MT²: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms

Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen,
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

https://www.usenix.org/conference/fast22/presentation/yi-mt2

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.
February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms

Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Non-volatile memory (NVM) has emerged as a new memory media, resulting in a hybrid NVM/DRAM configuration in typical servers. Memory-intensive applications competing for the scant memory bandwidth can yield degraded performance. Identifying the noisy neighbors and regulating the memory bandwidth usage of them can alleviate the contention and achieve better performance. This paper finds that bandwidth competition is more severe on hybrid platforms and can even significantly degrade the total system bandwidth. Besides the absolute bandwidth, the competition is also highly correlated with the bandwidth type. Unfortunately, NVM and DRAM share the same memory bus, and their traffic is mixed together and interferes with each other, making memory bandwidth regulation a challenge on hybrid NVM/DRAM platforms.

This paper first presents an analysis of memory traffic interference and then introduces MT^2 to regulate memory bandwidth among concurrent applications on hybrid NVM/DRAM platforms. Specifically, MT^2 first detects memory traffic interference and monitors different types of memory bandwidth of applications from the mixed traffic through hardware monitors and software reports. MT^2 then leverages a dynamic bandwidth throttling algorithm to regulate memory bandwidth with multiple mechanisms. To expose such a facility to applications, we integrate MT^2 into the cgroup mechanism by adding a new subsystem for memory bandwidth regulation. The evaluation shows that MT^2 can accurately identify the noisy neighbors, and the regulation on them allows other applications to improve performance by up to 2.6× compared to running with unrestricted noisy neighbors.

1 Introduction

Emerging fast, byte-addressable NVM, such as phase-change memory (PCM) [41, 52], STT-MRAM [25, 37], Memristor [56], and Intel's 3D-XPoint [55], are promising to be employed to build fast cloud data centers. Intel Optane DC Persistent Memory, the first commercially available NVM product, has been released in 2019 [28, 33] and deployed in cloud environments, such as Google Cloud [13].

NVM has attracted many research efforts on exploring its usage scenarios. Consequently, an increasing number of NVM-aware file systems [19–21, 38, 53, 54, 60, 62, 63], NVM programming libraries [9, 70], NVM data structures [26, 42, 46, 64, 73] and NVM-based databases [5, 10, 45] have been proposed and studied, which in turns accelerates the widespread deployment of NVM. NVM is being deployed in data centers as fast byte-addressable storage or large-volume runtime memory that lies side-by-side with the volatile DRAM, resulting in hybrid NVM/DRAM platforms.

However, the hybrid NVM/DRAM platforms exacerbate the noisy neighbor problem. In cloud environments, a physical platform may be shared by many users. Applications, containers, or VMs of different users inevitably share the same memory bus on the platform. Some applications may overutilize memory bandwidth, either accidentally or intentionally, and become the noisy neighbors that significantly affect the performance of other applications. On hybrid NVM/DRAM platforms, both NVM and DRAM are attached to the memory bus. As a result, different applications compete for the limited memory bandwidth, and different kinds of memory traffic interfere with each other, reducing the overall performance of all applications on the hybrid NVM/DRAM platform.

Memory bandwidth regulation is one common approach that reduces the interference of memory bandwidth usage to mitigate the noisy neighbor problem. With the commercial use of NVM in cloud data centers, the need for memory bandwidth regulation on hybrid platforms is imminent. However, several significant challenges hinder memory bandwidth regulation on NVM/DRAM hybrid platforms.

The first challenge is memory bandwidth asymmetry. On NVM/DRAM hybrid platforms, different memory accesses (i.e., DRAM reads, DRAM writes, NVM reads and NVM writes) yield different maximal memory bandwidth. The actual available memory bandwidth heavily depends on the proportions of different kinds of memory accesses in the workload. Thus, it is no longer appropriate to assume a static maximal memory bandwidth and disregard the difference between different memory accesses like in prior work [67–69].

Especially, the maximal NVM bandwidth is usually relatively smaller than DRAM bandwidth. Besides, we find that different types of memory accesses interfere with each other differently, making it even more difficult to estimate the available memory bandwidth under various workloads. Thus, the assumption that all memory accesses are equal (as in prior work [67–69]) does not hold anymore.

The second challenge stems from the fact that NVM shares the memory bus with DRAM on existing NVM/DRAM hybrid platforms [3]. On existing hybrid platforms, NVM traffic and DRAM traffic are inevitably mixed and difficult to separate. With the mixed memory traffic, monitoring different kinds of memory bandwidth on a per-process basis become almost impossible [49], which invalidates existing hardware and software regulation approaches designed for DRAM.

The third challenge is inadequate hardware and software mechanisms for memory regulation. As both NVM and DRAM are directly accessible by CPU load/store instructions, counting and throttling each memory access is impractical for the sake of performance. CPU vendors, such as Intel, provide hardware mechanisms to regulate the memory bandwidth. However, the bandwidth restriction is coarse-grained and qualitative, which is insufficient for precise memory bandwidth regulation. Some other approaches, such as frequency scaling and CPU scheduling, may provide relatively finer-grained bandwidth adjustment. However, they are also qualitative and slow down both computation and memory accesses, thus inefficient for the overall platform performance.

In this paper, we reveal severe bandwidth interference problems in hybrid memory platforms and propose MT^2 (short for Memory Traffic Throttle) to address the above challenges. MT^2 collaboratively leverages several hardware and software techniques to monitor real-time bandwidth of different types of memory accesses. To regulate memory bandwidth with non-static maximal memory bandwidth, MT^2 proposes a dynamic memory bandwidth throttling framework, combining both hardware and software techniques to provide efficient memory bandwidth regulation.

We have implemented MT^2 as a new subsystem in the existing Linux control groups (cgroups) and applied MT^2 to mitigate the noisy neighbor problem and demonstrate MT^2's effectiveness in two more scenarios: memory bandwidth allocation and cloud SLO guarantee. Performance evaluation shows that MT^2 can effectively regulate memory bandwidth on hybrid platforms with nearly zero performance overhead.

In summary, the contributions of this paper include:

- A survey uncovering the problem of memory bandwidth interference that leads to notable performance churn for memory-intensive applications on hybrid NVM/DRAM platforms (§2);
- The first study on existing hardware and software memory bandwidth regulation mechanisms on hybrid NVM/DRAM platforms (§3.3.1);
- The design and implementation of MT^2, the first comprehensive system that efficiently and effectively regulates memory bandwidth on hybrid NVM/DRAM platforms with thread-level granularity (§3 and §4);
- Detailed evaluation of MT^2 in noisy neighbor suppression and other two scenarios (§5) on Intel Optane PM to illustrate MT^2's effectiveness and overhead (§6).

2 Background

2.1 Noisy Neighbors

In complex modern multi-tenant cloud environments, memory bandwidth can significantly impact applications' overall performance. In a cloud data center, some applications may over-utilize memory bandwidth, which will affect the performance of other applications. These applications that over-utilize memory bandwidth are usually called *noisy neighbors*, and the affected applications are the *victims*.

Two strategies can mitigate the noisy neighbor problem. The *prevention* strategy proactively sets bandwidth limits for applications to keep anyone from being a potential noisy neighbor. The *remedy* strategy monitors the system for the presence of noisy neighbors and identifies then limits the bandwidth usage of appeared noisy neighbors. Both strategies require a system to monitor applications' bandwidth usage and/or the system-wide traffic interference level and provide effective mechanisms to limit applications' memory traffic.

2.2 NVM

The release of Intel Optane DC Persistent Memory (Optane PM) marks the widespread commercial deployment of NVM [28, 33]. With Intel's proprietary DDR-T protocol [33], Optane PM can be directly accessed via CPU load/store instructions. However, the actual bandwidth of NVM is still far below DRAM [34].

Before the public release of Optane PM, NVM has been widely studied in academia and industry. Some NVM-aware file systems, such as PMFS [53], NOVA [62,63], SoupFS [21], Strata [38], SplitFS [35] and ZoFS [20], are proposed to provide file abstraction over NVM. Applications can create files on these file systems and map the files using `mmap` to access NVM directly. For example, Marathe et al. [45] modify Memcached [7], a popular high-performance memory object caching system, to run upon NVM using files and `mmap`.

However, managing persistent files by hand can be laborious. Intel develops Persistent Memory Development Kit (PMDK) [9], which is a suite of open-source libraries to simplify the programming model of NVM. With PMDK, programmers do not need to manage persistent files by themselves. Instead, they can utilize PMDK abstractions, such as objects, transactions and simple persistent data structures, to develop NVM-aware applications more easily. Many in-memory or storage systems are ported to NVM using PMDK, such as PmemKV [5] and Pmem-RocksDB [10].

NVM is increasingly deployed in data centers. For example, SAP HANA has deployed Optane PM in its data plat-

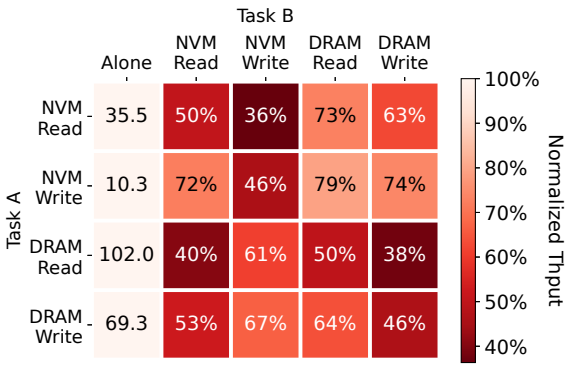

Figure 1: The impact of memory interference of two tasks. The first column is the bandwidth of Task A when it runs alone (in GB/s) and the last 4 columns are the bandwidth of Task A as a percentage of the first column when two tasks run simultaneously and compete for the bandwidth. Task B decrease Task A' throughput by 21 to 64%. The darkest block at the top row shows that NVM write bandwidth affect NVM read bandwidth significantly. Different types of bandwidth affect others differently.

forms [14]. Google Cloud has deployed Optane PM on its virtual machines in public clouds [13].

2.3 Memory Bandwidth Interference

Despite the advantages NVM has brought to the data center, the use of NVM on the hybrid NVM/DRAM platforms increases the complexity of the memory bandwidth interference due to the fact that NVM and DRAM share the memory bus.

To illustrate the impact between different types of bandwidth, we conducted an experiment in which two tasks run different kinds of workloads simultaneously. In the experiment, we run two Flexible I/O tester [2] (`fio`) workloads for the tasks to compete for the memory bandwidth. We test four workloads, namely *NVM Read*, *NVM Write*, *DRAM Read*, and *DRAM Write*, and use the mmap engine for the DRAM workloads and libpmem for the NVM workloads. The experiment setup is described in §6. To fully utilize the memory bandwidth, we use fourteen cores to run each workload, except for *NVM Write*. We use six cores to run *NVM Write* workload because its bandwidth drops significantly with more cores due to its own bandwidth competition.

We first run Task A alone and then run two tasks together with different workload combinations to illustrate the impact. To avoid the contention of CPU cache, we also leverage Intel CAT [4] to make each task run on different cache partitions in the experiment. Thus, the performance degradation in the figure is simply caused by memory bandwidth interference.

Figure 1 shows the results. For Task A (i.e., each row in the figure), the throughput (GB/s) of running alone is used as the baseline, as listed in the first column, and the throughput running simultaneously with Task B is normalized to the baseline. Thus, smaller numbers (i.e., the darker blocks) indicate a more significant impact of Task B.

We make two observations from the results.

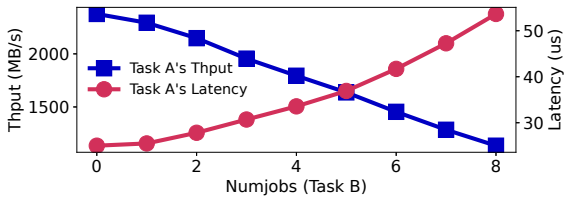

Figure 2: Relationship between bandwidth and latency of Task A (one-job NVM-Write `fio`) when running simultaneously with different number of Task B (NVM-Write `fio`). The bandwidth and latency of task A are negatively correlated. Notice that the Y-axis does not start from zero.)

1. *The impact of memory interference is closely related to the type of memory access.* Tasks that occupy a smaller bandwidth may have a more significant impact on other tasks than those with a larger bandwidth. A 102GB/s DRAM-read Task B can only reduce the bandwidth of an NVM-read Task A to 73% of the original, while an NVM-write Task B with only 10.3GB/s can bring it down to 36%. This observation indicates that the ability to distinguish between different bandwidth types is vital on hybrid platforms.
2. *NVM accesses affect other tasks more severely than DRAM accesses.* When Task B runs a 35.5GB/s NVM read workload, it drops the bandwidth of different Task A to 40%-72% (the second column in the figure) of what it would have been running alone. However, when Task B runs a 102GB/s DRAM read workload, Task A's bandwidth drops to only 50%-79% (the fourth column in the figure) of the original bandwidth. In particular, a 10.3GB/s NVM write task B can severely degrade the performance of other tasks. NVM writes can lead to severe interference with minimal absolute bandwidth, followed by NVM reads and finally DRAM accesses. In other words, applications that write NVM a lot are more likely to become the noisy neighbors and affect others.

While investigating the memory bandwidth interference, we also check the relationship between a task's throughput and latency. Figure 2 shows the throughput and latency of a one-job `fio` with the NVM-Write workload (Task A) when running simultaneously with Task B (NVM-Write `fio` with variable numjobs). As the numjobs of Task B grows, the bandwidth of Task A gradually decreases (due to the growth of bandwidth interference), while the latency of Task A increases. Together with the evaluation of other memory access type combinations, the results lead to another observation that *the memory access latency is negatively correlated to the bandwidth usage*, which indicates that we can detect the memory bandwidth interference by measuring the latency of different types of memory accesses.

2.4 Memory Bandwidth Monitoring (MBM)

Intel Memory Bandwidth Monitoring (MBM) [48] is a feature that allows monitoring bandwidth from the L3 cache to the next level of the memory hierarchy system, which can be

DRAM or NVM. It provides a hardware-level measurement of memory bandwidth on each logical core.

Each logical core can be assigned with a resource monitoring ID (RMID), and a group of logical cores can be assigned with the same RMID. The underlying hardware tracks memory bandwidth with the RMID and groups the memory bandwidth of processors with the same RMID. On a platform with the non-uniform memory access (NUMA) architecture, the MBM hardware on each NUMA node tracks two types of memory bandwidth for each RMID: the local external bandwidth and the total external bandwidth, indicating the memory traffic to the local NUMA node and all NUMA nodes respectively. System programmers can access model-specific registers (MSRs) to get the tracked bandwidth. To get the system-wide memory bandwidth of an RMID, programmers need to read the tracked total bandwidth from all NUMA nodes and add them together.

2.5 Memory Bandwidth Allocation (MBA)

Intel Memory Bandwidth Allocation (MBA) [29] is a hardware feature that provides indirect and approximate control over memory bandwidth with negligible overhead. MBA introduces a programmable request rate controller between each physical core and the shared L3 cache. The controller throttles the memory bandwidth usage by inserting delays to the memory requests. MBA defines throttling values to indicate how much delay is imposed. Due to the delay mechanism, the same throttling value might behave differently on applications with different memory access patterns [29]. The specific throttling values vary on different platforms. On our platform, the throttling values range from 10 to 100, with a precision of 10. For MBA, Intel also exposes a set of Classes of Service (CLOS) [29] into which threads can be assigned. To use MBA, administrators need to set a throttling value to a CLOS, after which all threads in the CLOS will be throttled.

3 MT^2 Design

3.1 Overview

To regulate bandwidth efficiently on a hybrid NVM/DRAM platform, we design a hybrid bandwidth regulation system called MT^2. Figure 3 shows the architecture of MT^2, which is designed to work in the kernel space. Though some functionalities of MT^2 can be implemented in user space, the kernel space environment makes it much easier and more efficient for MT^2 to access hardware features, cooperate with other kernel components, and put constraints on user-space threads.

System administrators communicate with MT^2 via exposed pseudo-filesystem interfaces in user space. Administrators can classify threads into different groups (same as cgroups) and specify a policy to regulate each group's bandwidth. We call these groups TGroups (i.e., Throttling Groups), which are the target of bandwidth monitoring and restriction in MT^2.

MT^2 consists of two parts: the monitor and the regulator. With data collected from VFS, PMU (Performance Monitoring Unit), and MBM, the monitor divides it into four types and forwards them and the interference information to the regulator. According to the monitored data and the regulation policy, the regulator makes decisions to limit the bandwidth with two mechanisms: adjusting the MBA throttling values and changing CPU quotas. MT^2 adopts a dynamic bandwidth throttling algorithm that constantly monitors and adjusts restrictions based on the real-time bandwidth and the interference level.

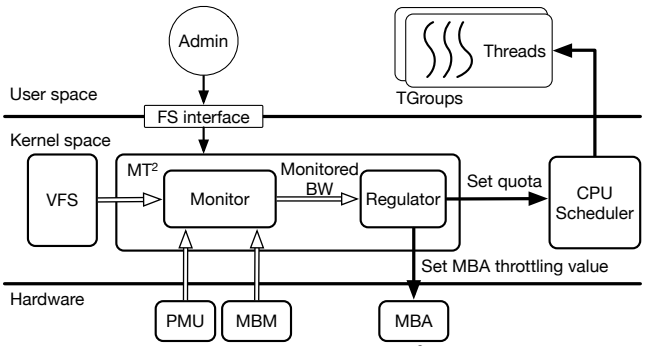

Figure 3: The overview architecture of MT^2. Threads are classified into TGroups (the unit of NVM bandwidth monitoring and restriction). The monitor computes NVM bandwidth with the data from VFS, MBM and PMU, and then pass the result to the regulator, who is responsible for restricting a TGroup's bandwidth with different mechanisms.

MT^2 provides two strategies to mitigate the noisy neighbor problem (*prevention* and *remedy* in § 2.1) to cope with different scenarios. For prevention, administrators are asked to set bandwidth caps for each TGroup. MT^2 monitors the precise real-time bandwidth and enforces all groups not to use more bandwidth than the caps. However, several TGroups that do not exceed the caps together may still cause strong bandwidth interference, which can be identified and restricted by the remedy strategy. The two strategies are orthogonal; thus, when and how to use the two strategies depends on the specific scenarios.

3.2 The Monitor

The monitor distinguishes different types of bandwidth with process granularity and detects the current memory interference level of the system. Unfortunately, existing hardware technology cannot achieve this directly [49]. For example, Intel MBM cannot distinguish between NVM bandwidth and DRAM bandwidth. The IMC performance counter [22] can help get different types of real-time bandwidth, but only with memory channel granularity rather than process granularity. Thus, the MT^2 monitor leverages various hardware and software techniques jointly.

3.2.1 Bandwidth Estimation

The monitor needs to get accurate or estimated bandwidth of each access type (i.e., BW_{DR} for DRAM reads, BW_{NR} for NVM reads, BW_{DW} for DRAM writes and BW_{NW} for NVM writes), so that the regulator can use these information to

decide whether and how to restrict each TGroup's memory bandwidth usage to avoid or suppress noisy neighbors.

MT^2 calculates the precise BW_{NR} of each process by retrieving the number of local NVM reads via the `ocr.all_data_rd.pmm_hit_local_pmm.any_snoop` PMU event counter and multiplying the value by the cache line size (64B). MT^2 calculates BW_{DR} similarly via the `ocr.all_data_rd.l3_miss_local_dram.any_snoop` PMU counter for DRAM reads of each process [30].

However, MT^2 cannot get BW_{DW} and BW_{NW} via PMU, since no similar performance events exist for write instructions. Fortunately, we can leverage MBM to monitor each TGroup's total memory access bandwidth, which is the sum of BW_{DR}, BW_{NR}, BW_{DW} and BW_{NW}. Given that we can calculate the precise value of BW_{DR} and BW_{NR} via PMU, we only need to know one of BW_{DW} and BW_{NW} or the ratio between them to calculate the two values via simple arithmetic.

We choose to calculate BW_{NW} of a TGroup by collecting the amount of NVM writes periodically since user-space applications can write to NVM in only two ways: the file APIs (such as `write`) and the CPU `store` instructions after memory mapping the file. For file APIs, MT^2 hooks the VFS in the kernel and tracks the amount of NVM writes for each TGroup. For memory-mapped accesses, we propose two different approaches according to whether the applications on the platform are trusted.

Trusted applications. Many cloud applications (such as those in private clouds) are from trusted users or cooperations; thus, we can rely on these trusted applications to collect and report to MT^2 its amount of writes to memory-mapped NVM. To facilitate the process, we provide a modified PMDK [9], which is the official and most popular library for NVM programming on Intel's NVM. Specifically, we hook the PMDK APIs that explicitly flush cache lines to NVM or perform non-temporal memory writes (e.g., `movnt`), by calculating and adding the amount of NVM writes to per-thread counters. To report the counters to MT^2, each process sets up a shared page with the kernel, and each thread in the process writes its per-thread counter value to a different slot in the page. MT^2 in the kernel checks the counters periodically and calculates the bandwidth of each TGroup.

To collaborate with MT^2, applications built on PMDK can directly link to our modified PMDK without source code modification; other applications are required to collect and report NVM writes by themselves, which should be a simple task since our modification to PMDK is merely 43 lines.

With the reported writes to mapped NVM and the NVM writes via file APIs, MT^2 calculates BW_{NW} for each TGroup. With the BW_{NW}, MT^2 further calculates the BW_{DW} by $BW_{DW} = BW_{Total} - BW_{DR} - BW_{NR} - BW_{NW}$.

Untrusted applications. Untrusted applications may not report their NVM write bandwidth faithfully. Thus, we provide another approach to roughly distinguish NVM writes and DRAM writes without the collaboration of applications.

We leverage Processor Event Based Sampling (PEBS) [31], an efficient sampling feature in modern Intel processors, to sample each TGroup's memory writes (`mem_inst_retired.all_stores`) with the target addresses. By comparing the sampled addresses to the address ranges of NVM, we can figure out the ratio of sampled writes to NVM and DRAM, with which we calculate BW_{NW} and BW_{DW} roughly.

Note that the BW_{NW} and BW_{DW} we calculated via PEBS are not precise due to the shadow effect [65]. But it would be sufficient for MT^2 to identify which TGroup is more likely to be the noisy neighbors.

3.2.2 Interference Detection

Even given the accurate bandwidth usage of four types of memory accesses, it is difficult to determine whether the bandwidth interference occurs and its severity, since both the decrease of memory access demand and the presence of noisy neighbors can cause an application to utilize less bandwidth.

Instead of detecting memory interference via memory bandwidth, MT^2 proposes to detect the interference level by measuring the latency of different kinds of memory accesses, which is supported by the observation that the memory access latency is negatively correlated to the bandwidth (§2.3).

We measure four types of memory accesses separately. For reads, we derive the latency from four performance events, `unc_m_pmm_rpq_occupancy.all` (RPQ_O), `unc_m_pmm_rpq_inserts` (RPQ_I), `unc_m_rpq_occupancy`, and `unc_m_rpq_inserts`. The latency of NVM reads can be calculated by RPQ_O/RPQ_I. The DRAM read latency can be obtained similarly. For writes, MT^2 periodically issues a few NVM and DRAM write requests and measures their completion time to obtain the latency of both types of write requests.

We set a threshold to determine whether bandwidth interference occurs. When the latency of a certain access request exceeds the corresponding threshold, relatively severe interference occurs on the platform and affects this type of memory access (as shown in Listing 1). The threshold can be tuned across different platforms by measuring the relationship between bandwidth and latency under different interference levels. On our platform, we use the latency at a 10% reduction in throughput as the threshold (THRESHOLD in Listing 1).

Listing 1: Interference detection

```
def detect_interference():
  for bt in bandwidth_type:
    if latency[bt] > THRESHOLD[bt]:
      return true
  return false
```

3.3 The Regulator

Monitoring the bandwidth information is the first step towards bandwidth regulation. The following step is to restrict the bandwidth a TGroup can occupy, which is handled by the regulator. The regulator takes the interference level and the

Table 1: Performance events

Performance Event Name	Description	Where we use (if not, why)
ocr.all_data_rd.pmm_hit_local_pmm.any_snoop	per-core: local NVM read	Bandwidth Estimation (§3.2.1)
mem_load_retired.local_pmm	per-core: memory load instructions retired that hit local NVM	No, the results are not precise without disabling the hardware prefetcher
ocr.all_data_rd.l3_miss_local_dram.any_snoop	per-core: local dram read	Bandwidth Estimation (§3.2.1)
mem_inst_retired.all_stores	per-core: all memory store instruction retired	Bandwidth Estimation (§3.2.1)
unc_m_pmm_rpq_occupancy.all	per-socket: NVM read pending queue occupancy	Interference Detection (§3.2.2)
unc_m_pmm_rpq_inserts	per-socket: NVM read pending queue inserts	Interference Detection (§3.2.2)
unc_m_pmm_wpq_occupancy.all	per-socket: NVM write pending queue occupancy time	No, wpq_occupancy/wpq_inserts is not inversely proportional to bandwidth
unc_m_pmm_wpq_inserts	per-socket: NVM write pending queue insert count	No, wpq_occupancy/wpq_inserts is not inversely proportional to bandwidth

monitored bandwidth as the input and decides what actions to take to adjust the bandwidth of the TGroup according to the regulation policy from system administrators.

3.3.1 Memory Regulation Mechanisms

MBA. To illustrate the effect of MBA, we use `fio` to generate different workloads under different MBA throttling values. Throttling value 100 means that there are no restrictions, while 10 represents the maximum MBA limit. The configuration of `fio` is the same as in §2.3.

The red lines in Figure 4 show the following phenomena. 1) MBA only supports limited throttling values, and not all throttling values are effective to workloads. This means that we cannot precisely control the bandwidth of threads with MBA. 2) MBA can restrict DRAM-intensive workloads better than NVM-intensive workloads. MBA is almost completely ineffective for NVM writes. Therefore, MBA alone is insufficient for controlling memory bandwidth. We must employ other techniques to restrict the NVM bandwidth.

CPU scheduling. An effective mechanism to control memory bandwidth is to reduce the number of cores allocated to an application [43]. We take a finer-grained approach by changing the CPU time (or CPU quota) of a thread with the help of the existing Linux CPU cgroup [6] controls. CPU quota in MT^2 defines an upper bound on CPU time allocated to the threads of a TGroup within a given period. TGroups with lower CPU quota take less CPU time, so it consumes less memory bandwidth. Since CPU quota leverages the CPU scheduler, it can provide a more fine-grained adjustment of memory bandwidth.

Effectiveness and comparison. CPU scheduling is a mechanism that supplements MBA. We repeat the same experiment with CPU scheduling as we do with MBA to compare these two mechanisms to decide how to cooperate better. Figure 4 shows the results. Take reading DRAM in figure 4(a) as an example (Read MBA and Read CPU two lines in the figure). When we don't enforce any limits on the workload (i.e., when the horizontal coordinate is 100), the throughput of 14 DRAM read `fio` workloads reachs 102GB/s on our platform. When the MBA throttling value keeps decreasing to 60, the throughput of the DRAM read does not change much. When this value decreases to 30, there is a significant drop in throughput. After we enforce the maximum limit via MBA, the throughput drops to about 28GB/s. For CPU scheduling, the throughput is proportional to the available CPU time. Thus, CPU scheduling can restrict memory bandwidth better than MBA.

Table 2: Execution time and max bandwidth of pagerank under different restrictions

	No limit	50% CPU	10% MBA
Execution Time(s)	56.459	118.179	78.662
Max BW_{read}(GB/s)	3.61	1.89	1.45
Max BW_{write}(GB/s)	4.84	2.53	1.87

The previous experiment is only for the effectiveness of bandwidth restriction. Table 2 gives some data when we run a real-world application, pagerank. We run the same task in three different situations. When this task runs without any limits, it takes 56 seconds to complete, of which the maximum read and write bandwidth is 3.61GB/s and 4.84GB/s, respectively. When we only allow it to use 50% of the CPU time, the task consumes 118 seconds, with the read and write bandwidth dropping to 1.89GB/s and 2.53GB/s. It seems that bandwidth usage is indeed changed to half while spending almost twice the original time. After we apply the maximum limit with MBA (10% MBA), its peak bandwidth is lower than that in 50% CPU, but it performs faster. This is because CPU scheduling will reduce the amount of CPU time the program can use. In contrast, MBA slows down the memory access operations and does not affect other operations, such as computation operations. We can conclude that the MBA mechanism is more efficient than CPU scheduling in restricting memory bandwidth.

Table 3 summarizes the characteristics of these two memory restriction mechanisms. As MBA has limited throttling values, it can only provide discrete restrictions on memory bandwidth. CPU scheduling can adjust memory bandwidth continuously, which could restrict the bandwidth finer. However, CPU scheduling is not as efficient as MBA since it is not friendly to the overall performance. At last, these mechanisms have different favor in memory access types. According

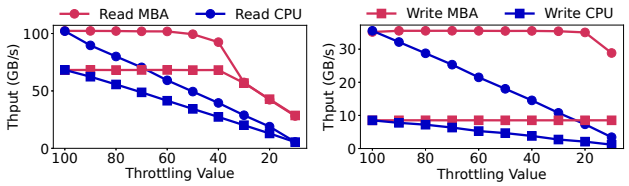

Figure 4: The effect of MBA and CPU scheduling on limiting DRAM (a) and NVM (b) bandwidth. Compared with MBA, CPU scheduling is more effective on restricting NVM bandwidth.

Table 3: Comparison of different memory restriction mechanisms

Mechanism	Granularity	Efficiency	Favor
MBA	Discrete	High	DRAM
CPU scheduling	Continuous	Low	Both

to Figure 4, MBA is good at restricting DRAM bandwidth, while CPU scheduling can cope with both DRAM and NVM bandwidth because it treats DRAM and NVM equally.

3.3.2 Dynamic Bandwidth Throttling

To ensure a relatively stable bandwidth for TGroups, we adopt an algorithm called dynamic bandwidth throttling that combines all mechanisms above. The algorithm first identifies noisy neighbors according to the information provided by the monitor and then takes actions to restrict the noisy neighbors' memory bandwidth.

Identifying the noisy neighbors. The algorithm will identify the noisy neighbors according to the enabled strategies. In the *prevention* strategy, the algorithm treats all TGroups that exceed their administrator-configured bandwidth limits as noisy neighbors. In the *remedy* strategy, the algorithm first checks whether there is severe memory interference on the platform according to the memory interference information provided by the monitor. If severe memory interference presents, the algorithm identifies noisy neighbors by each TGroup's current bandwidth use. According to the observations in the previous analysis (§2.3), a small amount of NVM writes can lead to severe bandwidth interference and NVM accesses affect others more severely than DRAM accesses. Thus, TGroups with the most NVM writes are more likely to become the noisy neighbors, followed by TGroups with most NVM reads, and finally the TGroups with more DRAM accesses. The algorithm picks the TGroup that is the most likely to be a noisy neighbor in the above order.

Regulating the memory bandwidth. The algorithm then chooses the memory regulation mechanism according to the types of memory bandwidth to restrict. To restrict NVM access bandwidth, the algorithm takes the CPU scheduling mechanism since MBA is almost ineffective for NVM. To restrict only DRAM access bandwidth, the algorithm chooses to decreases the MBA value of the target TGroups. If the MBA is already set to the lowest value, the algorithm uses CPU scheduling for further restriction.

Relaxing the memory regulation. Once the memory interference disappears, the algorithm attempts to relax the enforced bandwidth restrictions. The procedure is opposite to the way we add and enforce the restriction.

After the regulator finishes a single step of the algorithm (i.e., identifying then regulating/relaxing), it continues to wait for the next period in which another step will be taken according to the new information provided by the monitor. The step-by-step approach reduces the uncertainty of platform memory bandwidth changes and prevents unnecessary performance jitters for applications.

4 Implementation

As control groups (cgroups) [6] is an existing Linux kernel feature that manages resource usage of a collection of threads, we modify the Linux kernel 5.3.11 to add TGroup as a subsystem of cgroups. MT^2 is implemented as a kernel module that cooperates closely with the TGroup subsystem.

Cgroup interface. Cgroups exposes its interfaces via files in a pseudo-filesystem called cgroupfs. MT^2 follows the same approach as other cgroups subsystems. Specifically, an administrator first mounts the subsystem and creates a new directory in the subsystem mount point (i.e., creating a new TGroup). Then the administrator writes the pid of the process to the *cgroup.procs* file (i.e., adding the process to the TGroup). Three more files in this directory can be read/written to manage the TGroup:

1. The *priority* file is used to get and set the priority of a TGroup. Two priorities are currently supported. TGroups with *high* priority will not be restricted by the regulator, while the *low*-priority TGroups will be limited when interferences occur in the system.
2. The *bandwidth* file is read-only and returns the bandwidth of a TGroup for the last second.
3. The *limit* file is used to get and set the absolute bandwidth limit of a TGroup. Four comma-separated numbers can be written to this file as upper bandwidth limits of four types of memory accesses. When any one of the limits is exceeded, the TGroup will be throttled. A zero value indicates no limit, and the values take effect immediately.

When the write bandwidth cannot be separated accurately, only the first interface is valid. When the measured bandwidth is accurate, the last two can be used for prevention (§ 2.1). In this case, MT^2 allows the administrator to set four caps for four types of bandwidth for each TGroup. Once the real-time bandwidth used by one TGroup exceeds its limit, MT^2 enforces restrictions on that group, ensuring that each group does not use more bandwidth than the preset cap.

Thread creation. All the child processes are put in the same TGroup as their parent when created unless the administrator puts them manually into another TGroup. To achieve this, we also add a hook to the process/thread creation routine, which is the fork routine in the Linux kernel.

MBA. The MBA hardware supports ten MBA throttling values. However, there are only eight CLOS available on our platform. To support as many TGroups as possible, we do not assign a dedicated CLOS for each TGroup. Instead, we assign eight MBA throttling values to eight CLOS, respectively. We omit MBA throttling values 70 and 80 because the effect of the MBA throttling values 70, 80, and 90 are very similar across all workloads in Figure 4. As a result, each CLOS presents a different MBA throttling value. To restrict the bandwidth of a TGroup to an MBA throttling value, we assign all threads of the TGroup to the corresponding CLOS. Thus, by changing the CLOS of a TGroup, we can change its MBA throttling value. Since the MBA limits the memory bandwidth by adjusting the request rate between the physical core and the shared LLC, TGroups with the same CLOS get the same request rate without interference.

Context switches. To set up the MT^2 context for each thread, including setting the PMU related context, writing the MSR registers related to MBA and setting CPU quota, we add a hook to the scheduler. Each time a context switch happens, we set up the corresponding MT^2 context for the new thread that is going to run on this CPU core.

PMU. PMU is used to count read instructions that miss all caches and access the NVM and DRAM respectively. Using these data we are able to accurately calculate the DRAM and NVM read bandwidth. The latency of both types of read operations is also obtained through the PMU.

PEBS. We set the PEBS sample frequency to 10,007; thus, PEBS will record one linear address for every 10,007 events. As later evaluated in §6.2.2, this sample frequency is large enough to avoid noticeable overhead.

During context switches or PEBS interrupts occur, MT^2 reads all the samples in the PEBS buffer and filters out addresses in the kernel and the user stacks to mitigate the interference of irrelevant accesses and the CPU cache. MT^2 then translates the addresses to physical addresses and counts the number of NVM accesses and DRAM accesses, respectively. Finally, MT^2 stores the numbers in per-thread data structures, which will be used to estimate NVM bandwidth usage in the untrusted environment.

Listing 2: Kernel thread main loop

```
def kthread_main():
  start = current_time()
  interference = detect_interference()
  for group in TGroups:
    group.aggregate_bandwidths()
    group.adjust_bandwidths(interference)
  sleep(INTERVAL - (current_time() - start))
```

The dedicated kernel thread. A kernel thread is created at the initialization phase of MT^2 kernel module to periodically detect the interference, track the bandwidth and take actions generated by the dynamic bandwidth throttling algorithm. When interference is detected, the kernel thread calculates all types of bandwidth of all TGroups via information from MBM, VFS, and PMU. It then invokes the dynamic bandwidth throttling algorithm to adjust the bandwidth. The kernel thread runs at a configurable frequency (INTERVAL in Listing 2), which is once per 100ms in our implementation.

5 Other Use Cases

In addition to being used to prevent severe bandwidth interference caused by the noisy neighbors, MT^2 can also be used in more scenarios, such as memory bandwidth allocation, and cloud SLO guarantee.

5.1 Memory Bandwidth Allocation

For prevention, choosing and setting the maximum bandwidth for each application is a practical problem. A more reasonable solution in practice is the bandwidth guarantee, which assigns a minimum guarantee bandwidth to each task. As long as there is such a guarantee, a task will be able to use more bandwidth than this minimum guarantee when a task needs to use bandwidth, regardless of how much bandwidth other tasks are using at the same time.

Bandwidth allocation is essentially the same as bandwidth limiting since bandwidth resources are finite. The only method to reserve a minimum bandwidth for a program is to ensure that no other programs can consume excessive bandwidth resources. However, since the bandwidth resources in a hybrid system are not fixed, it is very difficult to give such a guarantee. We build an empirical model of four kinds of bandwidths on our platform to help us solve this problem. The input is the four bandwidths without interference, while the output is the actual bandwidths running simultaneously.

In this use case, each group of programs needs to predeclare their demand for each kind of bandwidth. Then we add up the demanded bandwidths of all the programs and pass to the empirical model to calculate the intensity of bandwidth competition. If the bandwidth competition is sufficiently intense, the minimum bandwidth for these programs cannot be guaranteed at the same time, and MT^2 will report an incident. If the bandwidth competition is low, we look for a point in the model where the system's bandwidth resources can be more fully utilized without excessive bandwidth competition (in our implementation, 90% of the desired value is considered to be no excessive bandwidth competition). Then MT^2 allocates the extra bandwidth resources proportionally to all programs, in such a way that each group of programs is guaranteed to use more than 90% of its own declared bandwidth.

5.2 Cloud SLO Guarantee

Service Level Objective (SLO) assurance is important for cloud users [18, 51]. For example, for users deploying KV-store applications, which primarily use memory bandwidth resources, the latency and the throughput of GET and PUT requests are what they value most. However, the request pattern of latency-critical (LC) tasks may not be fixed. There may not be any requests for a while, but at the next moment, the

requests become very intensive.

Cloud service providers want to make their devices as highly utilized as possible. When KV-store requests are not frequent, we can run some other best-effort (BE) background tasks simultaneously to make full use of the physical machine's bandwidth resources. When the foreground requests are dense, we can dynamically reduce the background tasks' bandwidth to ensure the SLO of the foreground tasks.

For trusted environment, we can use the bandwidth limiting (prevention) to prevent the BE applications from becoming noisy neighbors. We can divide the tasks into two TGroups: a high-priority TGroup for tasks that require a guaranteed SLO and a low-priority TGroup for other tasks. MT^2 first gives the foreground tasks a relatively smaller bandwidth guarantee. When the foreground tasks are about to run out of the allocated bandwidth, MT^2 allocates more bandwidth for them while reclaiming the bandwidth resources owned by the background BE tasks.

For untrusted environment, the two types of write bandwidth cannot be precisely separated. We can assign high priority to LC applications and low priority to the BE applications to mitigate the interference when the BE applications overuse memory bandwidth. This case is then transformed into remedy in noisy neighbor suppression.

6 Evaluation

In this section, we comprehensively evaluate MT^2 from multiple dimensions, including effectiveness for all use cases,the performance overhead, and the accuracy when the environment is trusted.

Experiment setup. Experiments are conducted on a server with two 28-core Intel® Xeon® Gold 6238R CPUs with hyper-threading disabled. The server has two NUMA nodes, and each is equipped with 6*32GB DDR4 DRAM and 6*128GB Optane™ PM configured in interleaved app-direct mode. All experiments are conducted on a single NUMA node.

6.1 Effectiveness

In this part, we evaluate the effectiveness of our three use cases: noisy neighbor suppression, memory bandwidth allocation and cloud SLO guarantee.

6.1.1 Noisy Neighbor Suppression

Effect of restrictions on noisy neighbors. We first re-conduct the experiment in §2 (results are shown in Figure 1) with and without MT^2 respectively to show the effect of MT^2 in micro-benchmarks. In this experiment, we run two `fio` simultaneously, one is marked as the noisy neighbor and the other as the victim. The throughputs when the victim runs alone are used as the baselines.

The results are shown in Figure 5. Generally, the columns with MT^2 have a much lighter color than the columns without MT^2, which indicates that MT^2 can effectively reduce noisy neighbors' interference by restricting their bandwidth

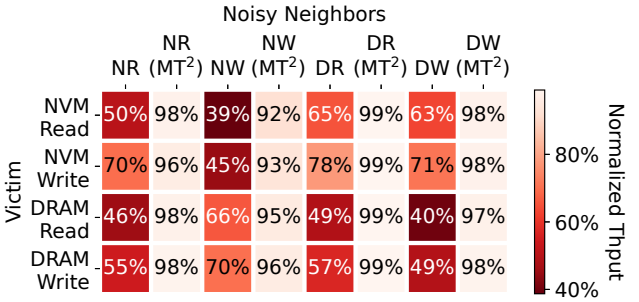

Figure 5: The normalized throughput of `fio` with/without MT^2. Noisy neighbors decrease the victim's throughput heavily while MT^2 can benefit the victim by restricting noisy neighbors' bandwidth.

usage. NVM Read noisy neighbor is abbreviated to NR in the following figures, and the others are similar. Take the NVM-Read workload as an example, four kinds of noisy neighbors decrease `fio`'s throughput to 50%, 39%, 65%, and 44% of the baseline. By restricting noisy neighbors' bandwidth with MT^2, `fio`'s throughput recovers to 98%, 92%, 99%, and 98% of the baseline. The other workloads present similar phenomena. The victim can run with a nearly maximal throughput with the help of MT^2.

Applications We evaluate three real-world applications, Hadoop [1], Graphchi [39] and Pmem-RocksDB [10], to check the effectiveness of MT^2. The computing tasks of these applications are conducted on DRAM, while the data is stored on NVM. Consequently, these applications will access both NVM and DRAM at the same time. The number of the `fio` noisy neighbors are the same as the configuration in §2.

On Hadoop 2.10.0 and Graphchi, we run a page-rank job on Twitter [11] social graph with 81,306 nodes and 1,768,149 edges. The iteration count of the page-rank is set to 3. Figure 6 gives the results. We treat the execution time when the victim application runs alone as the baseline. For both applications, MT^2 can mitigate the victim's performance slowdown well by restricting the noisy neighbors' bandwidth.

YCSB [12] is used to evaluate the throughput of RocksDB [8]. Before running the benchmark, we load 500,000 records, each with the size of 1KB, into RocksDB. Besides `fio`, we also use the aforementioned Graphchi with eight long jobs as the noisy neighbors (denoted by Graph in the figure). Figure 7 shows the results. Generally, the throughput of RocksDB with noisy neighbors is 61% to 77% of that without any noisy neighbors. When the noisy neighbors are restricted, RocksDB's throughput rises to about 94% to 100% of the original throughput. This indicates that MT^2 effectively reduces or even eliminates the impact of noisy neighbors to improve the high-priority application's bandwidth.

Response of limit update. A fast and accurate bandwidth limiting is the foundation of prevention. Figure 8 shows the applications' response to the update of the bandwidth limit in MT^2 at runtime. In this experiment, we run a six-job NVM Write `fio` and a fourteen-job NVM Read `fio` simultaneously.

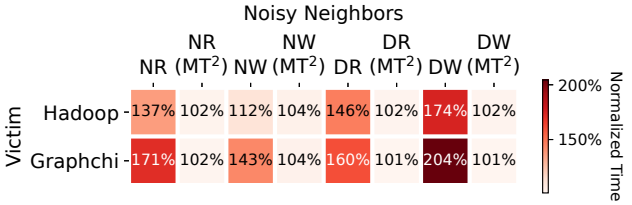

Figure 6: The normalized execution time of Hadoop and Graphchi when running page rank on Twitter social graph. Noisy neighbors slow down the execution and MT^2 mitigates the impact by restricting noisy neighbors' NVM bandwidth.

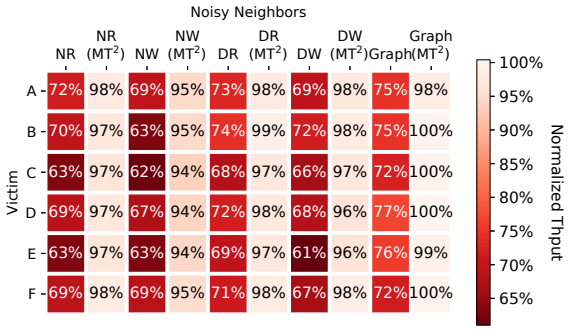

Figure 7: The throughput of YCSB's different workloads on RocksDB when running with/without MT^2 against eight noisy neighbors. Noisy neighbors decrease the throughput and MT^2 mitigates the impact by restricting noisy neighbors' bandwidth.

The concepts of victims and noisy neighbors are relative. We assume the former as the neighbor and the other as the victim. At first, the victim and neighbor run together without any restrictions. They can reach the throughput of 12.5GB/s and 7.2GB/s, respectively. After 5 seconds, we set the noisy neighbor's NVM write bandwidth limit to 5GB/s. It takes no more than one second that the noisy neighbor's throughput drops to 5GB/s, and the victim's throughput rises to 20GB/s due to less bandwidth interference. Similar results appear when noisy neighbors' NVM bandwidth limit is changed to 1GB/s after 15 seconds, 3GB/s after 25 seconds and unlimited after 35 seconds. The evaluation result shows that MT^2 can adjust a TGroup's throughput accurately and timely.

This also illustrates that it is not just applications that use plenty of bandwidth that can become noisy neighbors. An application that uses relatively small amounts of NVM bandwidth can have a significant impact on other applications. So the ability to distinguish between different types of bandwidth is critical in the hybrid NVM/DRAM platforms.

6.1.2 Memory Bandwidth Allocation

We run four `fio` tasks with different memory access patterns individually and record their throughputs. Then we run all four tasks simultaneously without MT^2 and with different guarantees with MT^2. As shown in Table 4, DRAM Write and NVM Read suffer the most severe bandwidth degradation when run together. DRAM Write has a whopping 66% drop in throughput (from 7.4GB/s to 2.5GB/s). We then assign

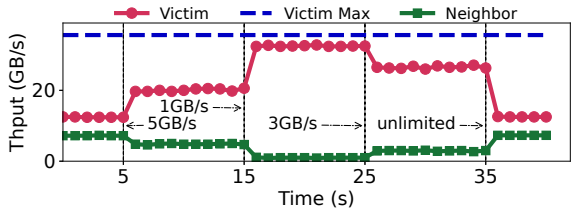

Figure 8: Response of the victim's and noisy neighbors' throughput when NVM-write intensive noisy neighbors' NVM write bandwidth limit is updated at run time. The limit is changed at 5s, 15s, 25s and 35s. When a lower limit is put on noisy neighbors, its throughput decreases and the victim's throughput increases and vice versa. The adjustment takes no more than 1 second and is very accurate.

Table 4: The throughput of `fio` tasks under BW allocation

Thput(GB/s)	Alone	w/o MT^2	Config 1	Config 2
DRAM Read	100	69.8	28.8(20)	11.5(10)
DRAM Write	7.4	2.5	5.3(5)	4.2(4)
NVM Read	7.2	3.4	4.2(4)	5.3(5)
NVM Write	5.0	3.8	4.5(4)	3.3(3)

different bandwidth guarantees (as indicated by the numbers in parentheses in the table) to these tasks, which are satisfied under the regulation of MT^2.

6.1.3 Cloud SLO Guarantee

We conduct three experiments to verify the effectiveness of the SLO guarantee. The first is a micro-benchmark that shows a breakdown of DRAM/NVM read/write bandwidth changes of both foreground tasks and background tasks. The second is a macro-benchmark which evaluates the 95th percentile latency of several LC tasks when running simultaneously with Graphchi [39] as the BE task. These two correspond to the first method in § 5.2 (similar to prevention). For the other method (like remedy), the third experiment is conducted. We run YCSB as the LC task with different types of memory accesses generated by `fio` to simulate different BE tasks.

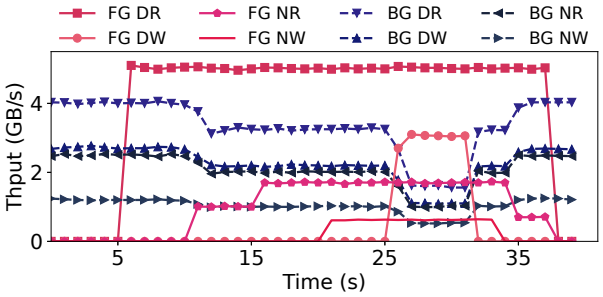

Figure 9: The throughput of the foreground and background (dashed lines) tasks. As foreground tasks use more and more bandwidth resources, if there are no sufficient bandwidth resources in the system, MT^2 will reduce the bandwidth for background tasks. When the foreground task reduce its memory usage, MT^2 will restore the bandwidth resources of the background tasks.

Table 5: The 95th percentile latency of several LC tasks

Workloads	w/o BEs	w/o MT^2	w/ MT^2
`img-dnn (ms)`	5.719	99.656	5.661
`masstree (ms)`	0.991	1.956	1.056
`YCSB-A-read (us)`	39	81	42
`YCSB-A-update (us)`	57	103	59

Breakdown of bandwidth changes. Figure 9 shows the result of dynamic bandwidth changes of foreground and background tasks. We use `fio` that read DRAM and NVM to act as the foreground workloads. For the background tasks, we choose all kinds of `fio` to show the impact to the hybrid bandwidth. The solid lines represent the bandwidth of foreground tasks, while the dashed lines represent background tasks. Before the beginning, MT^2 reserves 1GB/s bandwidth for each of the four types of memory accesses for the foreground tasks and allocate all the remaining bandwidth to the background tasks. After excluding the bandwidth reserved for foreground tasks, MT^2 lookups the empirical model and selects an appropriate bandwidth cap for background tasks to ensure that foreground tasks will not be affected until they use more bandwidth than reserved.

At first, the background tasks normally run with 4GB/s DRAM read, 2.7GB/s DRAM write, 2.5GB/s NVM read, and 1.2GB/s NVM write bandwidth consumption. After 5 seconds, a foreground task starts to read DRAM and takes 5GB/s DRAM read bandwidth. MT^2 increases the DRAM read bandwidth reservation for the foreground tasks to 6GB/s. As the existing four kinds of bandwidths do not exceed the limitation, MT^2 does not restrict the background tasks. Then the foreground tasks start to read NVM after 10 seconds. MT^2 finds that the foreground tasks occupy 1GB/s NVM read bandwidth, which exceeds 90% of the reservation, and assumes they may need extra NVM read bandwidth. So MT^2 lowers the NVM read bandwidth cap for the background tasks by 1GB/s, and then the NVM read bandwidth of background tasks exceeds the limit. As a result, MT^2 decides to tighten the restriction on background tasks. After 16 seconds, foreground tasks read NVM at 1.7GB/s, which is less than 90% of 2GB/s. Hence no additional NVM read bandwidth needs to be added to meet the SLO guarantee, i.e., MT^2 will not put more restrictions on background tasks.

The following bandwidth changes are all caused by the same reasons. With more bandwidth being taken by foreground tasks, the background tasks can use less bandwidth, and MT^2 puts more restrictions on them to ensure the foreground tasks' performance. After 32 seconds, the foreground tasks start to sleep one by one. Finally, all foreground tasks sleep, and background tasks occupy their original bandwidth.

Impact on latency. YCSB on RocksDB and two workloads from TailBench [36] (img-dnn [66] and masstree [44]) are selected as the latency-critical (LC) applications. First, we measure the 95th percentile latency of these LC tasks when running alone without any interference. Then we run the LC tasks together with 25 Graphchi (as the BE tasks) and measure their latency. We then group the LC tasks into one high-priority TGroup and the Graphchi into another (the BE TGroup) and repeat the same experiments. The results are shown in Table 5. Since YCSB's results are similar for all workloads, only the results for workload A are given.

Table 6: Tail latency of the LC task and throughput of the BEs

Workloads	alone	w/o MT^2	w/ MT^2
`95th YCSB-A-read (us)`	39	61	41
`95th YCSB-A-update (us)`	58	86	64
`99.9th YCSB-A-read (us)`	69	110	76
`99.9th YCSB-A-update (us)`	419	545	449
`FIO(DR) (GB/s)`	17.3	14.9	16.7
`FIO(DW) (GB/s)`	10.8	10.4	10.8
`FIO(NR) (GB/s)`	13.2	4.8	8.8
`FIO(NW) (GB/s)`	10.3	7.8	1.2

Without MT^2, the 95th percentile latency of `img-dnn` increases to 17.4x. MT^2 can restore all LC tasks' latency almost to the level when there is no interference at all. At the same time, the bandwidth of the BE tasks is limited to about 25% of the original. The performance of the BE tasks can be rapidly restored after the LC tasks are completed.

MT^2 in an untrusted environment. For this case, we use the hardware method (PEBS) to separate the two types of write bandwidth. We run YCSB as the LC application along with four fio tasks as the BE tasks. The four BE tasks perform read or write operations on NVM or DRAM respectively. MT^2 can optimize the latency of the LC application and improve the throughput of some BE applications by only restricting the bandwidth of the noisiest BE application as shown in Table 6. This thanks to the ability of distinguishing different types of memory bandwidth.

6.2 Performance Overhead

The performance overhead derives from three aspects: interference detection, bandwidth monitoring and restriction. The interference detection and monitoring overhead stems from setting and reading the PMU registers reading MBM data, and issuing write requests, while the restriction overhead comes from setting MT^2 context for threads, including MBA throttling value and CPU scheduling.

6.2.1 Trusted Environment

We measure the throughput or the execution time of some aforementioned test programs (`fio`, `graphchi`, `hadoop`, and `RocksDB`) with/without MT^2 to evaluate the performance overhead. As shown in Table 7, all overheads are less than 0.01%. The slight performance improvement in `graphchi` and `hadoop` is caused by noise. The overhead is negligible because most operations in MT^2 are performed by the dedicated kernel thread. For the interference detection, no overhead or additional bandwidth contention is introduced as only 400KB

Table 7: The performance overhead of MT^2

Thput/Time	w/o MT^2	w/ MT^2	Overhead
`fio`	31505 MB/s	31507 MB/s	< 0.01%
`graphchi`	321.64 s	321.55 s	< 0.01%
`hadoop`	54.93 s	54.93 s	< 0.01%
`RocksDB`	37770 ops/s	37767 ops/s	< 0.01%

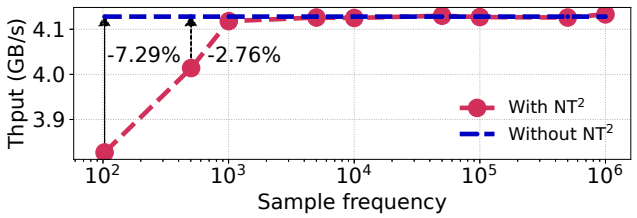

Figure 10: The throughput of `fio` under different sample frequencies. Frequencies no less than 10^3 introduce nearly no overhead. The Y-axis starts at 3.8GB/s to show the difference clearly.

data is written by the dedicated kernel thread for each period in our implementation, Others access the MSRs to use the hardware, which introduces little performance overhead.

We also run two applications on the same core to measure the overhead introduced in context switches. MT^2 increases an average of 900 cycles (less than 1 microsecond) in each context switch, which is insignificant compared to the millisecond-level scheduling period.

6.2.2 PEBS in the Untrusted Environment

In an untrusted environment, PEBS sampling is one of the main sources of performance overhead and is closely related to the sample frequency. We use `fio` to test the throughput under different sample frequencies and present the result in Figure 10. The overhead is negligible when the sample frequency is no less than 10^3.

6.3 Efficiency

We then split the regulation mechanisms to show the necessity of the two-stage algorithm design. First we run YCSB (as the victims) along with graphchi (as the noisy neighbors), and try using different techniques separately to restore the throughput of YCSB to 80% of the throughput it runs alone. We can not achieve the desired goal (restoring the throughput of YCSB to 80% of the initial) with MBA only. As shown in table 8, when we throttle the memory bandwidth only via CPU quota to restore the throughput of YCSB to 80%, the execution time of graphchi is 10m40s. The corresponding time is 9m56s when MT^2 is used. This indicates that MT^2 can control the bandwidth efficiently, consistent with our analysis in § 3.3.1.

Table 8: The efficiency of MT^2

Time	MT^2	CPU Scheduling
`graphchi`	9m56s	10m40s

Table 9: The deviation (in %) of monitored bandwidth in MT^2

Bandwidth(GB/s)	DR	DW	NR	NW
MT^2	10.51	4.19	3.84	2.79
PCM	10.69	4.22	3.89	2.81
Deviation	1.68%	0.71%	0.13%	0.71%

6.4 Accuracy

When the environment is trusted, applications faithfully report their NVM write bandwidth using the interface we provide. Although other bandwidths are obtained using reliable techniques, we still need to verify whether the results are accurate. PCM [58] is a software that can monitor different types of bandwidth of the whole system. When there is only one memory-intensive program in the system, the system-wide bandwidth reported by PCM is almost equal to the only program's bandwidth. So we run four `fio` simultaneously to generate different kinds of workloads and compare the bandwidths monitored by MT^2 with those of PCM since Intel also uses PCM's bandwidth as the baseline [32]. The results are shown in Table 9, where the results reported by MT^2 are very close to the ground-truth bandwidths.

7 Discussions

7.1 Limitations and Possible Mitigations

MT^2 brings the hybrid memory bandwidth regulation with several limitations. MT^2 relies on hardware mechanisms such as MBA, MBM, and PMU, which may conflict with applications that also depend on these techniques. The problem stems from the limited hardware resources. For example, there are only 79 RMIDs on our platform. It is possible to mitigate these limitations via virtualization or by more powerful hardware in the future.

Besides, the granularity of our empirical model is coarse, which may result in some bandwidth waste. Machine learning may be used to build more accurate models with a finer granularity in the future.

Currently, MT^2 is only able to accurately track the bandwidth of applications accessing the NVM via the file system and NVM programming libraries like PMDK. On trusted environments, MT^2 relies on applications to report their NVM write bandwidth honestly. Since many applications (such as PmemKV [5], Pmem-RocksDB [10], and Pangolin [71]) choose to use NVM programming libraries to manage the NVM, this can be easily achieved by slightly modifying the libraries. For untrusted environments, MT^2 can only monitor the write bandwidth roughly because of the hardware limitation, which can be addressed correctly by an update to the hardware mechanism.

7.2 NUMA

Currently, MT^2 does not support cross-NUMA bandwidth monitoring/regulation. MT^2 assumes that applications are bound to the same NUMA node where its NVM resides so

that it can monitor and regulate without cross-NUMA NVM accesses. The binding can be done manually (by the admins) or by the system's scheduler (e.g., in cloud environments). This assumption is reasonable and commonly stands in practice since applications and FS tend to access the local NUMA node to avoid degraded cross-NUMA accesses. For multi-socket machines, MT^2 can separate the monitoring and restriction policies for different sockets so that MT^2 will not regulate the TGroup in socket 1 when the bandwidth contention level is high in socket 0.

7.3 Future work

In some scenarios, cross-NUMA accesses are inevitable. A NUMA-and-NVM-aware scheduler can mitigate the memory bandwidth interference via more advanced scheduling policies, e.g., isolating DRAM-only applications and the NVM-intensive applications to different NUMA nodes. We leave the memory throttling in such scenarios as future work.

MT^2 prefers to limit TGroups with massive write NVM writes. However, these TGroups are not always the culprits of memory bandwidth contention. Accurately identifying the noisy neighbors remains a challenge and might require more hardware assistance. We leave the exploration of accurate bandwidth monitoring and regulation with hardware modifications as future work.

8 Related Work

DRAM bandwidth monitoring and regulation. MemGuard [67–69] is a DRAM bandwidth reservation system designed for real-time multi-core systems. It provides guaranteed and best-effort DRAM bandwidth for different applications. MemGuard monitors the DRAM traffic by accounting for the cache misses and suspends a task when it has exhausted its budgets in a given period.

Although both MemGuard and MT^2 aim to throttle memory bandwidth to avoid interference, MT^2 differs from MemGuard in several aspects. First, MemGuard is designed for DRAM in real-time systems, but MT^2 is proposed for hybrid NVM/-DRAM platforms. Second, MemGuard throttles DRAM bandwidth via a software budget-based throttling mechanism. MT^2 leverages both hardware and software mechanisms and proposes a dynamic bandwidth throttling algorithm to better regulate bandwidth for various applications.

LIKWID [59], Larysch [40], and Merlin [57] estimate memory bandwidth with L3 cache miss information collected from hardware performance counters. LibDistGen [17] estimates the memory bandwidth of applications based on stack reuse histograms. Mmbwmon [16] estimates the memory bandwidth consumption of applications by running benchmarks on other CPU cores of the system simultaneously. These techniques are proposed for DRAM and cannot be simply adopted to a system with both DRAM and NVM.

EMBA [61] models the relationship between performance and LLC occupancy and memory bandwidth and then proposes an algorithm with Intel MBA to restrict the memory bandwidth to improve the overall system performance in data centers. However, EMBA cannot control the memory bandwidth of a group of threads, and it cannot be used on hybrid NVM/DRAM platforms.

HyPart [50] consists of thread packing, clock modulation and Intel's MBA. MT^2 utilizes the CPU scheduler, thus providing finer-grained and precise control. Caladan [24] is a CPU scheduler that supports task monitoring and scheduling at the microsecond level. Some other studies [15, 23, 47, 72] also reduce resource contention with a modified scheduler.

None of these works can be used directly on hybrid memory NVM/DRAM platforms because the interference model is completely different from the DRAM-only platforms. They can only be used on hybrid platforms if they can separate the DRAM and NVM traffic at thread granularity as MT^2 does.

Hybrid NVM/DRAM bandwidth interference regulation. FairHym [27] will limit the frequencies of cores that perform NVM writes when the NVM write bandwidth exceeds a threshold to improve the inter-process fairness. It only concerns the bandwidth interference between NVM writes and DRAM accesses. It requires an impractical setup (installing DRAM and NVM on different NUMA nodes) to estimate the number of NVM writes. MT^2 has more flexible monitoring and allocation method that takes all types of bandwidth interference into account and can be used to meet the need of different user scenarios. Dicio [49] can control the bandwidth interference in a single LC and a single BE job situation. It blames and throttles the only BE job. In real-world scenarios, it cannot figure out which one to blame. In comparison, MT^2 targets a more practical setup (each NUMA node has both DRAM and NVM) and a more common scenario where multiple applications can run together.

9 Conclusions

This paper presents MT^2, the first comprehensive system to regulate memory bandwidth on the hybrid NVM/DRAM platforms. MT^2 first detects the bandwidth interference, monitors four types of memory bandwidth through various mechanisms and adjusts the bandwidth with a dynamic bandwidth throttling algorithm. Evaluation shows that MT^2 can effectively regulate the bandwidth among applications with nearly zero performance overhead and can be used in multiple use cases.

Acknowledgments

We sincerely thank our shepherd Sanidhya Kashyap and the anonymous reviewers from ATC '20, FAST '21, ATC '21, SoCC '21, and FAST '22 for constructive comments and insightful suggestions. This work is supported in part by the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), the National Natural Science Foundation of China (No. 61925206), and Huawei. Mingkai Dong (mingkaidong@sjtu.edu.cn) is the corresponding author.

References

[1] Apache hadoop. https://hadoop.apache.org/.

[2] Flexible i/o tester. https://fio.readthedocs.io/en/latest/index.html.

[3] Intel® optanetm dc persistent memory quick start guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[4] Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[5] Key/value datastore for persistent memory. https://github.com/pmem/pmemkv.

[6] Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[7] Memcached. https://memcached.org/.

[8] A persistent key-value store for fast storage environments. https://rocksdb.org.

[9] Persistent memory programming. https://pmem.io/pmdk/.

[10] Rocksdb on persistent memory. https://github.com/pmem/pmem-rocksdb.

[11] Twitter socail graph. http://snap.stanford.edu/data/ego-Twitter.html.

[12] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[13] Google Cloud. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, 2019.

[14] SAP HANA. https://www.sap.com/products/hana.html, 2019.

[15] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, page 1, USA, 2011. USENIX Association.

[16] Jens Breitbart, Simon Pickartz, Stefan Lankes, Josef Weidendorfer, and Antonello Monti. Dynamic co-scheduling driven by main memory bandwidth utilization. *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 400–409, 2017.

[17] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. Automatic co-scheduling based on main memory bandwidth usage. In *JSSPP*, 2015.

[18] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[20] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, 2019. ACM.

[21] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 719–731, Berkeley, CA, USA, 2017. USENIX Association.

[22] Intel Xeon Processor Scalable Memory Family. Uncore performance monitoring reference manual. *Intel Corporation, July*, 2017.

[23] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, page 25–38, USA, 2007. IEEE Computer Society.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 281–297, 2020.

[25] Yiming Huai et al. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS bulletin*, 18(6):33–40, 2008.

[26] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency

in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, page 187–200, USA, 2018. USENIX Association.

[27] S. Imamura and E. Yoshida. Fairhym: Improving interprocess fairness on hybrid memory systems. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2020.

[28] Intel. Intel optane dc persistent memory readies for widespread deployment. https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment/, 2018.

[29] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3: System Programming Guide*, pages Vol. 3B 17–64, 2019.

[30] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3: System Programming Guide*, 2019.

[31] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3: System Programming Guide*, pages Vol. 3B 18–19, 2019.

[32] Intel. Intel resource director technology (intel rdt) on 2nd generation intel xeon scalable processors reference manual. *Intel Resource Director Technology Reference Manual*, 2019.

[33] Intel. Intel(R) Optane(TM) DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2019.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[35] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[37] Takayuki Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Des. Test*, 28(1):52–63, January 2011.

[38] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[40] Florian Larysch. Fine-grained estimation of memory bandwidth utilization. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, March13 2016.

[41] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[42] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 257–270, USA, 2017. USENIX Association.

[43] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In Deborah T. Marr and David H. Albonesi, editors, *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 450–462. ACM, 2015.

[44] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[45] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[46] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile

data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 153–166, New York, NY, USA, 2010. Association for Computing Machinery.

[48] Khang T Nguyen. Introduction to memory bandwidth monitoring in the intel(r) xeon(r) processor e5 v4 family. https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-monitoring, 2016.

[49] Jinyoung Oh and Youngjin Kwon. Persistent memory aware performance isolation with dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 97–105, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206, 2020.

[52] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[53] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 15:1–15:15, 2014.

[54] Yujie Ren, Changwoo Min, and Sudarsun Kannan. Crossfs: A cross-layered direct-access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020.

[55] Smith Ryan. Intel announces optane storage brand for 3d xpoint products. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products, 2015.

[56] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453(7191):80, 2008.

[57] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[58] Thomas Willhalm and Roman Dementiev. Intel(r) performance counter monitor - a better way to measure cpu utilization. https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html, 2012.

[59] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, page 207–216, USA, 2010. IEEE Computer Society.

[60] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[61] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[63] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[64] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 167–181, USA, 2015. USENIX Association.

[65] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 98–105, 2020.

[66] Xingdi (Eric) Yuan. A deep network handwriting classifier. https://github.com/xingdi-eric-yuan/multi-layer-convnet, 2014.

[67] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 299–308. IEEE Computer Society, 2012.

[68] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 55–64. IEEE Computer Society, 2013.

[69] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Trans. Computers*, 65(2):562–576, 2016.

[70] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 897–911, USA, 2019. USENIX Association.

[71] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 897–912, 2019.

[72] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 129–142, New York, NY, USA, 2010. Association for Computing Machinery.

[73] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 461–476, USA, 2018. USENIX Association.