory system with different latency levels. In our evaluation, we showed that our measured bandwidth closely follows the actual bandwidth and demonstrated that our bandwidth measurement enables the memory tiering to achieve a greater overall bandwidth utilization that is 21% better than just using the hot tier and 8% better than interleaving each memory mapping across all memory tiers. The measured bandwidth shows some noise and spikes and the nid allocator has shortcomings. Nevertheless, we show that our bandwidth monitoring for memory mappings works and can be of advantage for applications like memory tiering. We discussed monitoring DAX mappings in TDMs to demote them to POSIX I/O to prevent bandwidth exhaustion of the hybrid SSD.

Bibliography

[1] 2019. DAMON: Data Access Monitor. Retrieved January 22, 2026 from https://damonitor.github.io/posts/damon/

[2] Direct Access for Files — The Linux Kernel Documentation. Retrieved January 12, 2026 from https://docs.kernel.org/6.6/filesystems/dax.html

[3] What Is NUMA? — The Linux Kernel Documentation. Retrieved January 12, 2026 from https://www.kernel.org/doc/html/v4.18/vm/numa.html

[4] Mmap(2) - Linux Manual Page. Retrieved May 20, 2026 from https://www.man7.org/linux/man-pages/man2/mmap.2.html

[5] Madvise(2) - Linux Manual Page. Retrieved May 20, 2026 from https://www.man7.org/linux/man-pages/man2/madvise.2.html

[6] _exit(2) - Linux Manual Page. Retrieved May 20, 2026 from https://www.man7.org/linux/man-pages/man2/_exit.2.html

[7] Physical Memory — The Linux Kernel Documentation. Retrieved July 7, 2026 from https://docs.kernel.org/next/mm/physical_memory.html#nodes

[8] Mbind(2) - Linux Manual Page. Retrieved May 20, 2026 from https://www.man7.org/linux/man-pages/man2/mbind.2.html

[9] Floating-Point API — The Linux Kernel Documentation. Retrieved July 8, 2026 from https://docs.kernel.org/core-api/floating-point.html

[10] Timers and Hrtimers - Linux Kernel Internals. Retrieved June 1, 2026 from https://kernel-internals.org/interrupts/timers/

[11] Proc_pid_maps(5) - Linux Manual Page. Retrieved June 30, 2026 from https://www.man7.org/linux/man-pages/man5/proc_pid_maps.5.html

[12] Proc_pid_smaps(5) - Linux Manual Page. Retrieved June 30, 2026 from https://www.man7.org/linux/man-pages/man5/proc_pid_smaps.5.html

[13] 5. Accessing PCI Device Resources through Sysfs — The Linux Kernel Documentation. Retrieved July 14, 2026 from https://www.kernel.org/doc/html/latest/PCI/sysfs-pci.html

[14] Altera Corporation. 2025. Agilex™ 7 FPGA I-Series Development Kit (2x R-Tile and 1x F-Tile) | Altera. Retrieved from https://www.altera.com/products/devkit/a1jui0000061ropmaq/agilex-7-fpga-i-series-development-kit-2x-r-tile-and-1x-f-tile

[15] John Backus. 1978. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Commun. ACM* 21, 8 (August 1978), 613–641. https://doi.org/10.1145/359576.359579

[16] Li Chen. 2026. Cxl: Avoid KVM Internal Error for Fixed Memory Windows | Patchew. Retrieved July 26, 2026 from https://patchew.org/QEMU/20260317033304.3185291-1-me@linux.beauty/

[17] Yiqi Chen, Xiping Dong, Zhe Zhou, Zhao Wang, Jie Zhang, and Guangyu Sun. 2025. FPGA-based Emulation and Device-Side Management for CXL-based Memory Tiering Systems. https://doi.org/10.48550/arXiv.2502.19233

[18] Compute Express Link Consortium and Inc. 2023. Compute Express Link (CXL): Specification. *CXL Consortium White Paper* 3, 1 (2023). Retrieved from https://computeexpresslink.org/cxl-specification/

[19] Jonathan Corbet. 2023. The Future of Memory Tiering. *LWN.net* (May 2023). Retrieved January 20, 2026 from https://lwn.net/Articles/931421/

[20] Jonathan Corbet. 2026. Support for Private Memory Nodes. *LWN.net* (May 2026). Retrieved July 25, 2026 from https://lwn.net/Articles/1072881/

[21] 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. https://doi.org/10.1145/2669594.2669599

[22] Daniel Habicht, Yussuf Khalil, Lukas Werling, Thorsten Gröninger, and Frank Bellosa. 2024. Fundamental OS Design Considerations for CXL-based Hybrid SSDs. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems*,

November 03, 2024. ACM, Austin TX USA, 51–59. https://doi.org/10.1145/3698783.3699380

[23] Intel Corporation. 2025. Intel Resource Director Technology (Intel RDT) Architecture Specification. Retrieved from https://cdrdv2-public.intel.com/%20789566/356688-intel-rdt-arch-spec.pdf

[24] Hyunjoo Jin, Fanny Potkin, Wen-Yee Lee, Anton Bridge, and Max A. Cherney. 2025. The AI Frenzy Is Driving a Memory Chip Supply Crisis. *Reuters* (December 2025). Retrieved March 4, 2026 from https://www.reuters.com/world/china/ai-frenzy-is-driving-new-global-supply-chain-crisis-2025-12-03/

[25] M. Jones. 2010. Timers and Lists in the 2.6 Kernel. Retrieved June 1, 2026 from https://developer.ibm.com/tutorials/l-timers-list/

[26] Linus Kämmerer. 2025. Hardware-Assisted Memory Bandwidth Management for CXL-based Memory Devices. (2025).

[27] Yussuf Khalil, Daniel Habicht, Pascal Ellinger, Frank Bellosa, Javier González, Adam Manzanares, and Vivek Shah. 2025. Transparent DAX Mappings: Towards Automatic Kernel Bypass with CXL-Based Hybrid SSDs. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems (DIMES '25)*, October 13, 2025. Association for Computing Machinery, New York, NY, USA, 54–62. https://doi.org/10.1145/3764862.3768178

[28] Yussuf Khalil. 2025. Chisel FPGA Implementation of a CXL Memory Expander.

[29] 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)*, March 25, 2023. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[30] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track (Middleware '19)*,

December 09, 2019. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3366626.3368125

[31] Gregory Price. 2026. Private Memory NUMA Nodes. Retrieved July 25, 2026 from https://lwn.net/Articles/1083791/

[32] Rafał Rudnicki. 2022. Memory Tiering (Part 1). Retrieved January 22, 2026 from https://pmem.io/blog/2022/06/memory-tiering-part-1/

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

[34] 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*. Association for Computing Machinery, New York, NY, USA, 604–621. Retrieved March 9, 2026 from https://dl.acm.org/doi/10.1145/3676641.3711999

[35] David Vernet. 2022. NUMA Rebalancing on Tiered-Memory Systems. *LWN.net* (May 2022). Retrieved July 20, 2026 from https://lwn.net/Articles/893024/

[36] 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. 2023. 601–617. Retrieved February 18, 2026 from https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[37] 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)*, November 2024. 1518–1531. https://doi.org/10.1109/MICRO61859.2024.00111