Don't Forget the I/O When Allocating Your LLC

Yifan Yuan[1], Mohammad Alian[2], Yipeng Wang[3], Ren Wang[3], Ilia Kurakin[3], Charlie Tai[3], Nam Sung Kim[1]
[1]*UIUC*, [2]*University of Kansas*, [3]*Intel*
{yifany3, nskim}@illinois.edu alian@ku.edu {yipeng1.wang, ren.wang, ilia.kurakin, charlie.tai}@intel.com

***Abstract*—In modern server CPUs, last-level cache (LLC) is a critical hardware resource that exerts significant influence on the performance of the workloads, and how to manage LLC is a key to the performance isolation and QoS in the cloud with multi-tenancy. In this paper, we argue that in addition to CPU cores, high-speed I/O is also important for LLC management. This is because of an Intel architectural innovation – Data Direct I/O (DDIO) – that directly injects the inbound I/O traffic to (part of) the LLC instead of the main memory. We summarize two problems caused by DDIO and show that (1) the default DDIO configuration may not always achieve optimal performance, (2) DDIO can decrease the performance of non-I/O workloads that share LLC with it by as high as 32%.**

We then present IAT, the first LLC management mechanism that treats the I/O as the first-class citizen. IAT monitors and analyzes the performance of the core/LLC/DDIO using CPU's hardware performance counters and adaptively adjusts the number of LLC ways for DDIO or the tenants that demand more LLC capacity. In addition, IAT dynamically chooses the tenants that share its LLC resource with DDIO to minimize the performance interference by both the tenants and the I/O. Our experiments with multiple microbenchmarks and real-world applications demonstrate that with minimal overhead, IAT can effectively and stably reduce the performance degradation caused by DDIO.

***Index Terms*—cache partitioning, DDIO, performance isolation**

I. Introduction

The world has seen the dominance of Infrastructure-as-a-Service (IaaS) in cloud data centers. IaaS hides the underlying hardware from the upper-level tenants and allows multiple tenants to share the same physical platform with virtualization technologies such as virtual machine (VM) and container (*i.e.*, workload collocation) [46, 59]. This not only facilitates the operation and management of the cloud but also achieves high efficiency and hardware utilization.

However, the benefits of the workload collocation in the multi-tenant cloud do not come for free. Different tenants may contend with each other for the shared hardware resources, which often incurs severe performance interference [37, 61]. Hence, we need to carefully allocate and isolate hardware resources for tenants. Among these resources, the CPU's last-level cache (LLC), with much higher access speed than the DRAM-based memory and limited capacity (*e.g.*, tens of MB), is a critical one [65, 78].

There have been a handful of proposals on how to partition LLC for different CPU cores (and thus tenants) with hardware or software methods [13, 39, 42, 62, 71, 72, 76]. Recently, Intel® Resource Director Technology (RDT) enables LLC partitioning and monitoring on commodity hardware in cache way granularity [22]. This spurs the innovation of LLC management mechanisms in the real world for multi-tenancy and workload collocation [11, 57, 61, 66, 73, 74]. However, the role and impact of high-speed I/O by Intel's Data Direct I/O (DDIO) technology [31] has not been well considered.

Traditionally, inbound data from (PCIe-based) I/O devices is delivered to the main memory, and the CPU core will fetch and process it later. However, such a scheme is inefficient w.r.t. data access latency and memory bandwidth consumption. Especially with the advent of I/O devices with extremely high bandwidth (*e.g.*, 100Gb network device and NVMe-based storage device) – to the memory, CPU is not able to process all inbound traffic in time. As a result, Rx/Tx buffers will overflow, and packet loss occurs. DDIO, instead, directly steers the inbound data to (part of) the LLC and thus significantly relieves the burden of the memory (see Sec. II-B), which results in low processing latency and high throughput from the core. In other words, DDIO lets the I/O share LLC's ownership with the core (*i.e.*, I/O can also read/write cachelines), which is especially meaningful for I/O-intensive platforms.

Typically, DDIO is completely transparent to the OS and applications. However, this may lead to sub-optimal performance since (1) the network traffic fluctuates over time, and so does the workload of each tenant, and (2) I/O devices can contend with the cores for the LLC resource. Previously, researchers [18, 54, 56, 69] have identified the "**Leaky DMA**" problem, *i.e.*, the device Rx ring buffer size can exceed the LLC capacity for DDIO, making data move back and forth between the LLC and main memory. While ResQ [69] proposed a simple solution for this by properly sizing the Rx buffer, our experiment shows that it often undesirably impacts the performance (see Sec. III-A). On the other hand, we also identify another DDIO-related inefficiency, the "**Latent Contender**" problem (see Sec. III-B). That is, without DDIO awareness, the CPU core is assigned with the same LLC ways that the DDIO is using, which incurs inefficient LLC utilization. Our experiment shows that this problem can incur 32% performance degradation even for non-I/O workloads. These two problems indicate the deficiency of pure core-oriented LLC management mechanisms and necessitate the configurability and awareness of DDIO for extreme I/O performance.

To this end, we propose IAT, the first, to the best of our knowledge, I/O-aware LLC management mechanism. IAT periodically collects statistics of the core, LLC, and I/O activities using CPU's hardware performance counters. Based on the statistics, IAT determines the current system state with a finite state machine (FSM), identifies whether the contention comes from the core or the I/O, and then adaptively allocates the LLC ways for either cores or DDIO. This helps alleviate

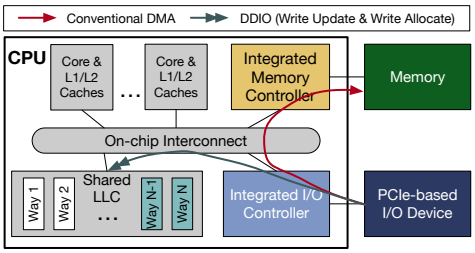


Fig. 1: Typical cache organization in modern server CPU, conventional DMA path, and DDIO for I/O device.

the impact of the Leaky DMA problem. Besides, IAT sorts and selects the least memory-intensive tenant(s) to share LLC ways with DDIO by shuffling the LLC ways allocation, so that the performance interference between the core and I/O (*i.e.*, the Latent Contender problem) can be reduced.

We develop IAT as a user-space daemon in Linux and evaluate it on a commodity server with high-bandwidth NICs.Our results with both microbenchmarks and real applications show that compared to a case running a single workload, applying IAT in co-running scenarios can restrict the performance degradation of both networking and non-networking applications to less than 10%, while without IAT, such degradation can be as high as $\sim$30%.

To facilitate the future DDIO-related research, we make our enhanced RDT library (`pqos`) with DDIO functionalities public at https://github.com/FAST-UIUC/iat-pqos.

II. Background

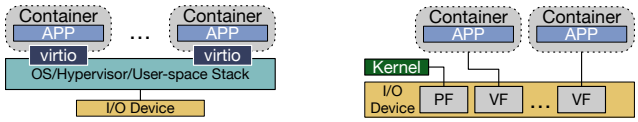
A. Managing LLC in Modern Server CPU

As studied by prior research [39, 49], sharing LLC can cause performance interference among the collocated VM/containers. This motivates the practice of LLC monitoring and partitioning on modern server CPUs. Since the Xeon® E5 v3 generation, Intel began to provide RDT [28] for resource management in the memory hierarchy. In RDT, Cache Monitoring Technology (CMT) provides the ability to monitor the LLC utilization by different cores; Cache Allocation Technology (CAT) can assign LLC ways to different cores (and thus different tenants) [22][1]. Programmers can leverage these techniques by simply accessing corresponding Model-Specific Registers (MSRs) or using high-level libraries [32]. Furthermore, dynamic mechanisms can be built atop RDT [11, 42, 57, 61, 66, 73, 74].

B. Data Direct I/O Technology

Conventionally, direct memory access (DMA) operations from a PCIe device use memory as the destination. That is, when being transferred from the device to the host, the data will be written to the memory with addresses designated by the device driver, as demonstrated in Fig. 1. Later, when the CPU core has been informed about the completed transfer, it will fetch the data from the memory to the cache hierarchy for future processing. However, due to the dramatic bandwidth increase

[1]With CAT, a core has to be assigned with at least one LLC way. A core (1) can only allocate cachelines to its assigned LLC ways, but (2) can still load/update cachelines from all the LLC ways.



(a) Aggregation. **(b)** Slicing.

Fig. 2: Two models of tenant-device interaction.

of the I/O devices over the past decades, two drawbacks of such a DMA scheme became salient: (1) Accessing memory is relatively time-consuming, which can potentially limit the performance of data processing. Suppose we have 100Gb inbound network traffic. For a 64B packet with 20B Ethernet overhead, the packet arrival rate is 148.8 Mpps. This means any component on the I/O path, like I/O controller or core, has to spend no more than 6.7ns on each packet, or packet loss will occur. (2) It consumes much memory bandwidth. Again with 100Gb inbound traffic, for each packet, it will be written to and read from memory at least once, which easily leads to $100Gb/s \times 2 = 25GB/s$ memory bandwidth consumption.

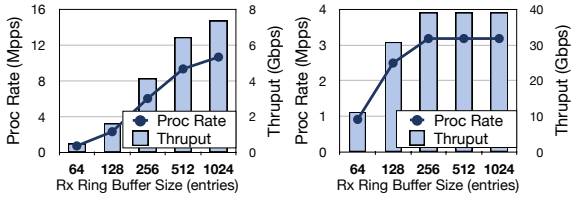
To relieve the burden of memory, Intel proposed Direct Cache Access (DCA) technique [23], allowing the device to write data directly to CPU's LLC. In modern Intel® Xeon® CPUs, this has been implemented as Data Direct I/O Technology (DDIO) [31], which is transparent to the software. Specifically, as shown in Fig. 1, when the CPU receives data from the device, an LLC lookup will be performed to check if the cacheline with the corresponding address is present with a valid state. If so, this cacheline will be updated with the inbound data (*i.e.*, `write update`). If not, the inbound data will be allocated to the LLC (*i.e.*, `write allocate`), and dirty cachelines may be evicted to the memory. By default, DDIO can only perform `write allocate` on two LLC ways (*i.e.*, Way $N-1$ and Way N in Fig. 1). Similarly, with DDIO, a device can directly `read` data from the LLC; if the data is not present, the device will read it from the memory but not allocate it in the LLC. Prior comprehensive studies [1, 36, 44] show that in most cases (except for those with persistent DIMM), compared to the DDIO-disabled system, enabling DDIO on the same system can improve the application performance by cutting the memory access latency and reducing memory bandwidth consumption. Note that even if DDIO is disabled, inbound data will still be in the cache at first (and immediately evicted to the memory). This is a performance consideration since after getting into the coherence domain (*i.e.*, cache), read/write operations with no dependency can be performed out-of-order.

Although DDIO is Intel-specific, other CPUs may have similar concepts (*e.g.*, ARM's Cache Stashing [5]). Most discussions in this paper are also applicable to them.

C. Tenant-device Interaction in Virtualized Servers

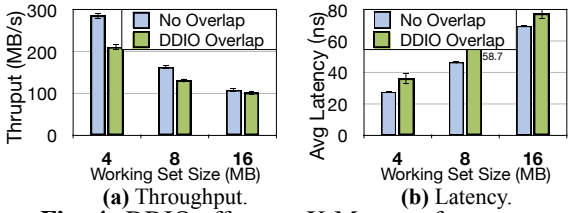
Modern data centers adopt two popular models to organize the I/O devices in multi-tenant virtualized servers with different trade-offs. As shown in Fig. 2, the key difference is the way they interact with the physical device.

In the first model, logically centralized software stacks have been deployed for I/O device interaction. It can run in OS, hypervisor, or even user-space. For example, SDN-compatible



(a) 64B small packet. **(b)** 1.5KB large packet.

Fig. 3: *l3fwd* results with different Rx ring sizes in RFC2544.



(a) Throughput. **(b)** Latency.

Fig. 4: DDIO effect on X-Mem performance.

virtual switches, such as Open vSwitch (OVS) [60] and VFP [15], have been developed for NIC. Regarding SSD, SPDK [75] is a high-performance and scalable user-space stack. As demonstrated in Fig. 2a, the software stack controls the physical device and sends/receives packets to/from it. Tenants are connected to the device via interfaces like virtio [64]. Since all traffic in this model needs to go through the software stack, we call this model "aggregation".

In the second model (Fig. 2b), the hardware-based single root input/output virtualization (SR-IOV) technique [10] is leveraged. With SR-IOV, a single physical device can be virtualized to multiple virtual functions (VFs). While the physical function (PF) is still connected to the host OS/hypervisor, we can bind the VFs directly to the tenants (*i.e.*, host-bypass). In other words, the basic switching functionality is offloaded to the hardware, and each tenant directly talks to the physical device for data reception and transmission. Since this model disaggregates the hardware resource and assigns it to different tenants, it is also called "slicing". Note that many hardware-offloading solutions for multi-tenancy [16, 45, 52] can be intrinsically treated as the slicing model.

III. Motivation: The Impact of I/O on LLC

A. The Leaky DMA Problem

The "Leaky DMA" problem has been observed by multiple papers [18, 54, 56, 69]. That is, since, by default, there are only two LLC ways for DDIO's `write allocate`, when the inbound data rate (*e.g.*, NIC Rx rate) is higher than the rate that CPU cores can process, the data in LLC waiting to be processed is likely to (1) be evicted to the memory by the newly incoming data, and (2) later be brought back to the LLC again when a core needs it. This is especially significant for large packets as with the same in-flight packet count, larger packets consume more cache space than smaller packets. Hence, this incurs extra memory read/write bandwidth consumption and increases the processing latency of each packet, eventually leading to a performance drop.

In ResQ [69], the authors propose to solve this problem by reducing the size of the Rx/Tx buffers. However, this workaround has drawbacks. In a cloud environment, tens or even hundreds of VMs/containers can share a couple of physical ports through the virtualization stack [24, 47]. If the total count of entries in all buffers is maintained below the default DDIO's LLC capacity, each VM/container only gets a very shallow buffer. For example, in an SR-IOV setup, we have 20 containers, each assigned a virtual function to receive traffic. To guarantee all buffers can be accommodated in the default DDIO's cache capacity (several MB), each buffer can only have a small number of entries. A shallow Rx/Tx buffer can lead to severe packet drop issues, especially when we have bursty traffic, which is ubiquitous in modern cloud services [4]. Hence, while this setting may work with statically balanced traffic, *dynamically imbalanced* traffic with certain "heavy hitter" container(s) will incur a performance drop.

Here we run a simple experiment to demonstrate such inefficiency (see Sec. VI-A for details of our setup). We set up DPDK *l3fwd* application on a single core of the testbed for traffic routing. It looks at the header of each network packet up against a flow table of 1M flows (to emulate real traffic). The packet is forwarded if a match is found. We run an RFC2544 test [53] (*i.e.*, measure the maximum throughput when there is zero packet drop) from a traffic generator machine with small (64B) or large (1.5KB) packets. From the results in Fig. 3, we observe that for the large-packet case (Fig. 3b), shrinking Rx buffer size may not be a problem – the throughput does not drop until the size is 1/8 of the typical value. However, the small-packet case is in a totally different situation (Fig. 3a) – by cutting half the buffer size (from 1024 to 512), the maximum throughput can drop by 13%. If we use a small buffer of 64 entries, the throughput is less than 10% of the original throughput. Between these two cases, the key factor is the packet processing rate. With a higher rate, small-packet traffic tends to press the CPU core more intensively (*i.e.*, less idle and busy polling time). As a result, any skew will lead to a producer-consumer imbalance in the Rx buffer, and a shallow buffer is easier to overflow (*i.e.*, packet drop). Hence, sizing the buffer is not a panacea for compound and dynamically changing traffic in the real world. **This motivates us not merely to size the buffer but also to tune the DDIO's LLC capacity adaptively.**

B. The Latent Contender Problem

We identify a second problem caused by DDIO – the "Latent Contender" problem. That is, since most current LLC management mechanisms are I/O-unaware, when allocating LLC ways for different cores with CAT, they may unconsciously allocate DDIO's LLC ways to certain cores running LLC-sensitive workloads. This means that even if these LLC ways are entirely isolated from the core's point of view, DDIO is actually still contending with the cores for the capacity.

We run another experiment to further demonstrate this problem. In this experiment, we first set up a container bound to one CPU core, two LLC ways (*i.e.*, Way $0-1$), and one NIC VF. This container is running DPDK *l3fwd* with 40Gb traffic. We then set up another container, which is running on another core. We run X-Mem [19], a microbenchmark for cloud application'

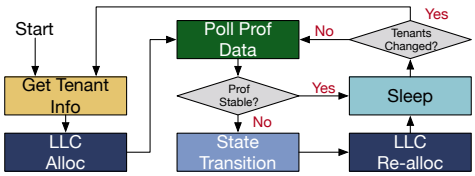


Fig. 5: Execution flow of IAT.

memory behavior characteristics. We increment the working set of X-Mem from 4MB to 16MB and apply the random-read memory access pattern to emulate real applications' behavior. We measure the average latency and throughput of X-Mem in two cases: (1) the container is bound to two dedicated LLC ways (*i.e.*, no overlap), and (2) the container is bound to the two DDIO's LLC ways (*i.e.*, DDIO overlap). As the results in Fig. 4 show, even if X-Mem and *l3fwd* do not explicitly share any LLC ways from the core point of view, DDIO may still worsen X-Mem's throughput by up to 26.0% and average latency by up to 32.0%. **This lets us think of how we should select tenants that share LLC ways with DDIO.**

Some previous works propose to never use DDIO's LLC ways at all for core's LLC allocation [14, 69]. We argue that they are sub-optimal for two fundamental reasons. (1) We have motivated that we should dynamically allocate more/less LLC ways for DDIO (Sec. III-A). If, in some cases, DDIO is occupying a large portion of LLC, there will be little room for cores' LLC allocation. (2) When the I/O traffic does not press the LLC, isolating DDIO's LLC ways is wasteful. It is better to make use of this LLC portion more efficiently.

IV. IAT Design

IAT is an I/O-aware LLC management mechanism that makes better use of DDIO technology for various situations in multi-tenant servers. When IAT detects an increasing amount of LLC misses from DDIO traffic, it first decides whether the misses are caused by the I/O traffic or the application running in the cores. Based on the decision, IAT allocates more or fewer LLC ways to either the core or the DDIO to mitigate the core-to-I/O or I/O-to-I/O interference. IAT can also shuffle the tenants' LLC allocation to further reduce core-I/O contention. Specifically, IAT performs six steps to achieve its objective, as depicted in Fig. 5.

A. Get Tenant Info and LLC Alloc

At initialization (or tenants change), IAT obtains the tenants' information and the available hardware resources through the `Get Tenant Info` step. Regarding hardware resources, it needs to know and remember the allocated cores and LLC ways for each tenant; regarding software, it needs to know two things. (1) Whether the tenant's workload is "I/O" (*e.g.*, "networking" in this paper) or not. This can help IAT decide whether a performance fluctuation is caused by I/O or not since non-I/O applications also have execution phases with different behaviors. Note that a non-I/O tenant may maintain the connection to the I/O device (for ssh, *etc.*, but not intensive I/O traffic). (2) The priority of each tenant. To improve resource utilization, modern data centers tend to collocate workloads with different priorities on the same physical platform [20, 49]. Since the cluster management software commonly provides hints for such priorities [68], IAT can obtain such information directly. In IAT, we assume two possible priorities (there can be more in real-world deployment) for each workload – "performance-critical (PC)" and "best-effort (BE)". Although the software stack in the aggregation model (*e.g.*, virtual switch) is not a tenant, we still keep the record for it and assign it with a special priority.

After getting the tenant information, IAT allocates the LLC ways for each tenant accordingly (*i.e.*, `LLC Alloc`).

B. Poll Prof Data

In this step, IAT polls the performance status of each tenant to decide the optimal LLC allocation. Using the application-level metrics (operations per second, tail latency, *etc.*) is not a good strategy since they vary across tenants. Instead, we directly get the profiling statistics of the following hardware events from the hardware counters.

Instruction per cycle (IPC). IPC is a commonly-used metric to measure the execution performance of a program on a CPU core [7, 37]. Although it is sensitive to some microarchitectural factors, such as branch-misprediction and serializing instructions, it is stable in our timescale (*i.e.*, hundreds of ms to s). We use it to detect tenants' performance degradation and improvement.

LLC reference and miss. LLC reference and miss counts reflect the memory access characteristic of a workload. We can also derive the LLC miss rate from these values, which is yet another critical metric for workload performance [70].

DDIO hit and miss. DDIO hit is the number of DDIO transactions that apply `write update`, meaning the targeted cacheline is already in the LLC; DDIO miss reflects the number of DDIO transactions that apply `write allocate`, which indicates a victim cacheline has to be evicted out of the LLC for the allocation. These two metrics reflect the intensity of the I/O traffic and the pressure it puts on the LLC.

IPC and LLC ref/miss are per-core metrics. If a tenant is occupying more than one core, we aggregate the values as the tenant's result. DDIO hit/miss are chip-wide metrics, which means we only need to collect them once per CPU and cannot distinguish between those caused by different devices or applications.

After collecting these events' data, IAT will compare them with those collected during the previous iteration. If the delta of one of the events is larger than a threshold *THRESHOLD_STABLE*, IAT will jump to the `State Transition` step to determine how to (potentially) adjust the LLC allocation. Otherwise, it will regard the system's status as unchanged and jump to the `Sleep` step, waiting for the next iteration. Also, there are three cases where we do not jump to the `State Transition` step. (1) If we only see IPC change but no significant LLC reference/miss and DDIO hit/miss count change, we assume that this change is attributed to neither cache/memory nor I/O. (2) If we observe IPC change of a non-I/O tenant (no DDIO overlap) with corresponding LLC reference/miss change but no significant DDIO hit/miss count change over the system, we know this is mainly caused by

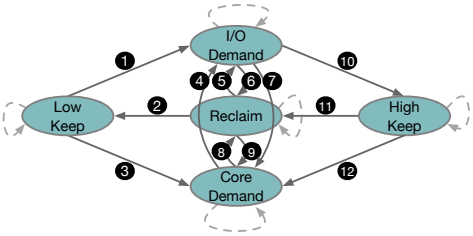


Fig. 6: State transition diagram of IAT.

CPU core's demand of LLC space. In this case, other existing mechanisms [11, 57, 66, 73, 74] can be called to allocate LLC ways for the tenant. (3) If we observe IPC change of a non-I/O tenant (with DDIO overlap) with corresponding LLC reference/miss change and DDIO hit/miss change, we will try shuffling LLC ways allocation (see Sec. IV-D) at first.

C. State Transition

The core of IAT design is a system-wide Mealy FSM, which decides the current system state based on the data from `Poll Prof Data`. For each iteration, the state transition (including self-transition) is triggered if changes happened in `Poll Prof Data`; otherwise, IAT will remain in the previous state. Fig. 6 shows the five states.

Low Keep. In this state, the I/O traffic is not intensive and does not press the LLC (*i.e.*, does not contend with cores for the LLC resource). IAT is in this state if the DDIO miss count is small. Here the DDIO hit count is not necessarily small, since if most DDIO transactions can end up with `write update`, LLC thrashing will not happen. Because I/O traffic does not trigger extensive cache misses, we keep the number of LLC ways for DDIO at the minimum value (*DDIO_WAYS_MIN*).

High Keep. This is a state where we have already allocated the largest number of LLC ways for DDIO (*DDIO_WAYS_MAX*), regardless of the numbers of DDIO miss and hit. We set such an upper bound because we do not expect DDIO to compete with cores without any constraints across the entire LLC, especially when there is a PC tenant running in the system with high priority.

I/O Demand. This is a state where the I/O contends with cores for the LLC resource. In this state, I/O traffic becomes intensive, and the LLC space for `write update` cannot satisfy the demand of DDIO transactions. As a result, `write allocate` (DDIO miss) happens more frequently in the system, which leads to a large amount of cacheline evictions.

Core Demand. In this state, the I/O also contends with cores for the LLC resource, but the reason is different. Specifically, now the core demands more LLC space. In other words, a memory-intensive I/O application is running on the core. As a result, the Rx buffer is frequently evicted from the LLC ways allocated for the core, leading to decreased DDIO hits and increased DDIO misses.

Reclaim. Similar to Low Keep, in this state, the I/O traffic is not intensive. The difference is that the number of LLC ways for DDIO is at a medium level, potentially wasteful. In this case, we should consider reclaiming some LLC ways from DDIO. Also, the LLC ways for a specific tenant can be more than enough, motivating us to reclaim LLC ways from the core.

IAT is initialized from the Low keep state. When the number of DDIO miss is greater than a threshold *THRESHOLD_MISS_LOW*, it indicates that the current LLC ways for DDIO are insufficient. IAT determines the next state by further examining the value of DDIO hit and LLC reference. Decrease of the DDIO hit count with more LLC references implies the core(s) is increasingly contending the LLC with DDIO, and the entries in the Rx buffer(s) are frequently evicted from the LLC. In this case, we move to the Core Demand state (❸). Otherwise (*i.e.*, an increase of DDIO hit count), we move to the I/O Demand state (❶) since the DDIO miss is attributed to the more intensive I/O traffic.

In the Core Demand state, if we observe the decrease of the DDIO miss count, we regard it as a signal of system balance and will go back to the Reclaim state (❽). If we observe an increase of DDIO miss count and no fewer DDIO hits, we go to I/O Demand state (❹) since right now, the core is no longer the major competitor. If we observe neither of the two events, IAT will stay at the Core Demand state.

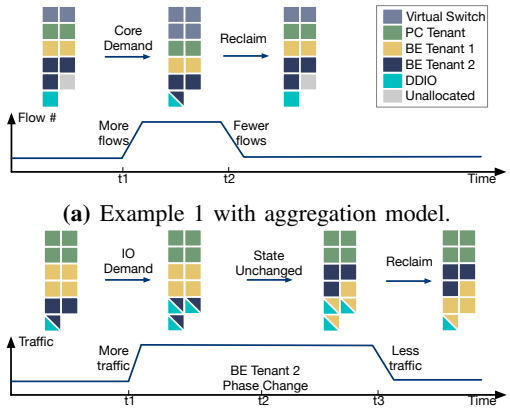
In the I/O Demand state, if we still observe a large amount of DDIO miss, we keep in this state until we have allocated *DDIO_WAYS_MAX* number of ways to DDIO and then transit to the High Keep state (❿). If a significant degradation of DDIO miss appears, we assume the LLC capacity for DDIO is over-provisioned and will go to the Reclaim state (❻). Meanwhile, fewer DDIO hits and stable or even more DDIO misses indicate that core is contending LLC, so we go to the Core Demand state (❼). Also, the High Keep state obeys the same rule (⓫ and ⓬).

We keep in the Reclaim state if we do not observe a meaningful increase of DDIO miss count until we have reached the *DDIO_WAYS_MIN* number of LLC ways for DDIO, then we move to the Low Keep state (❷). Otherwise, we move to the I/O Demand state to allocate more LLC ways for DDIO to amortize the pressure from intensive I/O traffic (❺). At the same time, if we also observe a decrease in DDIO hit count, we will go to the Core Demand state (❾).

D. LLC Re-alloc

After the state transition, IAT will take the corresponding actions, *i.e.*, re-allocate LLC ways for DDIO or cores.

First, IAT changes the number of LLC ways that are assigned to DDIO or tenants. Specifically, in the I/O Demand state, IAT increases the number of LLC ways for DDIO by one per iteration (miss-curve-based increment like UCP [62] can also be explored, same below). In the Core Demand state, IAT increases the number of LLC ways for the selected tenant by one per iteration. In the Low Keep and High Keep states, IAT does not change the LLC allocation. In the Reclaim state, IAT reclaims one LLC way from DDIO or core per iteration, depending on the values it observes (*e.g.*, smaller LLC miss count of the system or smaller LLC reference count of a tenant). All idle ways are in a pool to be selected for allocation. Since the current CAT only allows a core to have consecutive LLC



(a) Example 1 with aggregation model.

(b) Example 2 with slicing model.

Fig. 7: Two examples of LLC allocation with IAT.

ways, the selection of the idle ways should try to be consecutive with the existing allocation. Otherwise, shuffling may happen.

IAT should identify the workload that requires more or fewer LLC ways in the Core Demand and Reclaim states. The mechanism depends on the models of the tenant-device model we are applying. In the aggregation model, all the Rx/Tx buffers are allocated and managed by the centralized software stack. This means a performance drop of the software stack can bottleneck the performance of the I/O applications running in the attached tenants. So, in this case, IAT increases/decreases the number of LLC ways for software stack's cores at first. However, in the slicing model, each VF's Rx/Tx buffers are managed by the tenants themselves. IAT selects the tenant that needs more LLC ways from all I/O-related tenants by sorting their delta of LLC miss rate (in percentage) between the current and the last iteration and chooses the one with the most LLC miss rate increase. In this way, we are able to satisfy the LLC demand of the corresponding cores, thus reducing the DDIO miss.

Second, IAT will shuffle the LLC ways that have been assigned to tenants, *i.e.*, properly select the tenants that have minimal LLC pressure and move their assigned ways to the ways of DDIO. As we discussed in Sec. III-B, sharing LLC ways with DDIO may incur performance degradation of the core, even if the core is running a non-I/O workload. Hence, it is necessary to reduce such interference. First of all, we should avoid any core-I/O sharing of LLC ways if LLC ways have not been fully allocated. If sharing is necessary, intuitively, the tenants running PC workloads (high priority) should be isolated from LLC ways for DDIO as much as possible. IAT tries its best to only let BE tenants share ways with DDIO. Meanwhile, we do not want the BE tenants to contend LLC with DDIO too much since the PC tenants' performance is correlated to DDIO's [56, 69]. So before shuffling, IAT sorts all the BE tenants by their LLC reference count in the current iteration and chooses the one(s) with the smallest value to share LLC ways with DDIO. Note that according to Footnote 1, after shuffling, a tenant can still access its data in previously assigned LLC ways until it has been evicted by other tenants. Hence, shuffling will not lead to sudden cache under-utilization and performance degradation.

E. Sleep

The application performance may take some time to become stable after the `LLC Re-alloc` step. Also, polling the performance counters is not free (see Sec. VI-D). Hence, it is necessary to select a proper polling interval for IAT (*e.g.*, one second in our experiments). During this interval, IAT will simply sleep to let the OS schedule other tasks on the core. After each `Sleep`, if IAT is informed about changes of tenants (*e.g.*, tenant addition/removal, application phase change), it will go through the `Get Tenant Info` and `LLC Alloc` steps. Otherwise, it will conduct the next iteration of `Poll Prof Data`.

F. Putting It All Together: Two Examples

We use two tangible examples with NIC as the I/O device to illustrate how IAT works (see Fig. 7). In the first example (Fig. 7a) with the aggregation model, the throughput of network traffic is fixed. We have three tenants, one PC and two BE, each assigned with different LLC ways. In the beginning, the flow count of the network traffic is small, and BE Tenant 2 shares LLC ways with DDIO. At *t1*, a large number of flows appear in the traffic. As a result, the flow table in the virtual switch becomes larger and requires more space than the two LLC ways that are already assigned. Hence, IAT detects more DDIO misses and fewer DDIO hits and goes to the Core Demand state. Two more LLC ways are then assigned to the virtual switch (one for each iteration) so that the system reaches a new balance. To make room for the virtual switch, we shift the LLC ways of other tenants and let BE Tenant 2 share LLC ways with DDIO. At *t2*, many flows have ended, and there is no need for the virtual switch to maintain a big flow table in the LLC. IAT goes to the Reclaim state and reclaims two LLC ways from the virtual switch. Also, since now we have idle LLC ways, we remove the core-I/O sharing of LLC ways.

In the second example (Fig. 7b) with the same tenants setup but the slicing model, the throughput of the network traffic begins from a low level. At *t1*, more traffic comes into PC Tenant, and the number of LLC ways for DDIO becomes insufficient, which leads to more DDIO misses. IAT detects this situation and transits to the I/O Demand state, allocating more LLC ways for DDIO. At *t2*, the workload in BE Tenant 2 enters a new phase, which is LLC-consuming. IAT notices this by the delta of LLC reference count and let BE Tenant 1, which consumes less LLC, share the LLC ways with DDIO to reduce the performance interference. At *t3*, the amount of incoming network traffic is decreasing, and the LLC capacity for DDIO is more than enough. Thus, IAT can reclaim some LLC ways from DDIO.

V. Implementation

We implement IAT as a Linux user-space daemon, which is transparent to the application and the OS. Currently, we choose user-space implementation because it is more portable and flexible. However, IAT can be implemented in the kernel space as well. Since the x86 instructions for MSR manipulation (`rdmsr` and `wrmsr`) are ring-0, a kernel-space implementation can potentially have lower monitoring and control overhead. Another possibility is implementing IAT in the CPU power

TABLE I: Configuration of Intel® Xeon® 6140 CPU.

Cores	18 cores, 2.3GHz
Caches	8-way 32KB L1D/L1I, 16-way 1MB L2, **11-way 24.75MB non-inclusive shared LLC (split to 18 slices)**
Memory	Six DDR4-2666 channels

controller [77]. Note that IAT can also be integrated into other CPU resource management systems [57, 73, 74].

LLC allocation. For standard CAT functionalities (*i.e.*, allocating LLC ways for cores), we leverage APIs from the Intel `pqos` library [32]. To better isolate different applications and demonstrate the influence of DDIO, we do not allow LLC ways sharing across different tenants (but in real-world deployment, sharing should be explored [11, 66, 73]). For changing and querying the LLC ways for DDIO, we write and read the DDIO-related MSRs [26] via the `msr` kernel module.

Profiling and monitoring. Similarly, we use `pqos`'s APIs for regular profiling and monitoring (LLC miss, IPC, *etc.*). For monitoring DDIO's hit and miss, we use the uncore performance counters [30]. Modern Intel CPUs apply the non-uniform cache access (NUCA) architecture [41] to physically split the LLC into multiple slices. To reduce the monitoring overhead, for each DDIO event, we only use the performance counters in the Caching and Home Agent (CHA, the controller of each LLC slice in Intel CPUs) of one LLC slice. Since modern Intel CPUs apply a hashing mechanism for LLC addressing, the data (from both the core and the DDIO) is distributed to all the LLC slices evenly [34, 51]. Hence, by only accessing one LLC slice's performance counters, we can infer the full picture of the DDIO traffic by multiplying it by the number of slices.

Tenant awareness. Since CAT assigns LLC ways to cores, we need to know the core affiliation of each tenant. For simplicity, we keep such affiliation records in a text file. When the daemon is starting or is notified of a change, it will parse the records from this file. In the real-world cloud environment, IAT can have an interface to the orchestrator or scheduler to query the affiliation information dynamically.

VI. Evaluation

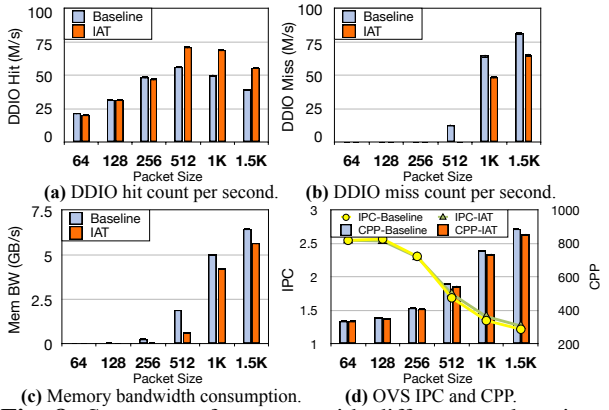
A. Setup

Hardware. We do experiments on a quad-socket Intel server with Xeon® Scalable Gold 6140 CPUs@2.3GHz [29] (Hyper-Threading and Turbo Boost disabled). See Tab. I for CPU configuration. It has 512GB DDR4 memory and two Intel® XL710 40GbE NICs [27] (both attached to socket-0). We connect each NIC directly to another server as the traffic generator. Using higher-bandwidth NIC will make the difference more significant but will not change the observation and conclusion.

System software. Since the current DDIO does not support remote socket [43], we run all experiments on socket-0. To reflect the multi-tenant cloud environment, we run applications in docker containers. For network connectivity, we have two models. (1) *Aggregation*: we connect the physical NICs and containers via OVS [60] (DPDK [25]-based). (2) *Slicing*: we bind a VF of the physical NIC to each container with SR-IOV. By default, we use 1024 entries as the Rx/Tx buffer size. For the containers that require a TCP/IP stack, we use DPDK-ANS [3] for high performance. Both the host and the containers run Ubuntu 18.04. To measure IAT's absolute overhead and not disturb tenants, we run IAT daemon on a dedicated core[2].

TABLE II: IAT parameters.

Name	Value
THRESHOLD_STABLE	3%
THRESHOLD_MISS_LOW	1M/s
DDIO_WAYS_MIN/MAX	1/6
Sleep interval	1 second



(a) DDIO hit count per second. **(b)** DDIO miss count per second.

(c) Memory bandwidth consumption. **(d)** OVS IPC and CPP.

Fig. 8: System performance with different packet sizes.

IAT parameters. We use empirical parameters (see Tab. II) to keep the balance of stability and agility. They can be tuned for various QoS requirements and hardware. The parameter sensitivity is similar to dCAT [74].

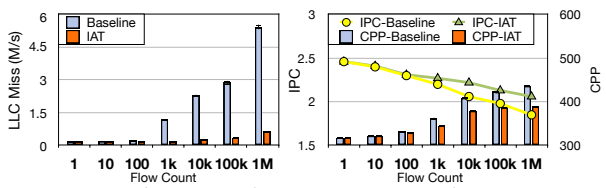
B. Microbenchmark Results

We first isolate the two problems in Sec. III and separately verify whether IAT can alleviate them in microbenchmarks.

Solving the Leaky DMA problem. Applying the aggregation model, we set up two containers running DPDK *test-pmd* (a simple program that bounces back the Rx traffic), each with two dedicated cores and one dedicated LLC way, both connected to OVS, which is running on two dedicated cores and two dedicated LLC ways. Also, the two NICs are connected to OVS. We insert four rules to OVS: "NIC0->Container0", "NIC1->Container1", "Container0->NIC0", and "Container1->NIC1". Both NICs send single-flow traffic with line rate. With such settings, the LLC miss of OVS itself is negligible and will not affect the performance. We start the experiment from 64B packet size, and over time, when OVS's performance gets stable, we double the packet size until the MTU size (1.5KB).

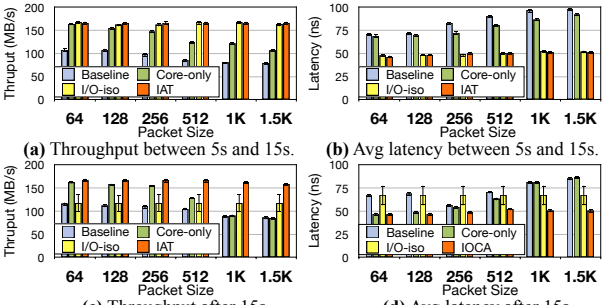
We collect the performance numbers of baseline (*i.e.*, default DDIO configuration without IAT, but with basic static CAT for cores) and IAT case and show them in Fig. 8. The most fundamental results are DDIO hit count (Fig. 8a) and miss count (Fig. 8b). When the packet size is small, the default two LLC ways for DDIO are enough to contain the on-the-fly inbound packets; however, as packet size increases

[2]We do encourage collocating IAT with other (ligh-weight) applications in real world deployment.



(a) LLC miss per second. **(b)** IPC and CPP.

Fig. 9: OVS performance with different flow counts.



(a) Throughput between 5s and 15s. **(b)** Avg latency between 5s and 15s.

(c) Throughput after 15s. **(d)** Avg latency after 15s.

Fig. 10: The (stable) performance of X-Mem in container 4.

over time, on-the-fly packets put much more pressure on the LLC, and thus the default two LLC ways become deficient, which is reflected in the increase of DDIO miss count. At this time, IAT detects the unstable status and transits its state to I/O Demand, allocating more LLC ways for DDIO (one by each time). As a result, the DDIO hit count with IAT is higher than the baseline, and the DDIO miss count is lower. This leads to better memory throughput performance in Fig. 8c – with IAT, the memory bandwidth consumption can be reduced by at most 15.6%. Note that due to the limited capacity of LLC, IAT is not able to eliminate the memory traffic. It is desirable to combine IAT and a slightly smaller Rx buffer (*e.g.*, 512 in Fig. 3) to achieve even better memory traffic reduction with modest throughput loss. We also plot OVS performance in Fig. 8d. Again, with large packet sizes, IAT is able to improve OVS's IPC by ∼5% and reduce cycle per packet (CPP).

Meanwhile, IAT can still identify the core's demand for LLC capacity in a networking application. We demonstrate this with a second experiment with similar settings. The difference is, we fix the traffic at 64B line-rate (so that cores will be the dominant source of LLC miss). We start the line-rate traffic from a single flow, gradually increase the number of flows in the traffic, and report the performance in Fig. 9. To maintain the growing number of flows in its flow table, OVS requires more memory space. Hence, if we keep the static initial LLC allocation for OVS, it will suffer from a higher LLC miss count (and thus lower IPC) after more than 1k flows. On the other hand, IAT, by detecting the drop of IPC and the increase of LLC miss rate, is able to identify the demand for LLC of OVS's cores and allocate more LLC ways for them. As a result, OVS maintains a low LLC miss count and gains at most 11.4% higher IPC than the baseline. Note that with more flows, the IPC and CPP inevitably worsen since OVS's design [60] leads to more (slower) wildcarding lookups instead of pure (faster) exact match lookups.

We also repeat the experiments with three, four, and five containers and observe comparable performance improvement.

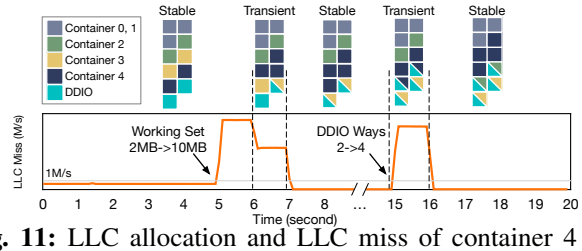


Fig. 11: LLC allocation and LLC miss of container 4 over time with IAT and 1.5KB packet size.

Solving the Latent Contender problem.[3] With the slicing model, we demonstrate that IAT can efficiently choose the LLC sharing policy between core and I/O. We have one VF for each NIC and bind them to two containers (0 and 1, marked as PC) running DPDK *test-pmd*. Each container runs on one dedicated core, and they share three dedicated LLC ways (no DDIO overlap). On each NIC, we generate single-flow line-rate traffic with different packet sizes. Besides, we have three identical containers (2 and 3 as BE, 4 as PC), each with one dedicated core and two dedicated LLC ways (no DDIO overlap) running X-Mem (random read pattern). In the beginning, all X-Mem containers have a working set of 2MB. At time *5s*, we increase the working set size of container 4 to 10MB (L2 cache size + 4 LLC ways size). Furthermore, at time *15s*, when the system has been stable, we manually increase LLC ways count for DDIO from two to four to see whether IAT can dynamically mitigate the DDIO overlapping interference. In addition to the baseline and IAT, we also test two cases: (1) Core-only, which means we only adjust the LLC allocation without I/O awareness[4]; (2) I/O-iso, based on Core-only, further excludes the DDIO's ways from core's LLC allocation. These are to emulate the behavior of other state-of-the-art LLC management mechanisms for comparison. We report the (stabilized) performance of X-Mem in container 4 (PC) in Fig. 10. After *5s*, when the working set size of container 4 increases dramatically, IAT starts allocating more LLC ways for container 4, which are shared with DDIO. To avoid contention between core and I/O, IAT shuffles the assigned LLC ways for container 4 and selects container 3 with BE workload to share LLC ways with DDIO. As seen from Fig. 10a, larger packet sizes will put higher pressure on the LLC ways for DDIO, interfering with the core more seriously and thus drag down the X-Mem's throughput. Core-only, by simply allocating two more "idle" (but actually shared with DDIO) LLC ways for X-Mem, performs well with small packet sizes but fails to maintain this trend with larger packet sizes since the core-I/O contention is mitigated but not eliminated. IAT is able to maintain constantly high throughput with all packet sizes (53.6%∼111.5% compared to baseline and 1.4%∼56.0% compared to Core-only) since it not only allows X-Mem to use more LLC space but also avoids core-I/O contention. Regarding latency (Fig. 10b), Core-only does not help since the randomly accessed data can be in the X-Mem-dedicated LLC ways, the two core-I/O shared LLC ways, or the memory. On average, the latency will not

[3]In this experiment, to isolate the problem, we temporarily disable IAT's functionality of changing LLC ways for DDIO.

[4]We do this by disabling I/O Demand state and LLC shuffling.

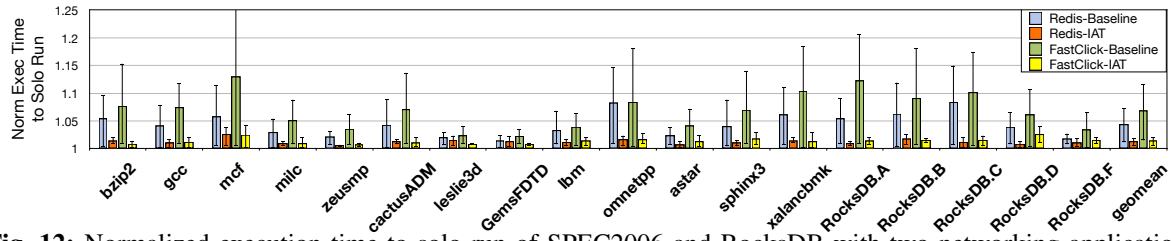


Fig. 12: Normalized execution time to solo run of SPEC2006 and RocksDB with two networking applications.

be much better than doing nothing (*i.e.*, baseline). However, IAT still maintains low latency (34.5%∼52.2% compared to baseline and 32.9%∼44.2% compared to Core-only) since it achieves 100% LLC isolation for container 4. In this phase, I/O-iso achieves performance similar to IAT, but in a different way: due to the limited available ways, it has to reduce the ways for BE container 2 and 3 to make room for the PC containers.

After *15s*, since DDIO shares (two) LLC ways with container 4 again, IAT detects the system's unstableness by the increasing LLC miss count of container 4's core and reshuffles the LLC ways allocation for as much LLC isolation as possible. Core-only, sharing all its four LLC ways with DDIO, suffers from more severe performance interference compared to it during *5s* and *15s*, which is especially significant when packet size is large. Both throughput (Fig. 10c) and latency (Fig. 10d) are very close to the baseline. It is worth noting that with small packet sizes, Core-only performs better in Fig. 10d than in Fig. 10b, which is because the LLC ways for DDIO in Fig. 10d are more than enough, and the inbound packet can be distributed to a larger space, which amortizes the contention. Regarding I/O-iso, it has even less LLC space ($11-4=7$ ways) for cores. Recalling Footnote 1, the PC containers have to share $7-2=5$ ways. Depending on the relative priority between container 4 and 0/1, container 4 can have 1∼3 ways (*i.e.*, the big range in Fig. 10c and Fig. 10d), leading to latency and throughput degradation anyway.

To show IAT's dynamic performance, we depict LLC allocation with IAT in 0∼20*s* and container 4's LLC miss (we have another independent `pqos` process to measure this every 0.1s) with 1.5K packet size in Fig. 11. We find IAT is able to react timely (within the timescale of sleep interval) to system/application phase changes, which are reflected in hardware metrics.

C. Application Results

We evaluate two sets of applications as the non-networking cloud workloads. (1) SPEC2006 benchmark suite [21]. We run selected memory-sensitive benchmarks [35], all with the `ref` input. (2) RocksDB [12], a persistent key-value store. We use YCSB [9] with 0.99 `Zipfian` distribution to test the performance of RocksDB. To avoid any storage I/O operations, we only load 10K records (1KB per record) so that all records are in RocksDB's memtable.

We choose in-memory key-value store (KVS) and network function virtualization (NFV) service chain as two representative networking workloads since they both involve tremendous network traffic and are cache-sensitive, which are the targeted usages for DDIO and IAT.

In-memory KVS. We use Redis [63], a popular in-memory KVS, to conduct the experiment. We run two Redis containers, each with two dedicated cores, and connect them to the OVS, which runs on another two dedicated cores. The OVS and two Redis containers share three LLC ways (no DDIO overlap). Besides, we have one PC container running either a SPEC2006 benchmark or RocksDB on one dedicated core and two LLC ways. We also have two BE containers, each with two LLC ways and one dedicated core, running X-Mem random-read, but with different working sets (one 1MB, one 10MB). In summary, nine cores are assigned. Initially, the LLC ways allocation of the three non-networking containers are *randomly shuffled*, and DDIO is not considered. The two NICs are connected to the OVS, and we run YCSB benchmarks from the traffic generator machines (each using eight threads). We pre-load 1M records with 1KB size and run different operations for testing.

NFV service chain. We run a FastClick [6]-based stateful service chain with and without IAT. This service chain consists of three network functions (NFs): a `classifier`-based firewall, an `AggregateIPFlows`-based flow stats, and a network address/-port translator (NAPT). Each of the two NICs is virtualized to two VFs with different VLAN tags. We have four identical containers, each bound to one VF (*i.e.*, each container processing one VLAN's traffic), running the service chain on four separate and dedicated cores. Such four containers share three LLC ways (no DDIO overlap). The non-networking workloads are the same as the KVS experiment. We generate traffic (all 1.5KB packets) of four VLANs from the two traffic generator machines with equal bandwidth, *i.e.*, 20Gbps per VLAN.

To isolate the performance impact caused by DDIO and highlight the two problems this work is committed to tackling, we temporarily disable IAT's functionality of assigning more/less LLC ways for tenants (but the ways for different tenants will still be shuffled). We first run each application solely (*i.e.*, *solo run*) to get purely isolated performance. We then co-run the applications in the scenarios above with and without IAT (*i.e.*, *baseline* and IAT). We run each case ten times and collect the performance degradation of each application.

We first report the execution time of each non-networking application normalized to solo run in Fig. 12. For SPEC2006, the working set size and cache sensitivity of each benchmark vary [35]. But in general, without DDIO awareness and IAT, we observe a 2.5% ∼ 14.8% performance degradation when co-running with Redis and 3.5% ∼ 24.9% when co-running with FastClick. Without DDIO awareness, a non-networking application is likely to be affected by the networking application

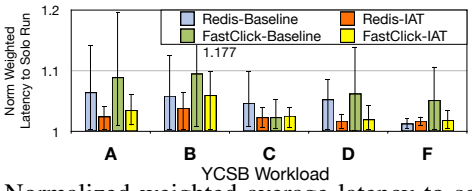
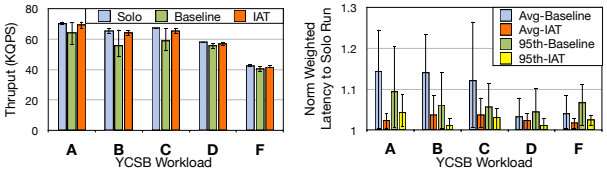


Fig. 13: Normalized weighted average latency to solo run of RocksDB with two networking applications.



(a) Throughput. **(b)** Latency.

Fig. 14: Redis performance on different YCSB workloads.

which seems to have complete isolation against it. In other words, if its LLC portion happens to overlap with DDIO (*e.g.*, baseline's maximum value in Fig. 12 and following figures), a significant degradation will appear; if there is no overlap (*e.g.*, baseline's minimum value in Fig. 12 and following figures), the impact is small – this incurs the big range of the baseline results (recall that LLC allocation was randomly shuffled at the beginning). However, IAT can effectively and stably maintain the performance isolation (at most 5.0% degradation). The reasons why it does not perfectly match the performance of solo run are: (1) partial LLC way overlap with DDIO may be inevitable when IAT assigns more LLC ways for DDIO (*e.g.*, High Keep state), and (2) the memory bandwidth consumed by the networking applications may also affect the performance of non-networking applications [57, 69]. Applying Intel® Memory Bandwidth Allocation (MBA) can solve this problem, which is out of the scope of this paper. Similarly, YCSB workloads for RocksDB have different cache locality requirements and thus are affected by networking applications to various extents (*i.e.*, 2.6%∼14.9% and 6.5%∼20.6%, respectively). Again, IAT shuffles the LLC ways of the non-networking application so that it is isolated from DDIO as much as possible, which leads to only 1.2%∼2.6% and 2.0%∼4.9% throughput degradation, respectively. Also note that, with more intensive network traffic (*i.e.*, line-rate for both inbound and outbound traffic), FastClick generally exerts more impact on the performance of non-networking applications than on Redis. We expect Redis to be impacted more severely when running more instances on a single server.

We also report the latency results of RocksDB in Fig. 13. Since there can be more than one type of operation in a single YCSB benchmark, we normalize each operation's latency and calculate the weighted average value (*i.e.*, normalized weighted latency). Since the key-value data of RocksDB can be evicted from the LLC to the main memory by the inbound DDIO data, the average latency performance can be much worse than solo run (*i.e.*, as high as 14.1% for Redis and 19.7% for FastClick). IAT can help mitigate such unexpected eviction by shuffling the LLC ways for the non-networking application, resulting in at most 6.4% and 9.9% longer latency, respectively.

We then discuss the performance of networking applications.

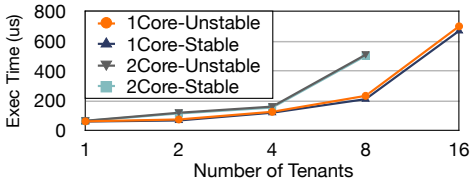


Fig. 15: IAT execution time with different tenant counts.

Fig. 14 depicts the YCSB results of Redis. In the baseline, since DDIO is not considered, if an application that heavily consumes cache resources (*e.g.*, X-Mem with 10MB working set, `mcf`, `omnetpp`, and `xalancbmk` in SPEC2006, and RocksDB) happens to be sharing LLC ways with DDIO, not only such non-networking application itself but also the networking applications will be adversely impacted. Specifically, we can see 7.1% ∼ 24.5% throughput degradation, 7.9% ∼ 26.5% longer average latency, and 10.1%∼20.4% longer tail latency among different YCSB workloads, especially with workloads that involve dense read operations (*i.e.*, A, B, and C). IAT mitigates such degradation by (1) allocating more LLC ways for DDIO to inject inbound packets into the LLC and (2) shuffling LLC ways to minimize, if not eliminate, the overlap between DDIO and cache-hungry applications. These two methods seem a little contradictory since more LLC ways for DDIO mean more chance to overlap with other applications. But actually, with more LLC ways for DDIO, inbound packets can be distributed evenly among LLC ways, amortizing pressure on each single LLC way. Even if a few ways are overlapped, the overall benefit still outperforms the adverse impact. As a result, IAT minimizes performance degradation to 2.8%∼5.6%, 2.9%∼8.9%, and 2.8%∼8.7%.

Regarding FastClick, since we are using large packets, the CPU core is not the bottleneck of packet processing, and we do not observe a meaningful throughput drop of the service chain. Also, due to the limitation of the software packet generator [40] we are using, we are not able to report the average and tail latency. However, we do see a lower maximum round-trip latency and fewer time variances (*i.e.*, a significant difference between the round-trip latency of two consecutive packets) with IAT, compared to baseline. This shows, allowing more packets to be fetched and processed from the LLC, IAT makes the FastClick service chain's performance more stable.

D. IAT *Overhead*

We set up different numbers of tenants and measure IAT daemon's execution time of each interval (excluding the initialization time and sleep time). We measure two cases: (1) each tenant has one dedicated core, and (2) each tenant has two dedicated cores. We run the IAT daemon on its dedicated core for 1000 seconds and report the average values. We classify the results into two categories, *i.e.*, Stable (only the `Poll Prof Data` time) and Unstable (`Poll Prof Data + State Transition + LLC Re-alloc` time), and depict them[5] in Fig. 15.

[5]Since our CPU only has 18 cores, we have at most eight tenants for the two cores per tenant case.

First, most of IAT's execution time is spent on the `Poll Prof Data` step, while conducting `State Transition` and `LLC Re-alloc` is relatively cheap. This is because, in the `Poll Prof Data` step, the daemon needs to read and write CPU hardware performance counters, each with costly context-switch. In contrast, the `State Transition` step is mainly branches and numerical comparison, and `LLC Re-alloc` typically only involves a couple of (*i.e.*, fewer than five) CPU register writes.

Second, IAT's execution time increases roughly sub-linearly with the number of cores it is monitoring. Monitoring more cores means more counter-read operations, which is the dominant part of the execution time. Also, with the same number of cores, fewer tenants correspond to a shorter time since the overhead of context-switch is alleviated. Even with a large number of cores, IAT's execution time does not exceed 800us, which shows the lightweightness and efficiency of IAT. That is, given the one-second interval in this paper, IAT, if co-running on a tenant's core, may only add negligible overhead (*i.e.*, at most 0.08%) to the system.

We also have an experiment that collocates IAT with an X-Mem container with $\sim$ 100% core utilization with configurations in Sec. III-B and see $\sim$0.5% impact on X-Mem throughput and practically no impact on X-Mem latency.

VII. Discussion

IAT's limitations. First, like most mechanisms based on modern Intel CPUs, LLC can only be partitioned in the way granularity in IAT, which is a commodity hardware limitation. And the low associativity (11-way) on Intel CPUs may lead to insufficiency (when allocating many tenants) and performance degradation [71]. One way to mitigate this (but loosen the strict isolation) is to group tenants and assign LLC ways [11].

Second, as a hard limitation of most dynamic LLC partitioning proposals, IAT is not able to handle microsecond-level traffic/workload changes [17, 33, 50]. This is mainly because (1) cache needs some time to take effect, and (2) short interval may cause fluctuation in sampling data. In other words, even in a single fine-grained routine, there can be steps with different characteristics, which is more transient. For example, when the interval is 1ms or shorter, some workloads' IPC value will fluctuate dramatically even if the monitored tenant is constantly doing a stable job without any interference. Also, the time of accessing CPU registers is not negligible at this timescale. In such cases, IAT should co-work with state-of-the-art CPU core scheduling mechanisms [17, 56] to maintain strict performance isolation.

Finally, IAT relies on some user-given tenant information to characterize workloads. However, we argue that this is not a concern since we target scenarios requiring extreme performance, which is more common in the private cloud than the public cloud[6]. In the private cloud, the basic tenant information should be transparent to the operator.

IAT in hardware. IAT can be implemented in the hardware. On the one hand, it may allow us to consider per-device DDIO statistics and adapt fine-grained DDIO based on those statistics at a faster rate without worrying about the performance overhead. On the other hand, a hardware implementation can enable DDIO to select LLC ways at runtime – it will allow DDIO to have more fine-grained (per-cacheline) control over the destination of data in the LLC and can detect/react to contention/congestion faster (microsecond-scale).

DDIO for the remote socket. Currently, DDIO only supports the local socket. That is, inbound data are only injected into the socket that the corresponding I/O device is attached to, even if the application is running on a remote socket [43]. One solution to overcome this constraint is the multi-socket NIC technology [55, 67], where the inbound data from the same NIC can be dispatched to different sockets. We also expect that DDIO will be extended to support remote sockets through socket interconnect (*i.e.*, Intel UPI).

Future DDIO consideration. The current DDIO implementation in Intel CPUs does not distinguish among devices and applications. That is, inbound traffic (both `write update` and `write allocate`) from various PCIe devices is treated the same. This, in turn, may cause performance interference between applications that use DDIO simultaneously. For example, a BE batch application (*e.g.*, Hadoop) with heavy inbound traffic may evict the data of other PC applications (*e.g.*, Redis, NFs) from DDIO's LLC ways, which leads to performance degradation of those PC applications. However, the batch applications, whose performance is insensitive to the memory access latency, cannot get significant benefit from fetching data from the LLC instead of the memory. We expect that DDIO in future Intel CPUs can be device-aware. *I.e.*, it can assign different LLC ways to different PCIe devices, or even different queues in a single device, just like what CAT does on CPU cores. Also, we expect that DDIO can be application-aware, meaning an application can choose whether or not to use DDIO entirely or partially. For instance, to avoid cache pollution, an application may enable DDIO only for packet header, while leaving the payload to the memory. IAT can further evolve to leverage such awareness.

VIII. Related Work

A. Cache Partitioning and Isolation

We have discussed prior research on cache partitioning with hardware/software techniques in previous sections. Most of them do not consider/leverage the I/O-related LLC during allocation. CacheDirector [13] proposes a means to better utilize DDIO feature by directing the most critical data to the core's local LLC slice, but it does not consider the performance interference and is applicable only if the per-core working set can fit into an LLC slice, which is unrealistic for many applications. IAT is complementary to these prior proposals and can co-work with them to provide a more comprehensive and robust cache QoS solution.

SLOMO [48] observes NFV performance contention caused by DDIO. However, it neither formulates the root causes nor proposes solutions. Farshin, Roozbeh, Maguire, and Kostic [14] conduct a performance analysis of DDIO in multiple scenarios

[6]In the public cloud, it is more economical to have a (relatively) looser resource allocation for tenants to avoid SLA violation.

and propose some optimization guidelines. However, they have inaccurate speculations of DDIO. For example, they speculate that data from I/O and core has different eviction priorities in different LLC ways (*e.g.*, "not bijective"), which is not true. Another example is, they describe that when disabling DDIO, LLC will be completely bypassed. But in fact, the data will still be in LLC (controller) first. We clarify the confusion of DDIO behavior in Sec. II-B and proposes a concrete and systematic solution for I/O-aware LLC management.

B. I/O Performance Partitioning

There are plenty of papers related to partitioning I/O for different applications/tenants, including I/O queuing and scheduling/throttling [2, 58], prioritizing/classifying applications [38, 50], and software-hardware co-partitioning [8]. While these solutions provide isolation from different levels (device, OS, application, *etc*.), none of them have investigated the I/O's interference to CPU's LLC, which inevitably leads to applications' performance drop in I/O-intensive scenarios. IAT provides the capability of identifying and alleviate such interference and thus can work with those I/O partitioning techniques together.

IX. Conclusion

In modern cloud servers with Intel® DDIO technology, I/O has become an important factor that affects CPU's LLC performance and utilization. In this paper, we first summarized two problems caused by DDIO, and then proposed IAT, the first I/O-aware mechanism for LLC management, which allocates LLC ways for not only the core but also the I/O. Our experiments showed that IAT is able to effectively reduce the performance interference caused by DDIO between applications. We hope this paper can attract more attention to the study of I/O-aware LLC management in the architecture and system communities.

Acknowledgment

We would like to thank Robert Blankenship, Miao Cai, Haiyang Ding, Andrew Herdrich, Ravi Iyer, Yen-Cheng Liu, Radhika Mittal, Amy Ousterhout, Anil Vasudevan, as well as the anonymous reviewers for their insightful and helpful information and feedback. This research is supported by National Science Foundation (No. CNS-1705047) and Intel Corporation's Academic research funding.

References

[1] M. Alian, Y. Yuan, J. Zhang, R. Wang, M. Jung, and N. S. Kim, "Data direct I/O characterization for future I/O system exploration," in *Proceedings of the 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'20)*, Virtual Event, Aug. 2020.

[2] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska, "End-to-end performance isolation through virtual datacenters," in *Proceedings of the 11st USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, Oct. 2014.

[3] Ansyun Inc., "DPDK native accelerated network stack," https://www.ansyun.com, accessed in 2021.

[4] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, Apr. 2018.

[5] arm, "Arm DynamIQ shared unit technical reference manual," https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/cache-stashing, accessed in 2021.

[6] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15)*, Oakland, CA, May 2015.

[7] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proceedinsg of the 41st IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, Lake Como, Italy, Nov. 2008.

[8] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, Providence, RI, Apr. 2019.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, Indianapolis, IN, Jun. 2010.

[10] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, 2012.

[11] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*, Vienna, Austria, Feb. 2018.

[12] Facebook, "RocksDB: A persistent key-value store for fast storage environments," https://rocksdb.org, accessed in 2021.

[13] A. Farshin, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, "Make the most out of last level cache in Intel processors," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*, Dresden, Germany, Mar. 2019.

[14] ——, "Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks," in *Proceedings of 2020 USENIX Annual Technical Conference (ATC'20)*, Virtual Event, Jul. 2020.

[15] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, Apr. 2017.

[16] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, L. Jack, L. Norman, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, M. Silva, Ganriel nd Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, Apr. 2018.

[17] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *Proceedigs of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, Nov. 2020.

[18] H. Golestani, A. Mirhosseini, and T. F. Wenisch, "Software data planes: You can't always spin to win," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC'19)*, Santa Cruz, CA, Nov. 2019.

[19] M. Gottscho, S. Govindan, B. Sharma, M. Shoaib, and P. Gupta, "X-Mem: A cross-platform and extensible memory

characterization tool for the cloud," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'16)*, Uppsala, Sweden, Jun. 2016.

[20] J. Han, S. Jeon, Y.-r. Choi, and J. Huh, "Interference management for distributed parallel applications in consolidated clusters," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, GA, Apr. 2016.

[21] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

[22] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*, Barcelona, Spain, Mar. 2016.

[23] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network I/O," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*, Madison, WI, Jun. 2005.

[24] J. Hwang, K. K. Ramakrishnan, and T. W. and, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, Seattle, WA, Apr. 2014.

[25] Intel Corporation, "Data plane development kit (DPDK)," https://www.dpdk.org, accessed in 2021.

[26] ——, "Intel 64 and IA-32 architectures software developer's manual volume 4: Model-specific registers," https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-4-model-specific-registers, accessed in 2021.

[27] ——, "Intel Ethernet converged network adapter XL710 10/40 GbE," https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-xl710-brief.html, accessed in 2021.

[28] ——, "Intel resource director technology (Intel RDT)," https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html, accessed in 2021.

[29] ——, "Intel Xeon Gold 6140 processor," https://ark.intel.com/content/www/us/en/ark/products/120485/intel-xeon-gold-6140-processor-24-75m-cache-2-30-ghz.html, accessed in 2021.

[30] ——, "Intel Xeon processor Scalable family uncore reference manual," https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-uncore-performance-monitoring-manual.html, accessed in 2021.

[31] ——, "Intel® data direct I/O (DDIO)," https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, accessed in 2021.

[32] ——, "User space software for Intel resource director technology," https://github.com/intel/intel-cmt-cat, accessed in 2021.

[33] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang, "Perfiso: Performance isolation for commercial latency-sensitive services," in *Proceedings of 2018 USENIX Annual Technical Conference (ATC'18)*, Boston, MA, Jul. 2018.

[34] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in intel processors," in *Proceedings of the 2015 Euromicro Conference on Digital System Design (DSD'15)*, Funchal, Madeira, Portugal, Aug. 2015.

[35] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf*, 2010.

[36] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*, Virtual Event, Oct. 2020.

[37] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, Nov. 2012.

[38] G. Kappes and S. V. Anastasiadis, "A user-level toolkit for storage I/O isolation on multitenant hosts," in *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*, Virtual Event, Oct. 2020.

[39] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, Salt Lake City, UT, Mar. 2014.

[40] Keith Wiles, "Pktgen - traffic generator powered by DPDK," https://github.com/pktgen/Pktgen-DPDK, accessed in 2021.

[41] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, San Jose, CA, Oct. 2002.

[42] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-driven resource management for interactive services on reconfigurable multicores," in *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*, Virtual Event, Oct. 2020.

[43] I. Kurakin and R. Khatko, "IO issues: Remote socket accesses," https://software.intel.com/en-us/vtune-cookbook-io-issues-remote-socket-accesses, accessed in 2021.

[44] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland'20)*, Virtual Event, May 2020.

[45] D. Kwon, J. Boo, D. Kim, and J. Kim, "FVM: FPGA-assisted virtual device emulation for fast, scalable, and flexible storage virtualization," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, Nov. 2020.

[46] Q. Llull, S. Fan, S. M. Zahedi, and B. C. Lee, "Cooper: Task colocation with cooperative games," in *Proceedings of 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, Austin, TX, Feb. 2017.

[47] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is lighter (and safer) than your container," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, Oct. 2017.

[48] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[49] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, Dec. 2011.

[50] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: A microkernel approach to

host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Huntsville, Canada, Oct. 2019.

[51] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*, Kyoto, Japan, Nov. 2015.

[52] NETRONOME, "Agilio OVS software," https://www.netronome.com/products/agilio-software/agilio-ovs-software/, accessed in 2021.

[53] Network Working Group, "Benchmarking methodology for network interconnect devices," https://tools.ietf.org/html/rfc2544, accessed in 2021.

[54] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*, Budapest, Hungary, Aug. 2018.

[55] Nvidia, "Socket direct adapters," https://www.nvidia.com/en-us/networking/ethernet/socket-direct/, accessed in 2021.

[56] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, Boston, MA, Feb. 2019.

[57] J. Park, S. Park, and W. Baek, "CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*, Dresden, Germany, Mar. 2019.

[58] T. Patel, R. Garg, and D. Tiwari, "GIFT: A coupon based throttle-and-reward mechanism for fair and efficient I/O bandwidth management on parallel storage systems," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, Santa Clara, CA, Feb. 2020.

[59] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *Proceedings of 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, San Diego, CA, Feb. 2020.

[60] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Okaland, CA, May 2015.

[61] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, Nov. 2020.

[62] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Orlando, FL, Dec. 2006.

[63] redislabs, "Redis," https://redis.io, accessed in 2021.

[64] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, Jul. 2008.

[65] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Modeling performance variation due to cache sharing," in *Proceedings of the 2013 IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*, Shenzhen, China, Feb. 2013.

[66] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with Intel's cache allocation technology," in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17)*, Portland, OR, Nov. 2017.

[67] I. Smolyar, A. Markuze, B. Pismenny, H. Eran, G. Zellweger, A. Bolen, L. Liss, A. Morrison, and D. Tsafrir, "IOctopus: Outsmarting nonuniform DMA," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Virtual Event, Mar. 2020.

[68] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, Virtual Event, Apr. 2020.

[69] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling SLOs in network function virtualization," in *Proceedings of 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, Apr. 2018.

[70] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: Accelerating resource allocation in virtualized environments," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, London, England, UK, Mar. 2012.

[71] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, Cambridge, UK, Dec. 2014.

[72] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support," in *Proceedings of 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, Austin, TX, Feb. 2017.

[73] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: Dynamic cache allocation with partial sharing," in *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, Porto, Portugal, Apr. 2018.

[74] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dCat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, Porto, Portugal, Apr. 2018.

[75] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A development kit to build high performance storage applications," in *Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'17)*, Hong Kong, China, Dec. 2017.

[76] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *Proceedings of the 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT'14)*, Edmonton, Canada, Aug. 2014.

[77] Y. Zhang, J. Chen, X. Jiang, Q. Liu, I. M. Steriner, A. J. Herdrich, K. Shu, R. Das, L. Cui, and L. Jiang, "LIBRA: Clearing the cloud through dynamic memory bandwidth management," in *Proceedings of the 27th IEEE International Symposium on High-performance Computer Architecture (HPCA'21)*, Virtual Event, Feb. 2021.

[78] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Atlanta, GA, Apr. 2016.