Persistent Memory Aware Performance Isolation with Dicio

Jinyoung Oh
KAIST
jinyoungoh@kaist.ac.kr

Youngjin Kwon
KAIST
yjkwon@kaist.ac.kr

ABSTRACT

Cloud vendors increasingly support persistent memory (PM) in their cloud platform to lower the total cost of ownership (TCO). They run multiple instances on cloud nodes supporting large memory with DRAM and PM. While PM enables the low-cost in-memory computations, adding PM to a cloud raises new challenges of managing the performance interference between co-running applications. Since PM has its own unique performance characteristics, cloud vendors need to rethink the performance isolation problem beyond the conventional performance interference models on DRAM.

This work observes interference effects between DRAM and PM (i.e. PM-by-PM, PM-by-DRAM, and DRAM-by-PM interference). Based on our observations, this paper provides a new model to minimize the performance impact of a latency-critical (LC) application by a best-effort (BE) application when they heavily use both DRAM and PM. Especially, this work presents Dicio, which carefully considers the internal bandwidth bloating by PM's write amplification to control performance interference. To compare with the state-of-the-art that does not support PM, we modify a recent system, Caladan, to support PM and compare Dicio with it. This work significantly improves the tail latency of memcached-pmem (LC) and the throughput of a BE process.

ACM Reference Format:
Jinyoung Oh and Youngjin Kwon. 2021. Persistent Memory Aware Performance Isolation with Dicio. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3476886.3477517

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APSys '21, August 24–25, 2021, Hong Kong, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8698-2/21/08…$15.00
https://doi.org/10.1145/3476886.3477517

1 INTRODUCTION

Cloud providers incorporate persistent memory (PM) into their service nodes to lower the total cost of ownership (TCO) for accommodating large memory applications [6]. Cloud service providers place several tasks together on the same physical machine to improve resource utilization [7, 14, 26]. In such a multi-tenant environment, a common practice is to colocate high-priority latency-critical (LC) applications and low-priority best-effort (BE) applications while isolating performance impact on LC applications from BE applications. To isolate performance interference on shared resources, there have been several approaches to monitor and control the performance of BE applications [2, 5, 16–18, 21, 23–25].

Adding PM to the resource controlling domain incurs new challenges because simply applying the existing approaches to confine performance interference by PM has limitations due to the unique characteristics of PM. First, a commercialized PM, Intel Optane PM [12], often exhibits a significant mismatch between the amount of bandwidth application requests (request-level bandwidth) and what PM device consumes (media-level bandwidth). As shown in Figure 1, the access granularity of Intel Optane PM is 256B [29] while the size of a cache line is 64B. When an application writes only 64B, Intel Optane PM performs 256 B write, inflating the internal bandwidth consumption 4 times. Therefore, the actual media-level bandwidth varies by an application's access patterns and sizes. To our best knowledge, no existing system considers the amplified media-level bandwidth for isolating PM interference. Second, PM offers a much smaller BW than DRAM so that even a single thread can saturate the entire bandwidth of it, causing a significant latency impact on co-running processes. Furthermore, a workload can exhibit sudden bandwidth peaks in a short window, requiring finer-grained monitoring and controlling mechanisms [5]. Third, when a BE process use both DRAM and PM, performance impact by DRAM and PM is combined and, furthermore, each of them has a distinct interference effect, requiring a sophisticated monitoring method to identify their individual interference impact.

This paper proposes, Dicio, a performance interference management system for systems with both DRAM and PM. Dicio identifies different types of performance interference:

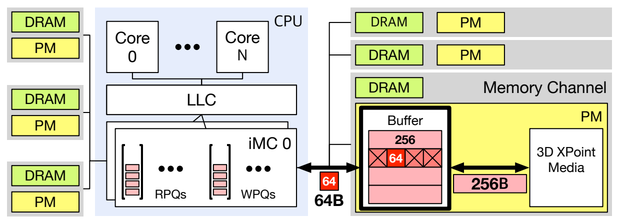


Figure 1: *Overview of PM-equipped system. Write-/read-amplification occurs when only part of 256B is written/read.*

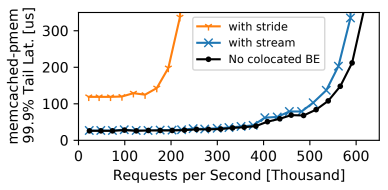


Figure 2: `stride` *causes QoS violation of* `memcached-pmem` *although both BE seems to consume the same request-level BW.*

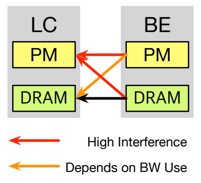


Figure 3: *Performance interference overview*

PM-by-PM, PM-by-DRAM, and DRAM-by-PM where PM-by-DRAM represents when an LC process using PM interfered by a BE process heavily uses DRAM. Dicio considers PM media-level BW instead of request-level BW to control the interference. To react to sudden change of workload, Dicio provides a novel approach to monitor PM performance at microseconds-scale to control a BE process. Dicio monitors the queuing delay in write pending queue (WPQ) for PM in memory controller which effectively detects PM latency impact by a BE process using both DRAM and PM. Based on the monitoring mechanism, Dicio controls the bandwidth of a BE process through adjusting the number of cores assigned to the BE process.

To evaluate Dicio, we augment the state-of-the-art system, Caladan [5] to manage PM BW and integrate Dicio to the Caladan's runtime. We run `memcached-pmem` as an LC process and a BE process to stress PM and DRAM bandwidth. In our evaluation, Dicio shows 8.17× lower 99.9%-tile tail-latency than Caladan in the PM-by-PM case and 58.8× lower in the PM-by-DRAM/PM case. Furthermore, Dicio enables BE jobs to consume up to 6GB/s of request-level bandwidth. With a policy based on request-level bandwidth instead of media-level bandwidth, such a high BW should not be allowed for BE job otherwise cause catastrophic performance interference.

2 BACKGROUND & RELATED WORKS

Colocating high-priority latency-critical (LC) jobs and low-priority best-effort(BE) jobs on a machine to improve resource utilization is a common practice in datacenters [7, 14, 26]. On such systems, Quality of Service (QoS) of LC jobs should be guaranteed while BE jobs get enough resources to achieve high throughput. Many researchers have proposed systems to solve it. They manage interference by isolating CPU (number of cores, LLC ways) [2, 5, 21–23, 25], memory (capacity, bandwidth) [2, 5, 21, 23–25], storage [2, 16, 25], and network [2, 17, 18, 21, 25].

Among many shared resources, memory bandwidth is one of critical resources that significantly harm performance when saturated [21]. To isolate DRAM bandwidth, existing systems [2, 5, 21, 23–25] monitors DRAM bandwidth consumption and throttles a BE job that consumes the bandwidth too much. Unfortunately, those approaches cannot be directly applied to isolate PM performance because of the unique characteristics of PM that we discuss in §3. In addition, to our knowledge, there is no work to isolate PM performance between an LC job and a BE job although isolating PM bandwidth is important because it is more prone to be saturated than DRAM and incurs significant degradation on both memory and storage subsystems.

Caladan [5] is a start-of-the-art approach to address performance interference. It focuses on mitigating performance interference by DRAM bandwidth at μs timescale while others manage interference in units of seconds. It detects DRAM BW saturation at μs timescale and mitigate it by reducing the number of cores that run a BE job. However, its policies and mechanisms could not be directly applied for PM because of the unique characteristics of PM. In this paper, we discuss the challenges to isolate performance of PM and how naive extension of prior works for PM fails to solve the problem, compared to our system.

3 MOTIVATION

This section discusses the performance interference problems of PM due to write amplification of PM and how DRAM and PM traffic interferes each other. Details of our experimental setup are described in § 5. As recommended by Intel [8], we followed the 2-2-2 topology to utilize all available DIMM slots. Under this configuration, as illustrated in Figure 1, there are two iMCs per CPU each supports three channels, and there are one DRAM and one PM per each memory channel. PMs are configured in App Direct Mode.

3.1 Access Pattern on PM Matters

Due to the discrepancy of access granularity of cache (64 B) and PM media (256 B), the *media-level bandwidth* consumption could be bigger than what amount of IO the application requests (*request-level bandwidth*). To identify the mismatch between the media-level and request-level bandwidth, we perform the following experiment. We set two different best-effort (BE) applications: storing 64B data sequentially (`stream`) and storing 64B data at each 256B-aligned address (`stride`). We inject nops between stores to regulate request-level bandwidth consumption; otherwise, they fully saturate

PM BW (discussed later). We observe the request-level bandwidth of `stream` and `stride` is 3.1GB/s. We run a latency-critical (LC) process, `memcached-pmem` with each BE process. Figure 2 shows 99.9%-tile latency of `memcached-pmem`. When `memcached-pmem` co-runs with `stream`, the tail latency is similar to the case of running it only. However, when it is colocated with `stride`, it shows much worse tail-latency. It is because `stride` consumes 4× more media-level bandwidth than `stream`, saturating the entire PM bandwidth. This mismatch between the request-level and the media-level poses a unique challenge to isolate the performance of PM-based applications. Previous work for performance isolation controls DRAM performance interference by monitoring and regulating the request-level bandwidth [5, 25]. However, relying only on it would lead to incorrect decisions because the actual media-level bandwidth consumption may differ from what we expect from the request-level bandwidth.

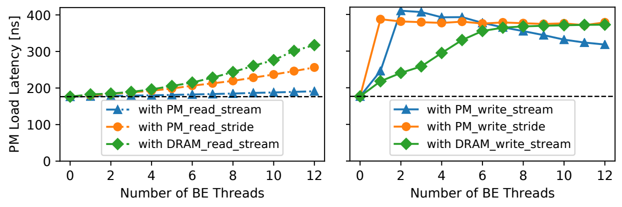
3.2 PM is More Susceptible to Interference

We extend the discussions of PM performance interference as shown in Figure 3. We focus on three performance interference cases: PM-by-PM, PM-by-DRAM, DRAM-by-PM where, for example, PM-by-DRAM represents interference on PM performance by heavy DRAM traffic from a colocated job. The interference between DRAM and DRAM is not a scope of our work because there exists much work to address the case [2, 5, 21, 23–25].

PM-by-PM/DRAM. We measure how a colocated BE process that heavily accesses either PM or DRAM affects the latency and bandwidth of PM. We see that a BE process that heavily use DRAM/PM would harm the performance of LC applications that utilizes PM such as key-value stores [3, 15, 20] and file systems [19, 28].

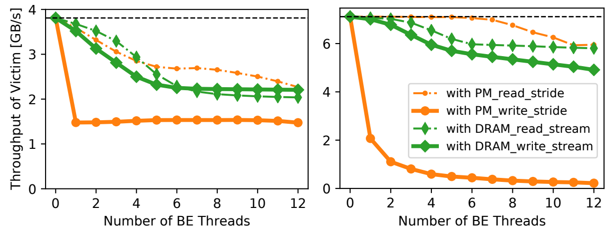
To generate performance interference, we co-run a microbenchmark that performs read or write in the `stream` or `stride` pattern as a BE process. Figure 4 represents PM load latency as increasing the number of BE threads accessing either PM or DRAM. We measure PM access latency with Intel Memory Latency Checker (MLC) [11]. When the BE process performs reads (Figure 4a), its impact is smaller than when the BE process performs write heavily (Figure 4b). Interestingly, in case of interference by a read-intensive BE, DRAM read (`DRAM_read_stream`) affects the PM latency more than PM read (`PM_read_stream/stride`). We suspect it is because DRAM read is 3× faster and consumes larger BW than PM read with the same number of threads, causing more memory bus contentions.

Performance interference by PM write (Figure 4b) is much severe than by PM read. Even only one thread of `PM_write_stride` or two threads of `PM_write_stream` saturates the entire PM media-level bandwidth (about 12.5GB/s), causing up



(a) *PM latency by DRAM/PM read* **(b)** *PM latency by DRAM/PM write*

Figure 4: *PM latency (lower is better) increases as more PM or DRAM traffic; especially when BE writes PM. We note that even one or two thread(s) can fully saturate PM bandwidth. The dotted line at 176ns represents the idle latency with no colocated BE.*



(a) *PM read throughput of victim* **(b)** *PM write throughput of victim*

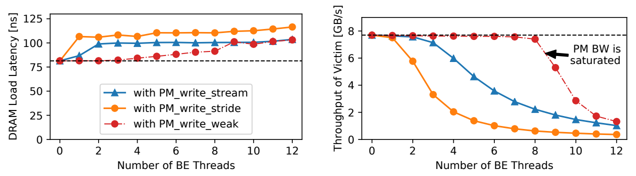
Figure 5: *PM throughput (higher is better) of the single-threaded victim is degraded by both DRAM and PM traffic. Even a thread of* `PM_write_stride` *significantly harms PM throughput.*

to 2.33× latency increase. Similar to DRAM read, `DRAM_write_stream` also affects PM latency as DRAM BW increases and reaches the peak when it consumes about 45GB/s of DRAM BW.

Figure 5 describes how PM read (Figure 5a) or write (Figure 5b) throughput of a single-threaded victim process affected by a colocated BE. The performance interference is most severe when the victim co-runs with `PM_write_stride`. It is because even only one thread of it fully saturates the write bandwidth of PM. It reminds us of the significance of the performance isolation on PM.

DRAM-by-PM. Figure 6 shows performance degradation of DRAM by a BE process writes on PM. Figure 6a shows how long DRAM latency takes according to the number of the BE process' threads. As the number of threads increases, the DRAM latency increases significantly starting from the point where PM write bandwidth is fully saturated (1 thread of `PM_write_stride` and 2 threads of `PM_write_stream`). With the same number of BE threads, `PM_write_stride` incurs more severe interference than `PM_write_stream`, in line with the other experiments. The single-thread DRAM write throughput (Figure 6b) is also degraded by PM write.

To clarify the interference by PM's bandwidth consumption, we run a BE process with the `stream` pattern that has nops between writes so that consume only 1.5GB/s of PM write media-level bandwidth per thread (`PM_write_weak`). It does not interfere with DRAM until it saturates the PM bandwidth. However, when it fully utilizes the PM bandwidth



(a) *DRAM latency by PM BE* **(b)** *DRAM write throughput by PM BE*

Figure 6: *DRAM performance is degraded significantly if PM media-level BW is saturated; otherwise, there is moderate interference.*

(with 8+ threads), the victim's throughput plummets (Figure 6b), indicating PM media-level bandwidth consumption of a BE job is an important factor to monitor and control even when the victim process only uses DRAM.

In summary, (1) Unlike DRAM, what we need to consider is PM media-level bandwidth rather than request-level bandwidth, (2) In PM, because a BE process degrades others significantly with only a few threads, fine-grained interference monitoring and control are required, and (3) Among various memory bandwidth traffic, PM write bandwidth is the most important and scarce resource that should be managed carefully so that it never be saturated.

4 DICIO

This section introduces Dicio, a PM-aware performance isolation system that considers media-level bandwidth of PM and performance interference from both DRAM and PM accesses. Dicio targets the common performance interference model that runs an LC process colocated with a BE process [14] and they consume both DRAM and PM resources. The goal of Dicio is to mitigate the tail latency impact of the LC process by the BE process while providing a good throughput of the BE process.

4.1 Bandwidth Monitoring Mechanism

As discussed, Dicio monitors the interference based on media-level bandwidth rather than request-level bandwidth. Dicio regulates the interference with fine-grained controls. This requires monitoring PM media-level bandwidth frequently. Currently, however, obtaining the information is challenging due to insufficient hardware supports.

Challenge. The only way to measure media bandwidth is using *per-DIMM* counter (*i.e.* `MediaReads`, `MediaWrites`) which tells how many 64B data have been read/written on PM media [13]. To derive the media bandwidth, one needs to read the counter twice, calculate the difference, and divide it by time between the two measurements.

However, it has a critical limitation to utilize it for performance isolation: reading the counter involves ACPI call to PM HW so takes too long and is even unpredictable. On our machine, it takes about 4.5*ms* when PM is idle; but it often takes 1.3× longer when PM is busy. Considering that many real-world applications show sudden resource consumption pattern at μs timescale [1, 5, 14], BW monitoring at millisecond timescale is not adequate to cope with the behavior, causing poor tail latency.

Our approach. To circumvent this challenge, Dicio uses the queuing delay of the write pending queue (WPQ) for PM in the memory controller to approximate media-level BW. We observe the WPQ queuing delay is highly correlated to PM access latency. To measure it, Dicio reads two hardware counters from performance monitoring unit (PMU) in the processor: 1) `UNC_M_PMM_WPQ_OCCUPANCY.ALL`, which counts the accumulated number of WPQ entries at each cycle and 2) `UNC_M_PMM_WPQ_INSERTS`, which counts how many entries have been inserted into WPQ [10]. And then Dicio derives the WPQ delay by calculating: `UNC_M_PMM_WPQ_OCCUPANCY.ALL / UNC_M_PMM_WPQ_INSERTS`. Along with WPQ queuing delay, it reads counters that indicate DRAM BW from PMU as well (`UNC_M_CAS_COUNT.ALL`). Reading PMU takes only a few μseconds, which is short enough to detect sudden PM BW peaks caused by workload change.

While Dicio uses the WPQ queuing delay as a major metric to start throttling a BE process, Dicio measures PM media-level and request-level BW from PM hardware counters as well. It is because the WPQ queuing delay is a *result* of contention and does not tell how much bandwidth is consumed.

Dicio runs as a privileged user-level process on a dedicated core and periodically reads PMU at every 10us (*Main thread*). Once the measurement completes, it reacts according to the policy discussed in §4.3. Dicio spawns another thread (*PM poll thread*) to monitor accurate media-level BW, which takes a long. It runs on the hyperthread sibling of the main thread, and tells the main thread via shared data structure. We explain the detailed use of them in §4.3.

4.2 Bandwidth Control Mechanism

Most mechanisms that prior works have used for DRAM bandwidth control do not work well for PM. We measure how existing DRAM throttle mechanisms are effective to regulate PM bandwidth use: Intel Memory Bandwidth Allocation (MBA) [23, 25] and halving the processor clock [24]. As shown in Figure 7, however, they do not work for throttling `PM_write_stride` which is the most radical PM BW antagonist that even a thread fully saturates PM BW (§3.2).

Instead, we opt-in a way to limit the number of memory operations by not scheduling it temporally. It is an extended form of thread packing that some prior works have used [5, 24]. We extend it by adding a feature to control the duty cycle of a BE job (*i.e.* a job runs on 0<x<1 core by carefully scheduling it in and out at microsecond timescale.) In this way, we can throttle the radical PM BW antagonist (Figure 7). To implement so, we need a fast scheduling mechanism, and

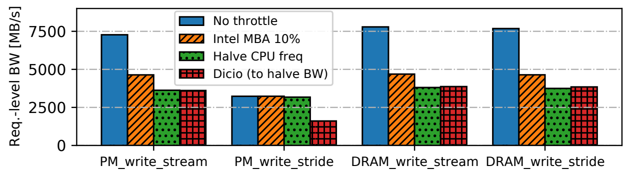


Figure 7: *Existing mechanisms cannot throttle* `PM_write_stride`.

recently, Fried *et al.* have proposed it [5]. Thus we adopt the scheduling mechanism in Dicio. In this regard, throttling a BE with Dicio means taking a core from it and relaxing means allowing one more core.

4.3 Interference Management Policy

To meet a QoS of an LC job, Dicio throttles a resource-hogging BE job when identifying a bandwidth contention. Meanwhile, to provide good throughput of the BE job, Dicio relaxes it when the contention is resolved and there is unused bandwidth to exploit. For each purpose, Dicio maintains two states: `Throttling` and `Relaxing` (and a special case).

`Throttling`. When *any of the following conditions are met*, Dicio takes a core from a BE job and puts it under `Throttling` state:

(1) PM media-level write BW $\geq T_{mediawrite}$
(2) PM media-level read BW $\geq T_{mediaread}$
(3) DRAM BW $\geq T_{DRAM}$
(4) WPQ queuing delay $\geq T_{WPQ}$
(5) Relaxed by prediction $\wedge$ Estimated media-level write BW $\geq 1.2 \times T_{mediawrite}$

where $T_{mediawrite}$, $T_{mediaread}$, T_{DRAM}, and T_{WPQ} are thresholds values.

The first and second conditions check whether actual PM media-level BW is saturated. As the measurement of the media-level BWs takes too long, we only evaluate those conditions only if the *PM poll thread* has read a new media-level BW. We set $T_{mediawrite}$ as 6GB/s and $T_{mediaread}$ as 10GB/s through our sensitivity study. We note that PM media-level read BW includes write BW because write is actually a read-modify-write operation in PM. Therefore, the second condition can address combined interference from PM read and write as well.

The third condition is to address combined interference from both PM and DRAM. We observe that the degree of interference to PM access by DRAM depends on current PM media-level BW. When PM media-level BW is consumed heavily, a small amount of DRAM traffic can affect an LC process; On the other hand, when PM media-level BW consumption is low, the same amount of DRAM traffic does not degrade the LC process. Therefore, Dicio applies dynamic thresholds for controlling DRAM consumption. When PM media-level BW consumption is low (*i.e.*, less than $0.4 \times T_{media}$), Dicio throttles a BE process to use DRAM BW up to 30 GB/s[1]. When PM media-level BW consumption is moderate (*i.e.*, less than $0.8 \times T_{media}$), Dicio throttles DRAM BW up to 15 GB/s. If PM media-level BW is consumed more, Dicio throttles DRAM BW up to 5GB/s. Possibly, applying finer-grained thresholds would improve the throughput of a BE process. We leave it as future work.

The fourth condition is the most crucial condition. Dicio evaluates the condition at a fine-grained interval (every 10μs). It detects a sudden increase of PM media-level write bandwidth and throttles a BE job eagerly without waiting for the long delay to obtain the media-level BW consumption. Through our sensitively study, Dicio sets T_{WPQ} as 700cycles as we found that it detects a sudden increase well while raises infrequent false alarm.

In case of the last condition, we explain its role in §4.3. Dicio computes *estimated media-level write BW* by, {request-level BW $\times$ recent PM write-amplification ratio}, where request-level BW is measured at μs timescale with PMU. The write-amplification ratio is obtained by (media-level BW / request-level BW); where both BWs are measured with PM counters at *ms* timescale.

`Relaxing`. Dicio gives an extra core for a BE job and puts it under `Relaxing` state when *none of the above conditions for* `Throttling` *is met*. This requires the measurement of PM media-level BW which takes a few *ms*, so it can only happen every a few *ms*. This policy makes relaxing BE more conservative and safer at the expense of reduced BE throughput.

In order to improve BE throughput, we additionally relax it by prediction without the measurement of media-level BW when *all of the following conditions are met*:

(1) BE is under `Relaxing` state.
(2) Estimated media-level write BW $< T_{mediawrite}$
(3) None of the conditions above for `Throttling` (except the first two conditions) is met.

In this case, we assume the write-amplification ratio would remain stable until the next measurement of actual media-level BW. The last condition above for `Throttling` is to recover fast when the relaxed thread consumes lots of request-level BW so media-level BW would exceed the threshold, under the same assumption. If the assumption has broken, it will increase WPQ queuing delay soon and be detected by the fourth condition for `Throttling`.

Special case: single thread saturates PM BW. To address a special case that exists only in PM where even a thread saturates the entire BW, Dicio has a special routine. When it detects a thread saturates the entire BW so oscillates between two states as given 1 or 0 core, it adjusts the duty

[1]We note that conventional DRAM-by-DRAM performance interference scenarios could be handled here.

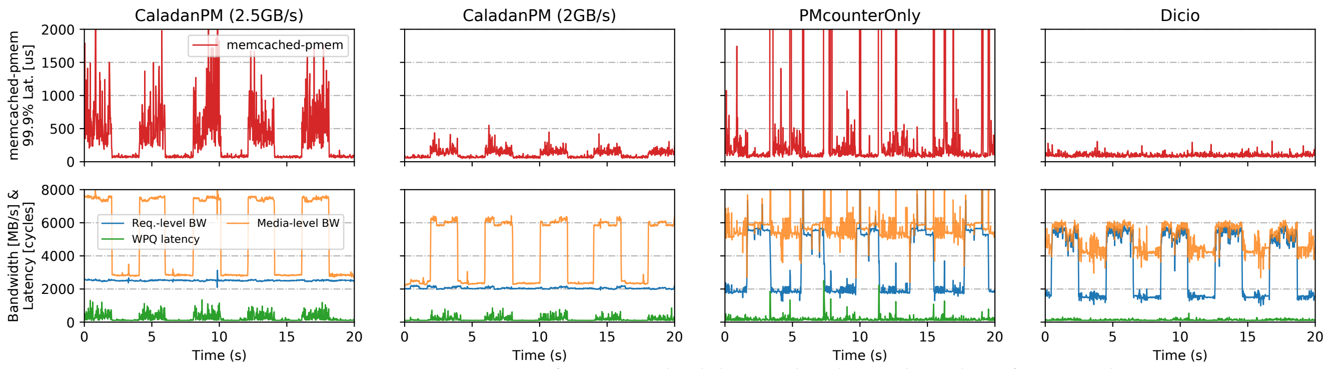


Figure 8: *Dicio can guarantee QoS for an LC job while providing better throughput for a BE job.*

cycle of the BE job so that it is given $0<x<1$ core where x is determined by estimated PM media BW consumption of it.

5 EVALUATION

Configuration. We evaluate Dicio on a machine with an Intel Xeon Gold 5220R (24 core, 48 hyperthread@2.2GHz) and 6×128GB Intel Optane DC Persistent Memory. The machines run Ubuntu 18.04 with kernel 5.1.0 and are connected via Mellanox ConnectX-5 25GbE NICs.

Comparison to other designs. There is no performance isolation system that considers PM. So we augment a recent system, Caladan [5] to monitor PM *request-level write bandwidth* by reading iMC counters, and call it `CaladanPM`. `CaladanPM` uses the same policy and mechanism as the original Caladan; it throttles or relaxes a BE job when the global PM request-level bandwidth is over or under the predefined threshold, respectively. We also compare to a system that controls PM *media-level bandwidth* using PM counters, and call it `PMcounterOnly`. It measures PM media-level bandwidth and then throttles or relaxes a BE job similar to `CaladanPM`. As the measurement of PM media-level bandwidth takes a few milliseconds, `PMcounterOnly` reacts at millisecond timescale. As an LC job, we run the PM-enabled version of `memcached`, `memcached-pmem` [20]. Since it utilizes PM to cache data, its performance depends on the degree of interference on PM. To be a fair comparison with the state-of-the-art, Caladan, we integrate Dicio to Shenango runtime [22] and modified the `memcached-pmem` to make it runs in Shenango runtime. We generate a load for it similar to YCSB-A workload which composes 50% of reads and 50% of writes [4]. We use the same load generator as in [5] running in a separate machine connected with Mellanox 25GbE NIC. To see the effect of bandwidth interference and bandwidth controller clearly, we pin `memcached-pmem` on 5-cores and do not manipulate the number of cores for it.

5.1 Mitigating PM-by-PM Interference

To show how Dicio manages PM bandwidth contention and achieves its goal, we colocate a synthetic microbenchmark, `PMantagonist` that heavily writes PM, with `memcached-pmem`. Similar to many real-world applications [1, 5, 14], the workload of `PMantagonist` changes dynamically. Its per-thread bandwidth consumption alters every 0.58 seconds. Also, its write-amplification ratio alters every 2 seconds to 1 or 3. We empirically found that some PM-unfriendly applications reach a write-amplification ratio of around 3.

Figure 8 shows how each system handles interference. Firstly, `CaladanPM` fails to keep QoS of LC and loses BE throughput. When we use 2.5GB/s as a threshold to PM request-level bandwidth, `PMantagonist` consumes 3× amplified media-level bandwidth and saturates PM. It results in increased WPQ latency (green line in Figure 8) and severe QoS violation to `memcached-pmem`: 99.9%-tile tail latency is 948μs. To prevent it, we need a lower threshold at the expense of reduced BE throughput; 2GB/s to achieve 169μs as 99.9%-tile tail latency. In addition, If we would like to provision for the worst-case where the write-amplification ratio is 4, we need a threshold around 1GB/s, which will compromise the throughput of the BE job more.

Secondly, if we control media-level bandwidth naively using only PM counter, it rather causes more catastrophic QoS violation (99.9th: 39788μs). When the write-amplification ratio or bandwidth consumption is changed, it takes a few milliseconds to be detected and addressed by `PMcounterOnly`. During this period, PM bandwidth is *fully* saturated and suffers a radical increase in WPQ latency. The sudden increase of PM latency prevents `memcached-pmem` to serve requests and QoS violation due to queueing delay even after `PMcounterOnly` resolves bandwidth contention. This reminds the importance of fine-grained and nimble performance isolation for PM.

On the other hand, Dicio guarantees QoS of the LC job and increased throughput for the BE job. It achieves 116μs of 99.9%-tile tail latency, which is comparable to the 90%-tile tail latency of `CaladanPM(2GB/s)`, 107μs. It is because it can detect a sudden change of workload immediately by increased WPQ latency and take action immediately. Also, it relaxes the BE job when media-level bandwidth is low so that provides

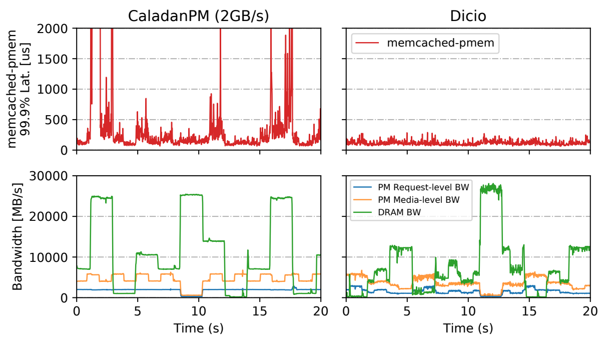


Figure 9: *Dicio can mitigate performance interference from both DRAM and PM.*

a much higher throughput for the BE job: average request-level bandwidth is 3430MB/s while CaladanPM(2GB/s) is only 2054MB/s. The benefit of Dicio would be larger when a BE is more PM-friendly (*i.e.* lower write-amplification ratio).

5.2 Mitigating Memory BW Interference

Now we colocate memcached-pmem with a more realistic synthetic BE job, MEMantagonist, to evaluate how Dicio addresses general memory bandwidth contention. MEMantagonist consumes both PM and DRAM bandwidth a lot, and the ratio of PM bandwidth and DRAM bandwidth fluctuates over time. In addition, similar to PMantagonist, the write-amplification ratio of PM traffic alters every second to 2 or 3.

Figure 9 depicts how CaladanPM and Dicio manage combined interference from PM and DRAM bandwidth. CaladanPM does not consider the combined effect of PM and DRAM bandwidth and tries to just keep PM request-level bandwidth and DRAM bandwidth lower than corresponding thresholds (2GB/s for PM, 25GB/s for DRAM). As a result, it fails to maintain the QoS of memcached-pmem: 99.9%-tile latency is 7998μs. We can see in Figure 9 that the degree of performance degradation depends on not only PM but also DRAM: at the same PM media-level bandwidth, tail latency increases as more DRAM bandwidth is consumed, and vice versa.

Meanwhile, Dicio can guarantee QoS for memcached-pmem. A sudden increase of memory bandwidth is detected by Dicio and it immediately throttles BE. Then Dicio relaxes BE until any of the conditions to throttle is met. This control loop keeps overall system bandwidths be not saturated and guarantees QoS: 99.9%-tile latency is 136μs, which is even lower than the 90%-tile latency of CaladanPM, 2646μs. However, BE throughput may be suboptimal due to the fixed thresholds. We leave adaptive tuning of thresholds as future work.

6 DISCUSSION & LIMITATION

Not using application-level metrics. Dicio doesn't take any hints from applications while some prior works make decisions based on app.-level information such as tail latency [21, 25]. While the information is useful, we believe the approaches are not adequate in our use cases. First, retrieving such information is challenging when we need to take action at μs timescale [22]. Second, Dicio does not trust applications because they may report maliciously to be assigned more resources.

Colocating more jobs. Currently, Dicio supports only one LC and one BE job. We believe PM performance isolation for multiple LCs and BEs would be challenging with current hardware support. When only an LC job and a BE job are colocated, Dicio could blame the BE and throttle it if PM bandwidth is saturated. However, when multiple same-priority jobs are colocated, Dicio needs to figure out which one should be throttled. To do so, it needs to know how much each process consumes each type of memory bandwidth (i.e. DRAM/PM read/write). Unfortunately, to our knowledge, there is no way to measure them separately on a per-core or per-process basis. For instance, the LLC miss counter and the Intel MBM, which were used to measure per-core DRAM bandwidth consumption in literature [5, 27], do not provide DRAM write bandwidth and PM write bandwidth separately. We have tried to separate four types of memory bandwidth from the aggregated bandwidth provided by Intel MBM, but found that MBM has several hardware bugs. First, as Intel has already admitted [9], it double-counts non-temporal write instructions that are widely used for PM. We additionally found that the bug conditionally occurs: double-counting does not happen when non-temporal write hits cache. Second, we found that bandwidth measurements are incorrect when vmovqda64 instruction is used. For these issues, we believe PM performance isolation between multiple applications at the same priority is challenging, without a proper hardware support. We leave it as future work to approximate BW consumption of each type of memory.

7 CONCLUSION

This paper proposes Dicio for controlling performance interference by PM. By monitoring media-level BW—instead of request-level BW—of PM, Dicio effectively curtails performance degradation even when a workload shows varying access patterns. We compare Dicio with CaladanPM and demonstrate that Dicio shows superior tail latency by regulating BW of PM and DRAM of a co-running process.

ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2020R1C1C1014940) and Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00503, Researches on next generation memory-centric computing system architecture)

REFERENCES

[1] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 405–417. https://www.usenix.org/conference/nsdi18/presentation/ardelean

[2] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[3] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

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

[6] Google. 2021. *Available first on Google Cloud: Intel Optane DC Persistent Memory*. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory

[7] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[8] Intel. 2020. *Intel(R) Optane(TM) 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.pdfl

[9] Intel. 2021. *2nd Gen Intel® Xeon® Scalable Processors Specification Update*. https://www.intel.la/content/www/xl/es/products/docs/processors/xeon/2nd-gen-xeon-scalable-spec-update.html

[10] Intel. 2021. *Events for Intel® microarchitecture code name Cascade Lake-X*. https://perfmon-events.intel.com/cascadelake_server.html

[11] Intel. 2021. *Intel® Memory Latency Checker v3.9*. https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html

[12] Intel. 2021. *Intel® Optane™ Persistent Memory*. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html

[13] Intel. 2021. *ipmctl*. https://github.com/intel/ipmctle

[14] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 519–532. https://www.usenix.org/conference/atc18/presentation/iorgulescu

[15] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[16] Giorgos Kappes and Stergios V. Anastasiadis. 2020. Libservices: Dynamic Storage Provisioning for Multitenant I/O Isolation. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 33–41. https://doi.org/10.1145/3409963.3410497

[17] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash = Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 345–359. https://doi.org/10.1145/3037697.3037732

[18] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 1–14. https://doi.org/10.1145/2829988.2787478

[19] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[20] Lenovo. 2021. *memcached-pmem*. https://github.com/lenovo/memcached-pmem

[21] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 450–462. https://doi.org/10.1145/2749469.2749475

[22] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[23] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3302424.3303963

[24] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. 2018. 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* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 5, 14 pages. https://doi.org/10.1145/3243176.3243211

[25] Tirthak Patel and Devesh Tiwari. 2020. 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)*. 193–206. https://doi.org/10.1109/HPCA47549.2020.00025

[26] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

[27] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. 2019. EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 16, 12 pages. https://doi.org/10.1145/3337821.3337863

[28] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[29] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang