CXL-Interplay: Unraveling and Characterizing CXL Interference in Modern Computer Systems

Shunyu Mao[†*], Jiajun Luo[†*], Yixin Li[†], Jiapeng Zhou[‡], Weidong Zhang[§], Zheng Liu[§¶], Teng Ma[§**], Shuwen Deng[†‖**]

[†]Department of Electronic Engineering, Tsinghua University
[‡]Institute of Computing Technology, Chinese Academy of Sciences
[§]Alibaba Group
[¶]Zhejiang University
[‖]Zhongguancun Laboratory

Abstract—**Compute Express Link (CXL) is a promising technology that addresses memory and storage challenges. Despite its advantages, CXL faces performance threats from external interference when co-existing with current memory and storage systems. This interference is under-explored in existing research. To address this, we develop CXL-Interplay, systematically characterizing and analyzing interference from memory and storage systems. To the best of our knowledge, we are the first to characterize CXL interference on real CXL hardware. We also provide reverse-reasoning analysis with performance counters and kernel functions. In the end, we propose and evaluate mitigation solutions.**

I. Introduction & Related Work

Compute Express Link (CXL) [8] provides a comprehensive solution to existing memory wall problems [13] and has garnered significant attention from both industry and academia. CXL offers several advantages, including high bandwidth density and a standardized interface for memory expansion and pooling [23]. These features make it an ideal solution to address the limitations of current cloud storage architectures, and have the potential to revolutionize data center architectures [4]. In industry, leading companies are actively exploring and adopting CXL to improve the efficiency and scalability of their data center products, including Intel [9], Samsung [25], and SK Hynix [19]. Academically, researchers are investigating various aspects of CXL, including its architecture [26], performance implications [28], [29], and potential security concerns [1].

However, in coexisting with current memory and storage systems in the server architecture, CXL faces understudied performance threats from external interference, which are not adequately reflected in existing research. Zooming out from CXL itself and considering its operational environment, CXL can potentially be affected by interference from 1) interactions between Main Memory (**MMEM**) and CXL and 2) interactions between neighboring storage components and CXL. Avoiding interference and maintaining performance isolation are critical for performance-sensitive applications. For instance, MT^2 [34] tries to explore the interference between persistent memory (PMEM) and DRAM, and it identifies noisy neighbors among concurrent applications to mitigate memory traffic interference. To the best of our knowledge, this *interference issue for CXL has not been extensively studied in existing research*. Current CXL simulators typically introduce delay factors manually to simulate memory delays [20], [33], which neither reflects the real world environment nor considers interference. Moreover, existing hardware profiling studies on CXL primarily focus on evaluating the behavior of CXL under different operations and applications [17], [28], [29], without considering the impact of external environmental factors.

Given that, we develop CXL-Interplay, which *systematically characterizes and analyzes the potential reasoning of interference between memory/storage systems and CXL memory* using carefully designed microbenchmarks and tailored real applications. We also discuss potential effective regulation that can be applied to fit real CXL interference scenarios.

First, in order to clearly identify and characterize CXL interference with memory and storage system, we develop a configurable microbenchmark and systematically go through the basic interference condition of CXL with MMEM and SSD system using *two different real CXL hardware setups*. As shown in Table I, we are the first to demonstrate and characterize the CXL interference using real CXL devices.

Furthermore, we conduct reverse-reasoning based on profiling evaluation results, exploring kernel functions and hardware performance counters, including the new CXL-related counters.

For the real testing setup, we explore file system, database, and machine learning applications for storage-related testing environment. For the memory-related testing environment, we focus on large-language model (LLM), in-memory database, and graph computing applications.

Given the interplay results, we further explore software and discuss hardware intervening solutions to provide effective regulation for the system to mitigate the interference impact. Based on our evaluation, it is feasible to restore bandwidth of the MMEM by 204GB per second, which reaches 99% of its original level, using software solutions, at the cost of a 5GB per second reduction in CXL bandwidth.

To summarize, this work makes the following contributions:

1) Systematically characterize the interference issue in current server architectures among CXL, MMEM, and storage devices. To the best of our knowledge, this is the first study examining interference of CXL on real hardware.
2) Conduct comprehensive microbenchmarks, evaluate memory-related and storage-related real applications and perform reverse-reasoning analysis for evaluation results.
3) Explore and evaluate software-based intervention solutions to address the interference problem and mitigate its impact on real hardware.

The code used in this paper will be released under open-source license at https://github.com/THU-HAS/CXL-Interplay.

II. Background & CXL-Interplay Setup

A. Compute Express Link (CXL)

Compute Express Link (CXL) [8], first developed in 2019, is an open standard interconnect designed to enhance the performance of

---

TABLE I: Comparison of CXL-INTERPLAY with state-of-the-art CXL characterization work. To the best of our knowledge, we are the first to investigate interference and regulation of real CXL devices and conduct reverse-reasoning analysis.

	Real ASIC Device	Inter-ference	Regulation	Reverse-reasoning Analysis
Emulation works [16], [27], [33]	✗	✗	✗	✗
Sun [28]	✓	✗	✗	✗
Tang [29]	✓	✗	✗	✗
Liu and Wang [17]	✓	✗	✗	✗
Our Work	✓	✓	✓	✓

TABLE II: Experimental setup of CXL-INTERPLAY.

	System A	System B
CPU	4th Gen Xeon Platinum 4.0 GHz	4th Gen Xeon Gold 4.0 GHz
Number of physical cores per socket	48	32
Memory per socket Channels Speed	1TB, 8 Channel, DDR5-4800	256GB, 8 Channel, DDR5-4800
Storage	SAS SSD	NVMe PCIe 4.0 MLC SSD
File System	XFS	XFS
CXL memory device	Montage ASIC 64GB	Intel Agilex FPGA 32GB DDR4-2400
OS	CentOS 7	Ubuntu 22.04.4 LTS

Fig. 1: Schematic illustration of hardware setup

data-centric applications by enabling high-speed, low-latency communication between CPU/accelerators, memory, and storage.

The CXL protocol stack is divided into three parts: CXL.io, CXL.cache, and CXL.mem, each serving distinct purposes to facilitate various types of data transmission and memory access. CXL.io ensures compatibility with PCIe, enabling traditional I/O operations and device discovery. CXL.cache supports accelerators accessing CPU's internal caches and host memory. CXL.mem allows direct memory access and pooling, extending memory across multiple devices.

CXL devices are classified into three types, each tailored for specific applications. Type-1 devices, such as SmartNICs, utilize CXL.io and CXL.cache to facilitate communication with host DDR memory. Type-2 devices, including GPUs, ASICs, and FPGAs, use CXL.io, CXL.cache, and CXL.mem to share memory resources with the processor. Type-3 devices rely on CXL.io and CXL.mem to enable memory expansion and pooling. Type-3 devices enable extra DRAM capacity and bandwidth.

There are currently two types of CXL devices: FPGA-based and ASIC-based. Intel Agilex FPGA integrates CXL hard IP, which is a standard component in recent CXL studies (e.g., Sun et al [28], Zhong et al [36]). Besides, emerging ASIC-based CXL memory devices are becoming available, with promises from several vendors such as Samsung [25], Montage [31] and Micron [18].

B. Memory System and CPUless Node

DRAM is widely used in memory systems as the main memory. It is typically interconnected via a parallel interface. However, the limited capacity and bandwidth blocks its scalability for memory-intensive applications.

Secondary memory such as persistent memory (PMEM) can meet the demands of memory-intensive applications, and its large capacity makes it an alternative to DIMM. PMEM can be identified as the memory of an additional node without CPU.

As a type of secondary memory, CXL memory provides CPUless node interface. However, unlike PMEM, CXL devices rely on PCIe, utilizing its physical layer and provide multiplexing features, resulting in physical topologies akin to those of PCIe.

C. CXL-INTERPLAY Hardware Setup

In this work, we systematically characterize and explore how CXL memory interferes with other memory-related access, specifically MMEM access, and storage-related access such as NVMe or SAS SSD, using microbenchmarks and real applications.

Table II demonstrates the experimental setup of CXL-INTERPLAY. We have two systems that will be referred to later as System A and

System B respectively. System A has SAS SSD and an ASIC-based CXL memory device manufactured by Montage. System B consists of NVMe SSD and an Intel FPGA-based CXL memory device.

Figure 1 illustrates the topology of our hardware setup, where NUMA node 0 and NUMA node 1 each possess their own cores and DDR5 DRAM, while the CXL Memory Expander is situated on NUMA node 2 with zero processor core, directly connected to Node 0. Therefore, tasks are always allocated to the cores on Node 0.

III. CXL-INTERPLAY MICROBENCHMARKS

In this section, we set up comprehensive microbenchmarks to examine the CXL interference problem with SSD and MMEM.

A. Microbenchmark Setup

In our microbenchmark, three different memory-related operations (load (*ld*), store (*st*), and non-temporal store (*ntst*)) [1] and two different storage-related operations (random-read (*randread*) and random-write (*randwrite*)) are cross-evaluated.

For memory-related operations, we develop our microbenchmark by measuring the bandwidth of CXL memory and MMEM with cxlMemTest [28]. We carry out our experiments with all possible numbers of threads. For better measurement, we disable Hyperthreading [5], lock CPU's frequency to 2GHz, and clear cache before each test. We also allocate main process and interfering process to separate cores of the same NUMA node. The measurement is repeated multiple times and get average. We also carry out case studies on random/sequential storage access pattern and MOVDIR64B instruction.

For storage-related operations, we employ FIO [3] with *libaio* as *ioengine* to perform sustained *randread*/*randwrite* operations with an *iodepth* of 16 and a *block size* of 4KB for 20 seconds.

---

[1] We do not have non-temporal load mainly considering its sparse applicability, which are demonstrated in a lot of works [12], [21], [22]

TABLE III: The schematic illustration of bandwidth across different systems and configurations. In each box, the x-axis and y-axis represent the operation of the background and the device under test, respectively. We categorize the influence into seven types, each represented by a color in the colorbar: "Deep Suppression", below -40%; "Moderate Suppression", -40% to -10%; "Mild Suppression", -5% to -10%; "No Influence", within -5%; Promotion follows the same pattern.

Evaluated Device	MMEM	SSD	CXL	
Background	CXL	CXL	MMEM	SSD

Bandwidth — System A

MMEM (Background CXL):

y \ x	ld	st	ntst
ld	+0.3%	+0.0%	-92.9%
st	+0.2%	+0.1%	-85.1%
ntst	-20.2%	-2.6%	-90.2%

SSD (Background CXL):

y \ x	ld	st	ntst
randread	-0.6%	-3.1%	-59.7%
randwrite	+1.0%	-3.7%	-61.3%

CXL (Background MMEM):

y \ x	ld	st	ntst
ld	+0.6%	+0.4%	-10.1%
st	+7.3%	+6.5%	+14.8%
ntst	+0.3%	+0.6%	-8.9%

CXL (Background SSD):

y \ x	randread	randwrite
ld	-0.8%	+0.0%
st	-4.9%	-2.4%
ntst	+0.0%	-0.1%

Bandwidth — System B

MMEM (Background CXL):

y \ x	ld	st	ntst
ld	-44.8%	-12.1%	-93.1%
st	-41.6%	-4.2%	-87.6%
ntst	-46.1%	-26.7%	-93.2%

SSD (Background CXL):

y \ x	ld	st	ntst
randread	+0.3%	+0.6%	-3.2%
randwrite	-3.3%	+0.2%	-53.3%

CXL (Background MMEM):

y \ x	ld	st	ntst
ld	-9.1%	-3.1%	-10.2%
st	+28.7%	+5.1%	+83.2%
ntst	-0.8%	-0.1%	-9.8%

CXL (Background SSD):

y \ x	randread	randwrite
ld	-0.7%	-1.0%
st	-0.1%	-0.7%
ntst	+0.3%	+0.0%

Colorbar: Deep Suppression — Moderate Suppression — Mild Suppression — No Influence — Mild Promotion — Moderate Promotion — Deep Promotion

B. Microbenchmark Characterization and Evaluation

1) MMEM/SSD with CXL Traffic: The left half of Table III illustrates the trends of MMEM/SSD's bandwidth under the interference of CXL. Each value represents the maximum interference observed across all thread count configurations. Our observations are as follows: a) The bandwidth of MMEM/SSD suffers significant interference when CXL performs *ntst*, reaching up to 93.2%. b) CXL *ld* and *st* can also be disruptive to MMEM, especially on System B, where over 40% interference is observed. c) As for SSD, the interference from CXL *ld* and *st* traffic is relatively mild.

INSIGHT #1: CXL traffic (especially non-temporal store (*ntst*)) significantly impairs bandwidth of concurrent MMEM and SSD traffic, with MMEM experiencing greater degradation than SSD.

2) CXL with MMEM/SSD Traffic: The right half of Table III illustrates the bandwidth trends of CXL under the interference of MMEM/SSD. Our observations are as follows: a) CXL bandwidth experiences performance boost when CXL performs *st* operations alongside MMEM traffic; b) In other MMEM cases, *ntst* operations exert a moderate to mild negative impact on CXL bandwidth. c) The bandwidth of CXL is not significantly influenced by SSD.

C. Reverse-reasoning Analysis

Figure 2 illustrates the datapath for CXL memory, SSD and MMEM workloads. Figure 2 depicts workload mixes involving SSD traffic and CXL traffic, where the CXL memory expander and storage device share the PCIe interface. It also shows workload mixes between CXL memory and MMEM, where interfaces are not shared, but sharing of network-on-chip (NoC) can lead to interference.

Performance counters. We discover that when SSD is under interference of CXL, its L2 and local LLC miss average latency becomes much higher than the no interference scenario, which prompts an investigation into the PMU counters listed in Table IV. It can be observed that there is a significant increase in L2 miss latency, up to a factor of 15, accompanied by a notable reduction in bandwidth. However, the L3 miss latency remains relatively constant. This suggests that an examination of network-on-chip (NoC) between the L2 and L3 will be necessary. CPUs need to maintain a record of in-flight transactions that access the L3 slices that are attached to other cores. This record table presumably will, however, become a bottleneck.
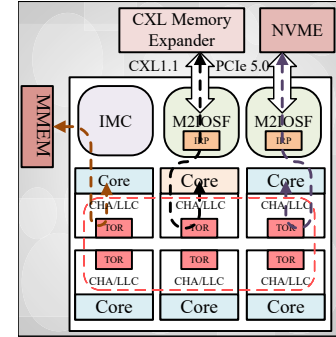

Fig. 2: Datapaths in filesystem, CXL and MMEM workloads.

From the discovery and the results in Table III and the datapaths in Figure 2, we then focus on: a) Table of Request (TOR), a queue between cores and LLC, tracking pending Cache Home Agent (CHA) transactions [15] of each core and PCIe interface; b) some OS *kernel functions*.

Correspondingly, we collect performance monitor unit (PMU) listed in Table IV, where $\frac{unc\_cha\_tor\_occupancy.ia\_miss}{unc\_cha\_tor\_inserts.ia\_miss}$ represents the average latency from cores that miss the LLC [15]. We can deduce that data transferred by CXL can be propagated throughout the entire LLC, causing congestion in TOR due to TOR's high occupancy behavior and affecting memory accesses of other cores. This can be demonstrated from value change of *unc_cha_tor_inserts.ia_miss* and *unc_cha_tor_occupancy.ia_miss*.

INSIGHT #2: CXL traffic occupies Table Of Request (TOR) for a lengthy period, causing congestion in TOR with other multicore process.

CXL counters. We also evaluate CXL performance in both instruction level (with tools we used above) and hardware level. We collect CXL-related PMU counters to compute CXL bandwidth. The results are consistent. For example, when CXL traffic with 16 *ntst* threads run under the interference of SSD traffic with 16 *randwrite* threads in System B, the bandwidth of CXL measured by cxlMemTest is 13169 MB/s, and the result computed by *cxlcm_txc_pack_buf_inserts.mem_data* is 13342 MB/s. The close agreement between the two values proves the reliability of our data.

TABLE IV: Ratio of related event and function of NVMe.

Related Event and Function	Description	Interference to No-interference Ratio
Load_L2_Miss_Latency	Average Latency for L2 cache miss demand Loads	15.2×
L3_Cache_Access_BW	Average per-core data access bandwidth to the L3 cache	0.14×
Load_L3_Miss_Latency	Average Latency for L3 cache miss demand Loads	1.01×
unc_cha_tor_occupancy.ia_miss	Occupancy number of TOR when request from iA misses LLC	4090×
unc_cha_tor_inserts.ia_miss	Inserts number of TOR when request from iA missses LLC	155×
memmove_sse2_unaligned_erms()	Memory move with Enhanced REP MOVSB, used in memcpy()	1.96×
copyout()	Kernel-to-User buffer copy	1.29×

Kernel functions. By profiling both SSD traffic alone and with CXL, two hotspot functions show significant changes. We notice that the time taken by *memmove_sse_unaligned_erms* and *copyout* (kernel-to-user copy) increases by 96.2% and 28.7%, respectively, when CXL is in the background. These functions, called by *memcpy* to move data in parallel, indicate that memory copy operations of system kernel processing data from devices are influenced.

D. Microbenchmarks on Other Configurations

1) Sequential vs. Random Access: We also explore whether the interference differs when SSD performs sequential writes versus random writes. Results in Figure 3a highlight that SSD is more susceptible to CXL interference when performing sequential writes (up to 80.9% performance loss) compared to random writes (up to 54.5% performance loss). The reason is attributed to the greater spatial locality of data in sequential writes. Random operations presumably encounter a large number of cache line conflicts and are limited by the small number of available DDIO ways. Consequently, random operations result in an increased number of cache misses and occupy a smaller proportion of LLC space (2MB) than sequential accesses (4MB), which makes sequential writes more vulnerable to interference from the cache side, originating from CXL.

INSIGHT #3: SSD sequential writes exhibit greater LLC occupancy, which is interfered more by CXL traffic than random writes.

2) MOVDIR64B Operations: MOVDIR64B [12] is a specialized instruction that moves 64 bytes of data directly from source memory to destination memory with write atomicity. [2] Compared to *ntst*, MOVDIR64B adheres to the Write Combining (WC) memory type protocol and is weakly ordered. We wonder whether direct-store instructions like MOVDIR64B perform similarly to *ntst*. Figure 3b shows the influence of CXL MOVDIR64B operations on SSD *randwrite* in comparison with SSD performance alone and with CXL *ntst*. The results indicate that the interference from MOVDIR64B is similar to that of *ntst*, but imposes relatively less suppression on SSD performance than *ntst*. The presumable underlying cause is that unlike *ntst*, MOVDIR64B is capable of immediate eviction from the WC buffer [12]. As a result, data of MOVDIR64B spends a shorter time within the WC buffer, reducing the cost of tracking outstanding instructions. Therefore, the impact of MOVDIR64B is less significant than that of *ntst*. It is noteworthy that when the destination address of MOVDIR64B is local memory, thereby performing a normal load operation on CXL, the suppression disappears. This observation corroborates that the primary source of interference from CXL is its impact on the cache side.

INSIGHT #4: CXL MOVDIR64B operations cause less SSD performance suppression than *ntst* due to immediate WC buffer eviction, reducing instruction tracking costs.

[2]This operation is available on the 4th generation Intel® Xeon® Scalable Processor Family based on Sapphire Rapids microarchitecture.
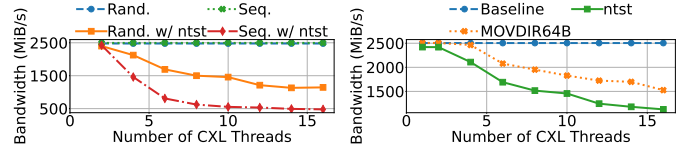
(a) NVMe randwrite and sequential write bandwidth under the interference of CXL non-temporal store.

(b) NVMe randwrite bandwidth across different memory access instructions.

Fig. 3: Microbenchmark case studies

TABLE V: Description of the selected applications.

Category	Application	Description
File-related	Filebench [32]: fileserver varmail	An open-source file system benchmark
		Large files with 1:2 read/write ratio
		Small files with 1:1 read/write ratio
	RocksDB [30]	Log-structured merge-tree database
		20-byte key size and 100-byte value size
		Metric: Operations per second
	MLPerf Storage [6]	Storage benchmark for deep learning
		Metric: Accelerator Utilization
Memory-related	GAPBS [7]	A graph computing benchmark
		PageRank Algorithm
		Metric: Trial time (lower is better)
	Redis [24]	An open-source in-memory database
		50% read and 50% update
		Metric: Operations per second
	Large Language Model	TinyLlama [35], a decoder-only LLM
		Metric: Inference time (lower is better)

IV. CXL-INTERPLAY REAL APPLICATION EVALUATIONS

We select a wide range of applications with different characteristics to study the interference problem. There are 4 types of interference scenarios: filesystem-related applications under CXL traffic, CXL-related applications under SSD traffic, MMEM-related applications under CXL traffic and CXL-related applications under MMEM traffic. We denote them as Type A, Type B, Type C and Type D respectively. The description of the selected applications is shown in Table V. The overview of the largest performance decline percentage evaluation result for each application is depicted in Figure 4. The key takeaway from this figure is that in most cases there is solid contention and interference across different access types and systems.

A. Filesystem-Related Applications

We select RocksDB database, Filebench, and machine learning workload MLPerf as our representative applications, while using CXL traffic as background to measure the interference for filesystem.

Figure 4 demonstrates that CXL's traffic has a considerable effect on the performance of most applications (Type A), with an observed
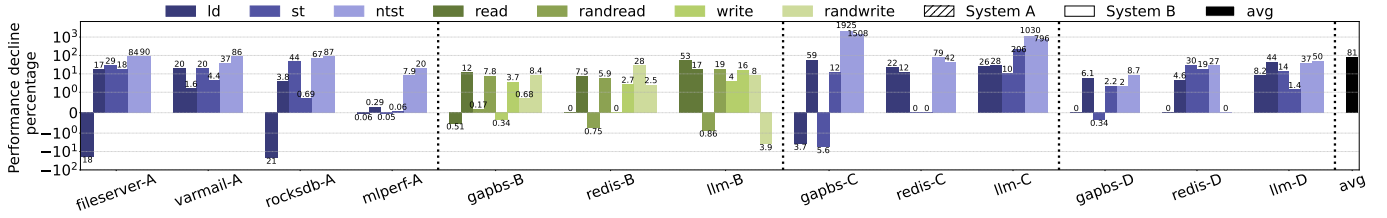
Fig. 4: CXL-related interference on real applications. The suffixes "-A/B/C/D" indicate the type to which the experiment belongs. We use different colors to differentiate background traffic types. For Type A, C and D, the background runs memory traffic which have three different types: *ld*, *st* and *ntst*. As for Type B, its background is file system and has four traffic types: *read*, *randread*, *write* and *randwrite*. We also use hatches to differentiate results from different systems. The height of the bars represents the maximum percentage of performance decline observed across all number of background thread count configurations for the corresponding scenarios. The y-axis is on logarithmic scale. [3]
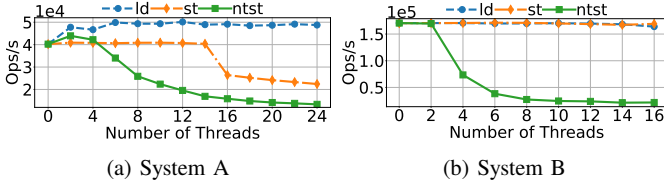
Fig. 5: The performance (operations per second) of RocksDB under CXL traffic. The x-axis indicates the number of background threads.
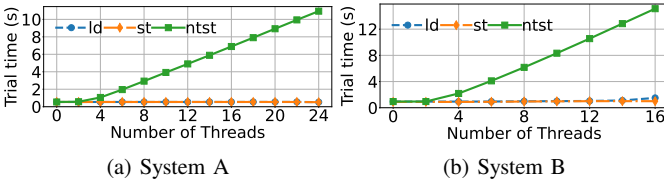

Fig. 6: The performance (trial time) of GAPBS benchmark in Type C. The x-axis indicates the number of background threads.

detrimental impact reaching up to 90% for *ntst*. For *ld* and *st*, the impact can also be as high as 20% and 44% respectively. Moreover, the results presented in Figure 5 indicate an interesting and unexpected phenomenon that CXL's ld traffic can even enhance RocksDB's performance by up to 21.2% on system A.

Discussion. It can be observed that an increase in the thread count—and consequently the bandwidth—of the background CXL *ld* workload is associated with improved performance. To verify this, two other methods are applied to adjust CXL memory bandwidth: periodic cache flushing and injecting numerous **nop** instructions. We observe that the performance enhancement is directly linked to CXL memory bandwidth, occurring only when it exceeds 8 GB/s. Between 8 GB/s and 20 GB/s, the promotion grows linearly, while beyond that, the enhancement plateaus.

Albeit system-specific, we try to explore the root cause for promotion in System A with performance counters. for RocksDB, running a fixed number of operations both with and without CXL *ld* interference. The most notable observation is a 24% reduction in the number of instructions, especially memory instructions, under interference. This reduction in instructions plays a key role in the performance improvement.

INSIGHT #5: SSD workload bandwidth can benefit from a reduced instruction count when having sufficient CXL *ld* memory bandwidth as background.

B. Memory-Related Applications

We select graph benchmark GAPBS, in-memory database Redis and an LLM as our representative memory applications, covering Type B, C and D. It can be clearly observed that CXL memory's operations have a substantial impact on MMEM (Type C), consistent

with microbenchmark results. Figure 6 shows a representative with up to 20.8× performance decline.

Conversely, the impact of MMEM on CXL memory is significantly less pronounced than that observed in Type C. We hypothesize that the observed limited impact is due to the fact that, although MMEM traffic is high, the low latency of MMEM does not result in significant consistence overhead, allowing CXL to operate normally.

Regarding SSD applications, their impact on CXL memory is insignificant, potentially due to the fact that the LLC consumed by SSD is a relatively small section (4MB) in comparison to those occupied by MMEM (50MB). Nevertheless, it should be noted that in some cases, particularly under *read* operation background, up to 12% decline in CXL performance can be observed.

INSIGHT #6: The interference from CXL memory to MMEM is significant. However, MMEM and SSD have mild impact on CXL memory in most cases due to DRAM's lower latency and SSD's lower LLC utilization.

V. INTERVENT THE INTERFERENCE AND EXPERIMENT ON CXL-RELATED REGULATION

Given the interference discussed in Section III and Section IV, we have observed and analyzed various interplay between CXL and SSD, as well as CXL and MMEM. In this section, we provide different trials and solutions to mitigate the interference with real CXL hardware. We consider using CPU usage restriction, such as time quota allocating, frequency scaling, memory bandwidth restriction, etc., to demonstrate the effectiveness of regulation on CXL real hardware.

A. CPU Usage Restrictions

To validate the functionality of CPU usage restriction, we choose three scenarios of severe interference: MMEM *ld*, MMEM GAPBS and filebench *varmail*, all influenced by 16 CXL *ntst* threads. We first use the counters mentioned in Section III-C and bandwidth monitoring tools [11] to identify the processes causing high CXL traffic and LLC occupancy periodically. Then we use Linux's *cgroups* [14] to limit their CPU quota. If the process still causes severe impact in the next period, we further adjust its resource allocation to achieve dynamic control.

As illustrated in Figure 7, reducing the allowed CPU usage quota leads to a recovery in MMEM bandwidth and a decrease in latency. Specifically, when the distribution of CPU cycles for CXL is restricted to 100% (1/16 of the time originally occupied), the MMEM bandwidth recovers to 190 GB/s, which is approximately 92% of the bandwidth observed without interference. Additionally,

---

[3]In certain scenarios, the results are unstable and exhibit significant fluctuations. In such cases, the maximum decline is not meaningful, and we represent it as 0 in the figure.
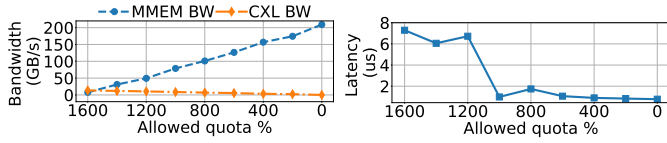
Fig. 7: Bandwidth and latency of MMEM when applying different CPU usage quotas to CXL threads. The horizontal axis is the allowed percentage of CPU usage.

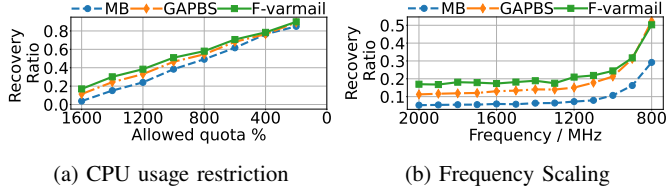(a) CPU usage restriction   (b) Frequency Scaling

Fig. 8: Effectiveness of the two methods. Vertical axis is normalized to the performance without interference.

latency returns to nearly its original level. This improvement in MMEM performance comes at the cost of a bandwidth reduction of about 10 GB/s on CXL. The results for the three applications are illustrated in Figure 8a. The linear and consistent impact observed across the three applications is due to the characteristic of CPU usage regulation, which influences a wide range of instructions. This approach facilitates a satisfactory and adjustable degree of recovery, albeit with varying degrees of performance decline in the background process.

B. Frequency Scaling

We also consider frequency scaling as a potential solution to mitigate the interference between CXL and MMEM. As is shown in Figure 8b, the recovery of MMEM threads is observed to be within the range of 30% to 52% when the background is constrained to very low frequency. This phenomenon can be explained by the fact that, except for very low frequency cases, memory instructions continue to fill the load/store queue. Consequently, memory-intensive tasks tend to demonstrate an exponential effect as the frequency decreases, yet still exhibit a suboptimal recovery rate. Although this method is less effective, it does not cause any damage to the memory-only background program.

C. Memory Bandwidth Restriction

The solution of regulating time slice and frequency is purely software-based and universal. But it also restricts the execution of memory-unrelated instructions, which causes additional performance degradation. On the other hand, memory bandwidth restriction presents a promising approach to mitigate interference between CXL and MMEM. We evaluate the effectiveness of using Intel MBA [10], which is a feature that allows users to allocate a portion of the memory bandwidth to specific processes. Limited by the I/O block configuration, it is only possible to restrict the CXL bandwidth to approximately less than half of its original value, which corresponds to a reduction of 5GB/s. This approach proves highly effective, enabling the three applications to recover to 99%, 95%, and 98% respectively, without affecting the compute instructions. However, this method is unable to provide control in a responsive manner and takes seconds to take effect, making it more suitable for implementation on the CXL device side.

D. Discussions on Other Methods

There are multiple solutions proposed to restrict the noisy neighbors and mitigate their negative impact on other threads. Apart

from the methods discussed above, other methods include 1) cache partitioning, which is enabled by CPU vendors, such as Intel's Cache Allocation Technology (CAT) and 2) specialized memory bandwidth restriction, such as AMD's Slow Memory Bandwidth Allocation (SMBA) [2] targeting CXL devices, which has not been released. However, cache partitioning has minimal impact on operations that bypass the cache hierarchy, such as *ntst* operations, where it proves ineffective.

For hardware implementation, additional latency can be inserted into CXL device side after CPU has detected actual interference and then write the control register to change the added latency. However, this modification can highly depend on specific hardware which makes it harder to quickly deploy and generalize to different vendors.

VI. PRACTICAL TAKEAWAYS

Given the characterization and regulation experiments of CXL interference above, we provide the following takeaways for software (S) and hardware (H) developers respectively when developing future CXL-related systems:

(S1) Limit the number of threads or bandwidth of CXL *ntst* operations when possible to minimize their impact on MMEM and SSD workloads. (#1)

(S2) When running under CXL *ntst* traffic, SSD workloads should consider the trade-off between different access patterns such as random vs. sequential writes to maintain bandwidth. (#3)

(S3) For some systems, it is beneficial of lifting bandwidth to schedule SSD workloads with workloads having intensive CXL memory *ld* operations. (#5)

(H1) Hardware vendors can develop specialized TOR for slow tier memory to mitigate CXL contention with MMEM and SSD workloads. (#2)

(H2) To regulate the interference, it is advised to monitor and regulate on the CXL.mem device side with a more fine-grained MBA-like regulator.

VII. CONCLUSION

Real devices utilizing the emerging CXL technology are now finally commercially available. Researchers have actively characterized their features as isolated components. In this paper, we propose that since CXL devices need to interact with other components, it is necessary to characterize them in a coexisting manner. We develop customized microbenchmarks as well as real applications to characterize the interference problem. Our evaluation shows that this interference can cause a performance drop of up to 93.2%, along with other interesting observations. We also delve deep into the root causes of these observations. We provide concrete insights for readers and takeaways for developers. To mitigate the impact of this issue, we propose software mechanisms to manage CXL traffic. The results demonstrate the efficacy and trade-off of several methods.

References

[1] R. Abdullah, H. Lee, H. Zhou, and A. Awad, "Salus: Efficient security support for cxl-expanded gpu memory," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 1–15.

[2] *AMD64 Technology Platform Quality of Service Extensions*, Advanced Micro Devices, Inc., February 2022.

[3] J. Axboe, "Flexible i/o tester," 2022. [Online]. Available: https://github.com/axboe/fio

[4] E. L. Bai and S. A. Raut, "An analysis on compute express link with rich protocols and use cases for data centers," in *International Conference on Image Processing and Capsule Networks*. Springer, 2022, pp. 787–800.

[5] D. Bakhvalov, *Performance Analysis and Tuning on Modern CPUs: Squeeze the last bit of performance from your application.* Independently published, 2020.

[6] O. Balmau, "Characterizing i/o in machine learning with mlperf storage," *ACM SIGMOD Record*, vol. 51, no. 3, pp. 47–48, 2022.

[7] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017. [Online]. Available: https://arxiv.org/abs/1508.03619

[8] C. E. L. Consortium *et al.*, "Compute express link specification," 2020.

[9] I. Corporation, "Cxl protocol hard ip functionality accelerates a wide range of data-centric workloads," https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html.

[10] I. Corporation, "Intel® resource director technology (intel® rdt) framework," https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html, 2015.

[11] I. Corporation, "Intel performance counter monitor," 2024, accessed: 2024-08-02. [Online]. Available: https://github.com/intel/pcm

[12] I. Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, M-U*. Intel Corporation, 2024.

[13] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, "Ai and memory wall," *IEEE Micro*, 2024.

[14] T. Heo, *Control Groups v2*, 2015. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[15] *4th Gen Intel® Xeon® Scalable Processor XCC (Codename Sapphire Rapids) Uncore Performance - Monitoring Guide*, Intel Corporation, April 2024.

[16] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 17–34.

[17] J. Liu, X. Wang, J. Wu, S. Yang, J. Ren, B. Shankar, and D. Li, "Exploring and evaluating real-world cxl: Use cases and system adoption," 2024. [Online]. Available: https://arxiv.org/abs/2405.14209

[18] MICRON, "Cz120 memory expansion module," https://www.micron.com/products/memory/cxl-memory.

[19] S. H. NEWSROOM, "Sk hynix introduces industry's first cxl-based computational memory solution (cms) at the ocp global summit," https://news.skhynix.com/sk-hynix-introduces-industrys-first-cxl-based-cms-at-the-ocp-global-summit/, 2022.

[20] A. Puri, K. Bellamkonda, K. Narreddy, J. Jose, V. Tamarapalli, and V. Narayanan, "Dracksim: Simulating cxl-enabled large-scale disaggregated memory systems," in *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2024, pp. 3–14.

[21] A. Raad, L. Maranget, and V. Vafeiadis, "Extending intel-x86 consistency and persistency: formalising the semantics of intel-x86 memory types and non-temporal stores," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498683

[22] B. Ramesh, N. Contini, N. Alnaasan, K. K. Suresh, M. Abduljabbar, A. Shafi, H. Subramoni, and D. K. D K Panda, "Hint: Designing cache-efficient mpi_alltoall using hybrid memory copy ordering and non-temporal instructions," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 802–813.

[23] S. Ryu, S. Kim, J. Jun, D. Moon, K. Lee, J. Choi, S. Kim, H. Kim, L. Kim, W. H. Choi, M. Nam, D. Hwang, H. Roh, and Y. Joo, "System optimization of data analytics platforms using compute express link (cxl) memory," in *2023 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2023, pp. 9–12.

[24] S. Sanfilippo, "Redis," 2009. [Online]. Available: https://redis.io/

[25] S. Semiconductor, "Samsung develops industry's first cxl dram supporting cxl 2.0," https://semiconductor.samsung.com/news-events/news/samsung-develops-industrys-first-cxl-dram-supporting-cxl-2-0/, 2022.

[26] D. D. Sharma, "Novel composable and scaleout architectures using compute express link," *IEEE Micro*, vol. 43, no. 2, pp. 9–19, 2023.

[27] K. Song, J. Yang, S. Liu, and G. Pekhimenko, "Lightweight frequency-based tiering for cxl memory systems," *arXiv preprint arXiv:2312.04789*, 2023.

[28] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying cxl memory with genuine cxl-ready systems and devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 105–121. [Online]. Available: https://doi.org/10.1145/3613424.3614256

[29] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao *et al.*, "Exploring performance and cost optimization with asic-based cxl memory," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 818–833.

[30] F. R. team, "A persistent key-value store for fast storage environments," https://rocksdb.org/, 2024.

[31] M. Technology, "Cxl memory expander controller (mxc)," https://www.montage-tech.com/MXC.

[32] T. Vasily, "Filebench: A flexible framework for file system benchmarking.; login," *The USENIX Magazine*, vol. 41, p. 6, 2016.

[33] Y. Yang, P. Safayenikoo, J. Ma, T. A. Khan, and A. Quinn, "Cxlmemsim: A pure software simulated cxl. mem for performance characterization," *arXiv preprint arXiv:2303.06153*, 2023.

[34] J. Yi, B. Dong, M. Dong, R. Tong, and H. Chen, "{MT^ 2}: Memory bandwidth regulation on hybrid {NVM/DRAM} platforms," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022, pp. 199–216.

[35] P. Zhang, G. Zeng, T. Wang, and W. Lu, "Tinyllama: An open-source small language model," 2024.

[36] Y. Zhong, D. S. Berger, C. Waldspurger, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury, and A. Cidon, "Managing memory tiers with cxl in virtualized environments," in *Symposium on Operating Systems Design and Implementation*, 2024.