Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks

Alireza Farshin, *KTH Royal Institute of Technology;* Amir Roozbeh, *KTH Royal Institute of Technology and Ericsson Research;* Gerald Q. Maguire Jr. and Dejan Kostić, *KTH Royal Institute of Technology*

https://www.usenix.org/conference/atc20/presentation/farshin

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks

Alireza Farshin[*†]
KTH Royal Institute of Technology

Amir Roozbeh[*]
KTH Royal Institute of Technology
Ericsson Research

Gerald Q. Maguire Jr.
KTH Royal Institute of Technology

Dejan Kostić
KTH Royal Institute of Technology

Abstract

Memory access is the major bottleneck in realizing multi-hundred-gigabit networks with commodity hardware, hence it is essential to make good use of cache memory that is a faster, but smaller memory closer to the processor. Our goal is to study the impact of cache management on the performance of I/O intensive applications. Specifically, this paper looks at one of the bottlenecks in packet processing, i.e., direct cache access (DCA). We systematically studied the current implementation of DCA in Intel® processors, particularly Data Direct I/O technology (DDIO), which directly transfers data between I/O devices and the processor's cache. Our empirical study enables system designers/developers to optimize DDIO-enabled systems for I/O intensive applications. We demonstrate that optimizing DDIO could reduce the latency of I/O intensive network functions running at 100 Gbps by up to ~30%. Moreover, we show that DDIO causes a 30% *increase* in tail latencies when processing packets at 200 Gbps, hence it is crucial to selectively inject data into the cache or to explicitly bypass it.

1 Introduction

While the computer architecture community continues to focus on hardware specialization, the networking community tries to achieve greater flexibility with Software-defined Networking (SDN) together with Network Function Virtualization (NFV) by moving from specialized hardware toward commodity hardware. However, greater flexibility comes at the price of lower performance compared to specialized hardware. This approach has become more complex due to the end of Moore's law and Dennard scaling [14]. Furthermore, commercially available 100-Gbps networking interfaces have revealed many challenges for commodity hardware to support packet processing at multi-hundred-gigabit rates. More specifically, the interarrival time of small packets is shrinking to a few nanoseconds (i.e., less than Last Level Cache (LLC) latency). Consequently, any costly computation prevents commodity hardware from processing packets at these rates, thereby causing a tremendous amount of buffering and/or packet loss. As accessing main memory is impossible at these line rates, it is essential to take *greater* advantage of the processor's cache [81]. Processor vendors (e.g., Intel®) introduced new monitoring/controlling capabilities in the processor's cache, e.g., Cache Allocation Technology (CAT) [59]. In alignment with the desire for better cache management, this paper studies the current implementation of Direct Cache Access (DCA) in Intel processors, i.e., Data Direct I/O technology (DDIO), which facilitates the direct communication between the network interface card (NIC) and the processor's cache while avoiding transferring packets to main memory. Our goal is to complete the recent set of studies focusing on understanding the leading technologies for fast networking, i.e., Peripheral Component Interconnect express (PCIe) [58] and Remote Direct Memory Access (RDMA) [37]. We believe that understanding & optimizing DDIO is the missing piece of the puzzle to realize high-performance I/O intensive applications. In this regard, we empirically reverse-engineer DDIO's implementation details, evaluate its effectiveness at 100/200 Gbps, discuss its shortcomings, and propose a set of optimization guidelines to realize performance isolation & achieve better performance for multi-hundred-gigabit rates. Moreover, we exploit a little-discussed feature of Xeon® processors to demonstrate that *fine-tuning* DDIO could improve the performance of I/O intensive applications by up to ~30%. To the best of our knowledge, we are the first to: (i) systematically study and reveal details of DDIO and (ii) take advantage of this knowledge to process packets more efficiently at 200 Gbps.

Why DCA matters? Meeting strict Service Level Objectives (SLO) and offering bounded latency for Internet services is becoming one of the critical challenges of data centers while operating on commodity hardware [54]. Consequently, it is essential to identify the sources of performance variability in commodity hardware and *tame* them [51]. In computer

[*]Both authors contributed equally to the paper.
[†]This author has made all open-source contributions.

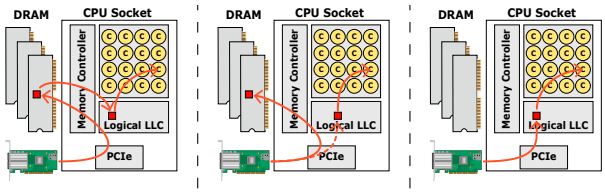
systems, one of these sources of variability is the cache hierarchy, which can introduce uncertainty in service times, especially in tail latencies. Additionally, the advent of modern network equipment [82] enables applications to push costly calculations closer to the network while keeping & performing only stateful functions at the processors [36, 38], thereby making modern network applications ever *more* I/O intensive. Hence, taming the performance variability imposed by the cache, especially for I/O, is now more crucial than before. Moreover, as CPU core count goes up, it is important to be able to deliver appropriate I/O bandwidth to them. Therefore, we *go one level deeper* [61] to investigate the impact of I/O cache management, done by DCA, on the performance of multi-hundred-gigabit networks.

Contributions. In this paper, we:

1. Design a set of micro-benchmarks to reveal little-known details of DDIO's implementation* (§4),
2. Extensively study the characteristics of DDIO in different scenarios and identify its shortcomings* (§5),
3. Show the importance of balancing load among cores and tuning DDIO capacity when scaling up (§6),
4. Measure the sensitivity of multiple applications (i.e., Memcached, NVMe benchmarks, NFV service chains) to DDIO (§7),
5. Demonstrate the necessity and benefits of bypassing cache while receiving packets at 200 Gbps (§8),
6. Discuss the lessons learned from our study that are essential for optimizing DDIO-enabled systems receiving traffic at multi-hundred-gigabit rates (§9).

2 Direct Cache Access (DCA)

A standard method to transfer data from an I/O device to a processor is Direct Memory Access (DMA). In this mechanism, a processor, typically instructed by software, provides a set of memory addresses, aka receive (RX) descriptors, to the I/O device. Later, the I/O device directly reads/writes data from/to main memory without involving the processor. For inbound traffic, the processor can be informed about newly DMA-ed data either by receiving an interrupt or polling the I/O device. Next, the processor fetches the I/O data from main memory to its cache in order to process the data. For outbound traffic, the processor informs the I/O device (via transmit (TX) descriptors) of data that is ready to be DMA-ed from main memory to the device. The main source or destination of traditional DMA transfers is main memory, see Fig. 1a. However, the data actually needs to be loaded into the processor's cache for processing. Therefore, this method is inefficient and costly in terms of (i) number of accesses to main memory [43] (i.e., $2n+5$ for n cache lines [43]), (ii) access latency to the I/O data, and (iii) memory bandwidth usage. Moreover, the negative impact of these inefficiencies becomes increasingly severe with higher link speeds. For instance, a server has 6.72 ns to process small packets at 100 Gbps, whereas every access to main memory takes ~100 ns, 15× more expensive. Therefore, placing the I/O data directly in the processor's cache rather than in main memory is desirable. The advent of faster I/O technologies motivated researchers to introduce Direct Cache Access (DCA) [25, 42, 43]. DCA exploits PCIe Transaction Layer Packet Processing Hint [30], making it possible to prefetch portions of I/O data to the processor's cache, see Fig. 1b. Potentially, this overcomes the drawbacks of traditional DMA, thereby achieving maximal I/O bandwidth and reducing processor stall time. Although this way of realizing DCA can effectively prefetch the desired portions of I/O data (e.g., descriptors and packet header), it is still inefficient in terms of memory bandwidth usage since the whole packet is DMA-ed into main memory. Additionally, this requires operating system (OS) intervention and support from the I/O device, system chipset, *and* processor [1]. To address these limitations and avoid ping-ponging data between main memory & the processor's cache, Intel rearchitected the prefetch hint-based DCA, introducing Data Direct I/O technology (DDIO) [28].



(a) Trad. DMA. (b) DCA. (c) DDIO.

Figure 1: Different approaches of DMA for transferring data from an I/O device (e.g., NIC). Red arrows show the path that a packet traverses *before* reaching the processing core.

3 Data Direct I/O Technology (DDIO)

Intel introduced DDIO technology with the Xeon E5 family. With DDIO, I/O devices perform DMA directly to/from Last Level Cache (LLC) rather than system memory, see Fig. 1c. DDIO is also known as write-allocate-write-update-capable DCA (wauDCA) [45], as it uses this policy to update cache lines in an n-way set associative LLC, where n cache lines form one set. For packet processing applications, NICs can send/receive both RX/TX descriptors and the packets themselves via the LLC, thereby improving applications' response time & throughput†. DDIO works as follows [41]:

Writing packets. When a NIC writes a cache line to LLC via PCIe, DDIO overwrites the cache line if it is already present in *any* LLC way (aka a PCIe write hit or *write update*). Otherwise, the cache line is allocated in the LLC and DDIO writes the data into the newly allocated cache line (aka a PCIe write miss or *write allocate*). In the latter case, DDIO is restricted to use only a limited portion of LLC when allocating

*The source code is available at: https://github.com/aliireza/ddio-bench

†We will use the terms I/O device and NIC interchangeably.

cache lines. It is possible to *artificially* increase this portion by warming up the cache with processor writes to the address of these buffers, then DDIO performs write-updates [16].

Reading packets. A NIC can read a cache line from LLC if the cache line is present in *any* LLC way (aka a PCIe read hit). Otherwise, the NIC reads a cache-line-sized chunk from system memory (aka a PCIe read miss).

To monitor DDIO and its interaction with I/O devices, Intel added uncore performance counters to its processors [29]. The Intel Performance Counter Monitor (PCM) tool (e.g., `pcm-pcie.x`*) [86] can count the number of PCIe write hits/misses (represented as an ItoM event) and PCIe read hits/misses (represented as a PCIeRdCur event). Next, we discuss the inherent problem of DDIO, which makes it hard to achieve low-latency for multi-hundred-gigabit NICs.

3.1 How can DDIO become a Bottleneck?

Researchers have shown some scenarios in which DDIO cannot provide the expected benefits [11, 41, 50, 83]. Two typical cases occur when new incoming packets repeatedly evict the previously DMA-ed packets (i.e., *not-yet-processed* and *already-processed packets*) in the LLC. Consequently, the processor has to load not-yet-processed packets from main memory rather than LLC and the NIC needs to DMA the already-processed packets from the main memory, thereby missing the benefits of DDIO. Tootoonchian et al. referred to this problem as *the leaky DMA problem* [83]. To mitigate this problem, they proposed reducing the number of "in-flight" buffers (i.e., descriptors) such that all incoming packets fit in the limited portion of LLC used for I/O. Thus, performance isolation can be done using *only* CAT (i.e., cache partitioning). Unfortunately, reducing the number of RX descriptors is only a temporary solution due to increasing link speeds. Multi-hundred-gigabit NICs introduce new challenges, specifically:

① **Packet loss.** At sub-hundred-gigabit link speeds reducing the number of RX descriptors may not result in a high packet loss rate, but at $\geq$100 Gbps packet loss increases due to the tight processing time budget before buffering/queuing happens. For instance, every extra ~7 ns spent stalling or processing/accessing a packet causes another packet to be buffered when receiving 64-B packets at 100 Gbps. When there are insufficient resources for immediate processing, increasing the number of RX descriptors permits packets to be buffered rather than dropped. Delays in processing might occur because of interrupt handling, prolonged processing, or a sudden increase in the packet arrival rate [17]; therefore, multi-hundred-gigabit networks cannot avoid packet loss without having a *sufficiently large* number of descriptors. Increasing the number of processing cores can reduce the packet loss rate, but applications that are compute- or memory-intensive require many cores to operate at the speed of the underlying hardware, e.g., Thomas et al. [81] mention that a server performing *one* DRAM access per packet needs 79 cores to process packets at 400 Gbps.

② **TX buffering.** One of the scenarios that makes DDIO inefficient is the eviction of already-processed packets. Reducing the number of RX descriptors may solve this problem for systems that require a small number of TX descriptors, but this is not the case for 100-Gbps NICs. Unfortunately, the *de facto* medium for DMA-ing packets (i.e., PCIe 3.0) induces some transmission limitations [58]. Consequently, packets often need to be buffered in the computer system for some time before being DMA-ed to the NIC. This buffering can be realized by either a software queue or increasing the number of TX descriptors [35]. Unfortunately, either of these alternatives increases the probability of eviction of already-processed packets. Therefore, completely solving the leaky DMA problem requires fine-tuning *both* the size of the software queue and the number of RX & TX descriptors.

③ **PAUSE frames.** To alleviate packet loss, one can use Ethernet flow control mechanisms (e.g., PAUSE frames) that cause packets to be buffered earlier in the network, i.e., PAUSE frames stop the previous network node from transmitting packets for a short period. However, these mechanisms are costly in terms of latency, making them less desirable than packet loss for time-critical applications. The minimum and maximum pause duration of a 100-Gbps interface are 5.12 ns and 335.5 µs [56]. Our measurements show that a core that is simply forwarding packets at 100 Gbps with 1024 RX & TX descriptors causes the NIC to send ~179 k PAUSE frames while receiving ~80 M packets.

Dynamic reduction. As reducing the number of RX buffers cannot *fully* solve the problem and it shifts the problem to another part of the network, most probably the previous node; therefore, an alternative is to *dynamically* reduce the pressure on the LLC when the number of I/O caused cache evictions starts to increase†. These cache evictions can be tracked by monitoring either PCIe events or the length of the software queue. After detecting a problem, the processor should fetch a smaller number of packets from the NIC (i.e., reducing the RX burst size). Thus, the processor passes fewer free buffers to the NIC, reducing the number of DMA transactions. Unfortunately, this approach does not perform well, hence we need a *proactive* solution, not a reactive one.

Is it sufficient to scale up? Due to the demise of the Dennard scaling [14], processors are now shipped with more cores rather than higher clock frequencies. Moreover, the per-core cache quota (i.e., LLC slices) has decreased in recent Xeon processors, i.e., the size of LLC slices reduced from 2.5 MiB to 1.375 MiB in the Xeon scalable family (i.e., Skylake) [55]. This reduction in per-core cache size directly affects the optimal number of descriptors as these are proportional to the limited space for DDIO. For instance, using 18 cores, each having 256 RX descriptors, requires ~6.5 MiB, which is equal

*The description of events can be found in [27] and pp. 63-66 of [41].

†Our implementation is available at: https://github.com/tbarbette/fastclick/tree/DMAdynamic

to ~26.6% of the LLC in this processor and greater than the available DDIO capacity (see §4.1).

Our approach. To overcome these challenges, it is necessary to study and analyze DDIO empirically in order to *make the best use of it*. A better understanding of DDIO and its implementation can help us optimize current computer systems and enables us to propose a better DCA design for future computer systems that could accommodate the ever-increasing NIC link speeds. For instance, Fig. 2 demonstrates that tuning DDIO's capacity makes it possible to achieve a suitable performance while using a large number of descriptors (our approach), as opposed to using a limited number of descriptors (ResQ's approach proposed by Tootoonchian et al. [83]).

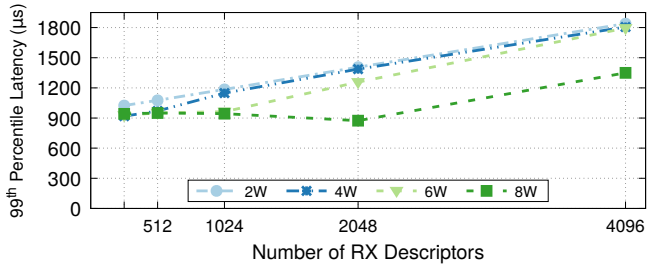


Figure 2: Using more DDIO ways ("W") enables 2 cores to forward 1500-B packets at 100 Gbps with a larger number of descriptors while achieving better or similar tail latency.

4 Understanding Details of DDIO

This section discusses four questions: ① What part of LLC is used for I/O? ② How does I/O interact with other applications? ③ Does DMA via remote sockets pollute LLC? and ④ Is it possible to disable/tune DDIO?

Testbed. We use a testbed with the configuration shown in Table 1 running Ubuntu 18.04.2 (Linux kernel-4.15.0-54). We use the Skylake server unless stated otherwise. FastClick [9] is used to generate & process packets. Additionally, we use a campus trace as a real workload (with mixed-size packets) and generate synthetic traces (with fixed-size packets). For our multicore experiment, we use RSS [24] to distribute packets among different queues (one queue per core), unless stated otherwise. Furthermore, we isolate the one CPU socket on which we run the experiment to increase the accuracy of the measurements. PAUSE frames are disabled to avoid taking into account pause duration in the end-to-end latency. In all experiments, the NIC driver sets the appropriate number of TX descriptors based on the number of TX queues, and to avoid extra looping at the transmitting side FastClick buffers up to 1024 packets. We use the Network Performance Framework (NPF) tool [57] to run the experiments.

4.1 Occupancy

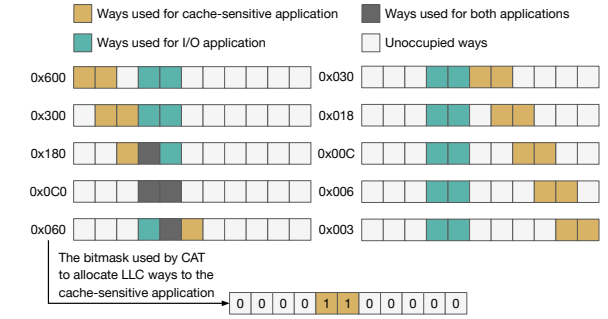
Initially, Intel announced that DDIO only uses 10% of LLC [28] and did not mention what *part* of the LLC is used (i.e., ways, sets, or slices [15]). Recent Intel technical reports mention that DDIO only uses a subset of LLC ways, by default two ways [41, 72]. However, it is still unclear whether this "subset" is fixed or whether it can be dynamically selected using a variant of Least Recently Used (LRU) policies [33, 34, 65, 87]. Knowledge of these details could avoid I/O contention and optimize performance isolation [83] by performing precise cache management/partitioning [13, 62] (e.g., way partitioning with CAT [59]). This issue becomes increasingly critical for newer generations of Xeon processors that have lower LLC set-associativity (e.g., 11 ways in some Skylake processors, as opposed to 20 ways in Haswell processors), thereby using a larger portion ($\frac{2}{11} \approx 18\%$) of the LLC for I/O. Lower set-associativity makes the cache less flexible when the LLC is divided into multiple partitions, each of which could be used to accommodate different applications' code & data. To clarify this, we assumed that the ways that are used for DDIO are *fixed* and then try to confirm this with an experiment in which we co-run an I/O and a cache-sensitive application. To increase the pressure on the LLC by DMA-ing more cache lines, we used an L2 forwarding DPDK-based application as the I/O intensive application. Specifically, it receives large packets (1024-B) at a high rate (~82 Gbps) using a large number of RX descriptors (4096 RX descriptors). For the cache-sensitive application, we chose `water_nsquared` from the Splash-3 benchmark suite [62, 66, 69] since it performs a large number of LLC accesses; hence, it interferes with the I/O application.

Each application is run on a different core and CAT is used to allocate different cache ways to each core. We allocate two *fixed* ways to the I/O application and two *variable* ways to the cache-sensitive application. To avoid memory bandwidth contention, we also used Memory Bandwidth Allocation (MBA) technology [21] to limit the memory bandwidth of each core to 40%. Fig. 3a shows the CAT configuration used in the experiment. We start by allocating the two leftmost ways (i.e., bitmask of 0x600) to the cache-sensitive application and then we keep shifting the allocated ways one

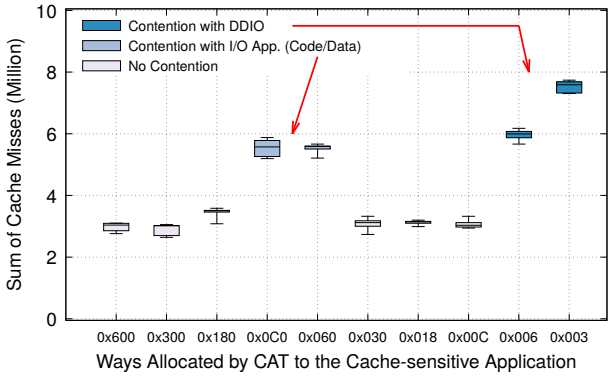
Table 1: Details of our testbed. In each case, the NIC is a Mellanox ConnectX-5 VPI.

Configuration / Machine	Intel Xeon Processor			Memory	Last Level Cache (LLC)	
	Model	Frequency	#Cores		Size	Associativity
Packet generator (Skylake)	Gold 6134	3.2 GHz	8	512 GiB	18×1.375 MiB	11
Server (Skylake)	Gold 6140	2.3 GHz	18	256 GiB	18×1.375 MiB	11
Server (Haswell)	E5-2667 v3	3.2 GHz	8	128 GiB	8×2.5 MiB	20

to the right until we cover all the LLC ways while measuring the LLC misses of the I/O application. Fig. 3b shows the results of this experiment. These results demonstrate that the cache-sensitive application interferes with the I/O application in *two* regions. The first (see 0x0C0 in Fig. 3b) occurs when the cache-sensitive application uses the same ways as the I/O application, due to the code/data interference of the two applications. However, the second (see 0x003 in Fig. 3b) cannot be explained with this same argument since the I/O application is limited to using other ways (i.e., 0x0C0). Furthermore, since the CPU socket is isolated, no other application can cause cache misses. CAT only mitigates the contention induced by code/data not DDIO. Therefore, we conclude that the second interference is most probably due to I/O, which means DDIO uses the two rightmost ways in LLC (i.e., bitmask of 0x003). The interference is proportional to the number of received packets per second $\times$ average packet size. We expected to see roughly the *same* amount of cache misses for bitmasks of 0x180 and 0x060, as they are completely symmetrical in terms of way occupancy. However, the undocumented LRU policy of the CPU may affect how the application uses the cache ways.



(a) CAT configuration.



(b) Sum of cache misses for the I/O application.

Figure 3: Interference of an I/O and a cache-sensitive application using the `parsec_native` configuration (to cause a high rate of cache misses) when the cache-sensitive application uses different LLC ways. The rise in the rightmost side shows the contention with DDIO ways.

4.2 I/O Contention

One of the established mechanisms to ensure performance isolation and mitigate cache contention is CAT, which limits different applications to a subset of LLC ways. However, §4.1 showed that DDIO uses two *fixed* LLC ways. Therefore, isolating applications using CAT may not *fully* ensure performance isolation, due to cache contention caused by I/O. Such contention may occur in two common scenarios:

① **I/O vs. Code/Data.** When an application is limited to using those ways which are also used by DDIO, then cache lines allocated in LLC for DDIO may evict the code/data of any application (i.e., either I/O or non-I/O application). This issue was discussed by Tootoonchian et al. [83]. Their proposed framework, ResQ, uses only 90% of LLC to avoid interfering with DDIO's reserved space, but does not mention which part of LLC is isolated. §4.1 showed the destructive (i.e., ~2.5×) impact on the cache misses of the I/O application due to a cache-hungry application overlapping with DDIO, see the rise in cache misses at the right side of Fig. 3b. However, it did not show the impact of contention on the cache-hungry application; therefore, we repeated the experiment and measured the cache misses of the cache-sensitive application while using a lighter configuration. Fig. 4 illustrates that the cache misses of the cache-sensitive application were similarly adversely affected. Therefore, overlapping any application with DDIO ways in LLC can reduce the performance of *both* applications. To tackle this, one can isolate the I/O portion of LLC (e.g., the two ways used for DDIO) by using CAT so that applications share the LLC *without* overlapping with I/O. Comparing Fig. 3b and 4, we see that an unexpected rise (almost 3×) in cache misses occurs in a different region (i.e., bitmask of 0x600 in Fig. 4 as opposed to bitmask of 0x003 in Fig. 3b) when I/O is evicting code/data. Hence, we speculate that CAT does not use a *bijective* function to map I/O & code/data to ways, thus $f : \text{code/data} \rightarrow \text{Ways}$ is not equivalent to $g : \text{I/O} \rightarrow \text{Ways}$. Specifically, I/O evicts code/data when the latter is located in the two leftmost ways whereas code/data evicts I/O when the latter is using the two rightmost ways. Such information is useful to know, as it will give us an understanding of the eviction policy and the default priority of code/data and I/O.

② **I/O vs. I/O.** When multiple I/O applications are isolated from each other with CAT, they could still *unintentionally* compete for the *fixed* ways allocated to DDIO. §8.1 elaborates the negative impact of this type of contention.

Security implication. Since DDIO uses two *fixed* ways in LLC, it is possible to extend microarchitectural attacks to extract useful information from I/O data (e.g., NetCAT [44] and Packet Chasing [76, 77]). Furthermore, I/O applications can be vulnerable to performance attacks.

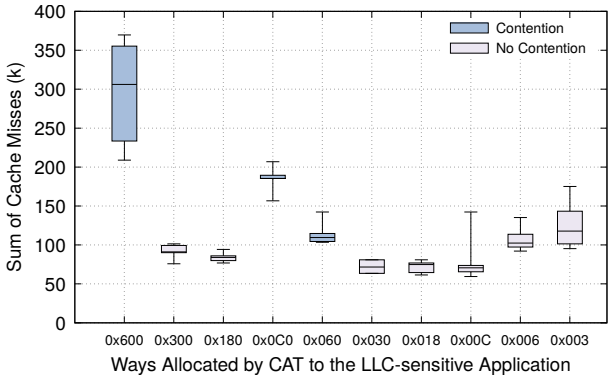


Figure 4: Interference of the cache-sensitive and the I/O applications. Y axis shows the sum of cache misses of the cache-sensitive application. The cache-sensitive application uses a lighter configuration (i.e., `ddio_sim`), which causes fewer cache misses than the I/O application.

4.3 DMA via Remote Socket

According to Intel [16, 32], the current implementation of DDIO only affects the local socket. Consequently, if a core accesses I/O data from an I/O device connected to a remote socket, the data has to traverse the inter-core interconnect, i.e., Intel QuickPath Interconnect (QPI) or Intel Ultra path Interconnect (UPI). It was uncertain whether data traversing the inter-core interconnect is loaded into the LLC of the remote socket or not. We clarified this by running the same experiment discussed in §4.2 while the NIC is connected to a remote socket. The result (removed for brevity) showed that cache misses of neither application were affected by the *I/O* cache lines, hence packets coming through the UPI links *do not end up in the local LLC*. Additionally, the cache misses of the I/O application dramatically increased to 20× greater than when receiving packets via the local socket without any contention. Thus, DDIO is ineffective for the remote socket *and* it pollutes the LLC on the socket connected to the NIC.

4.4 Tuning Occupancy and Disabling DDIO

Although [20, 72] mention that DDIO uses two ways by *default*, there is no mention of whether it is *possible* to increase or decrease the number of ways used by DDIO. A little-discussed Model Specific Register (MSR) called "IIO LLC WAYS" with the address of 0xC8B* is discussed in a few online resources [64, 79] and server manuals [73, 74]. For Skylake, the default value of this register is equal to 0x600 (i.e., two bits set). While these bits cannot be unset, it is possible to set additional bits and the maximum value for this register on our CPU is 0x7FF (i.e., 11 bits set: the same as the number of LLC ways). New values for this register follow the same format as CAT bitmasks. On a processor with the Skylake microarchitecture, these new values should contain consecutive ones, while the Haswell microarchitecture does not require this (i.e., allowing any value in [0x60000, 0xFFFFF]).

To see whether this MSR register has an effect on performance, we measured the PCIe read/write hit rates (i.e., ItoM and PCIeRdCur events) while using different values for IIO LLC WAYS. We calculate the hit rate based on the number of hits and misses during an experiment where an I/O application processes packets of 1024 B at 100 Gbps while using 4096 RX descriptors. Fig. 5 shows that increasing the value of this MSR register leads to a higher PCIe read/write hit rate. This suggests that increasing the value of this register could improve the ability of the system to handle packets at high rates. We believe that the value of this register is *positively* correlated with the fraction of LLC used by DDIO. Using the technique in §4.1, we could not detect the newly added I/O ways, thus we speculate that the newly added ways follow a different policy (e.g., LRU) than the first two ways used for I/O. Therefore, we assume that the number of bits set specifies the number of ways used by DDIO.

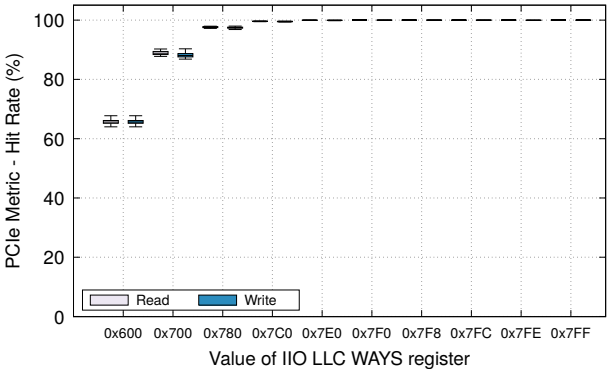


Figure 5: Tuning IIO LLC WAYS register increases PCIe read/write hit rates. The achieved throughput is 82-86 Gbps in this experiment.

Disabling DDIO. DDIO is bundled as a part of Intel Virtualization Technology (Intel VT), hence it is possible to enable/disable it in BIOS for some vendors [16, 23, 88]. According to [44, 72], DDIO can be disabled globally (i.e., by setting the `Disable_All_Allocating_Flows` bit in "iiomiscctrl" register) **or** per-root PCIe port (i.e., setting bit `NoSnoopOpWrEn` and unsetting bit `Use_Allocating_Flow_Wr` in "perfctrlsts_0" register). Some brief discussions of the benefits of disabling DDIO exist [11, 78], but we elaborate this more thoroughly in §7. We implemented an element for FastClick, called *DDIOTune*, which can enable/disable/tune DDIO†.

*One can read/write this register via `msr-tools` (e.g., rdmsr and wrmsr).

†The element is available at: https://github.com/tbarbette/fastclick/wiki/DDIOTune

5 Characterization of DDIO

This section scrutinizes the performance of DDIO in different scenarios while exploiting the tuning capability of DDIO. The goal is to show where DDIO becomes a bottleneck and when tuning DDIO matters. Therefore, we examined the impact of both system parameters (i.e., #RX descriptors, #cores, and processing time) and workload characteristics (i.e., packet size and rate) on DDIO performance. All of these measurements were done 20 times for both Skylake and Haswell microarchitectures. We observed the same behavior in both cases, but only discuss the Skylake results for the sake of brevity. We initially focus on the performance of an L2 forwarding network function, as an example of an I/O intensive application. Later, we discuss the impact of applications requiring more processing time per packet.

5.1 Packet Size and RX Descriptors

§3.1 discussed the negative consequence of a large number of RX descriptors on DDIO performance. This section continues this discussion by looking at the PCIe read/write hit rate metrics for different numbers of RX descriptors and different packet sizes. Fig. 6 shows the results of our experiments for PCIe write hit rate. PCIe read hit rates (not included for brevity) demonstrate similar behavior. When packets are >512 B, the PCIe read/write hit rates monotonically decrease with an increasing number of RX descriptors. More specifically, sending 1500-B packets, even with a relatively small number of RX descriptors (i.e., 128), causes 10% misses for both PCIe read and PCIe write hit rates. Furthermore, increasing the number of RX descriptors to 4096 makes DDIO operate at ~40% hit rate, hence 60% of packets require cache allocation and they had to be DMA-ed back to the NIC from main memory rather than LLC. Note that the packet generator is generating packets as fast as possible. Therefore, small packets show the case when the arrival *rate* is maximal, while large packets demonstrate maximal *throughput*, see Fig. 7.

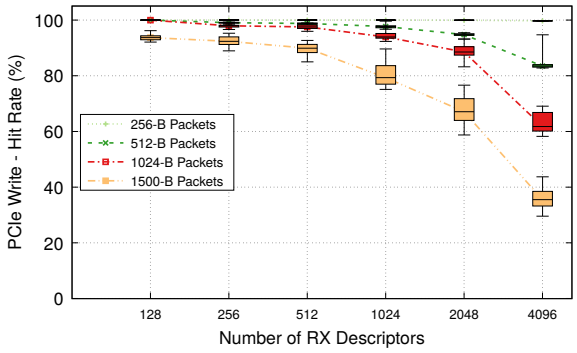
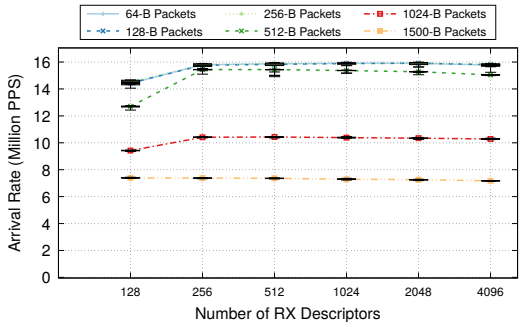
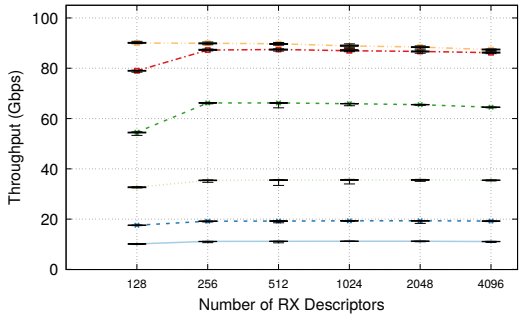


Figure 6: Increasing the number of descriptors and/or packet size *adversely* affects the performance of 2-way DDIO, while one core is forwarding packets at the maximum possible rate. We removed the results for 64-B and 128-B packets, as they show a behavior similar to 256-B packets.



(a) Arrival rate.



(b) Throughput.

Figure 7: Increasing the packet size reduces the arrival rate, i.e., the number of received/processed packets per second, due to NIC and PCIe limitations. Note that our testbed cannot exceed 90 Gbps when *only* one core is forwarding packets.

Unexpected I/O evictions. In some cases (e.g., 1500-B packets with 128 RX descriptors in Fig. 6), the size of the injected data is smaller than the DDIO capacity (i.e., 187.5 KiB $\ll$ 4.5 MiB). Even taking into account the TX descriptors and the FastClick's software queue, the maximum cache footprint of this workload is ~2 MiB. However, DDIO still experiences ~10% misses. We believe that this behavior may occur when an application cannot use the whole DDIO capacity due to (i) the undocumented cache replacement policy and/or (ii) the cache's complex addressing [15], thus multiple buffers may be loaded into the same cache set.

5.2 Packet Rate and Processing Time

§5.1 demonstrated that DDIO performs extremely poorly when a core does minimal processing at 100 Gbps. Next, we focus on the worst-case scenario of the previous experiment (i.e., sending 1500-B packets with 4096 RX descriptors) while changing the packet rate. To achieve 100 Gbps, we use two cores. Fig. 8 shows the PCIe read and PCIe write hit rates. The PCIe read metric results reveal that DDIO performs relatively well until reaching 98 Gbps. However, the PCIe write results indicate that DDIO has to continually allocate cache lines in LLC for 25% of packets at most of these throughputs, due to insufficient space for all of the buffers. Moreover, throughputs above 75 Gbps exacerbate this problem.

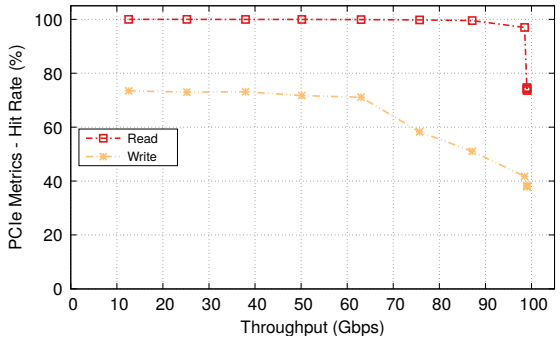


Figure 8: Increasing packet rates *negatively* impact the PCIe metrics, when 2 cores forward 1500-B packets with 4096 RX descriptors. The PCIe write metric is more degradation-prone.

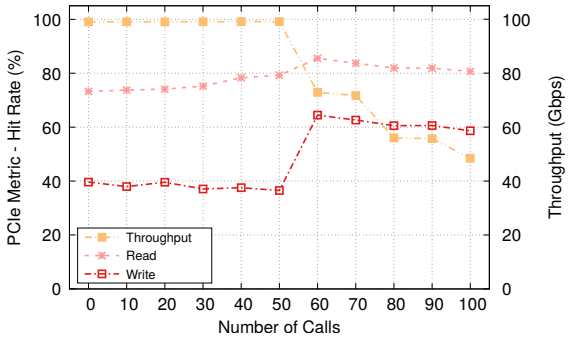


Figure 9: Making an application more compute-intensive results in better PCIe metrics, but *lower* throughput. In addition to forwarding packets, two cores call a dummy computation, while receiving 1500-B packets with a total of 4096 RX descriptors at 100 Gbps.

So far, we analyzed DDIO performance when cores performed minimal processing (i.e., swapping MAC addresses). Now, we analyze DDIO performance for more compute/memory-intensive I/O applications. Memory-intensive applications access memory frequently and execute few instructions per memory access. The time to accessing memory differs depending upon the availability of a cache line in a given part of the memory hierarchy. Therefore, we focus on the number of CPU cycles of the computation; noting that a memory access can be accounted for as given number of cycles. Note that increasing the processing time can change the memory access pattern, as packets continue to be injected by the NIC while some packets are enqueued in the LLC. To see the impact of different packet processing times on the performance of DDIO, we vary the amount of computation per packet by calling the `std::mt1993` random number generator multiple times. Ten such calls take ~70 cycles. Fig. 9 illustrates the effect of increasing per-packet processing time on the PCIe metrics & achieved throughput. These results demonstrate that increasing processing time slightly improves PCIe read hits rates up to ~60 calls, i.e., 400 cycles. This is expected, as increasing processing makes the application less I/O intensive as the application provides buffers to the NIC at a slower pace. However, increasing processing causes the available processing power (i.e., #cores) to become a bottleneck, substantially decreasing throughput. Similarly, PCIe write hit rates increases after exceeding 60 calls, due to a decrease in throughput & amount of cache injection. Therefore, DDIO performance *matters most* when an application is I/O bound, rather than CPU/memory bound.

5.3 Numbers of Cores and DDIO Capacity

When processing power limits an application's performance, the system should scale up/out. However, this scaling can affect DDIO's performance. To see the effect of scaling up, we measured the PCIe metrics while different numbers of cores were forwarding large packets. Fig. 10 shows that when an application is I/O intensive, increasing the number of cores improves the PCIe read/write hit rate, as it enhances the packet transmission rate because of more TX queues and faster consumption of packets enqueued in the LLC. To avoid synchronization problems, every queue is bound to one core. However, beyond a certain point (i.e., four cores in our testbed), increasing the number of cores causes *more* contention in the cache, as every core loads packets independently into the limited DDIO capacity. Furthermore, since newer processors are shipped with more cores, scaling up, even with a small number of RX descriptors, eventually causes the leaky DMA problem–the same problem as having a large number of descriptors (see §3.1).

Fig. 11 shows PCIe metrics for 1, 2, and 4 cores while changing the number of DDIO ways. Comparing the DDIO performance of different numbers of cores/DDIO ways, we conclude that increasing DDIO capacity leads to similar improvements for PCIe metrics. Therefore, increasing the DDIO capacity rather than the number of cores is beneficial when an application's bottleneck is *not* processing power or number of TX queues. Unless scaling up happens efficiently, some cores may receive more packets than others, causing

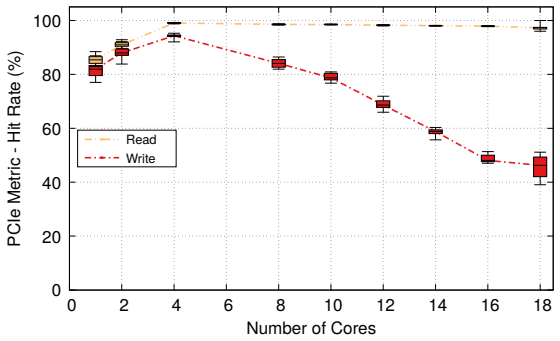


Figure 10: Increasing the number of cores does *not* always improve PCIe metrics for an I/O intensive application. Different numbers of cores are forwarding 1500-B packets at 100 Gbps with 256 RX descriptors per core.

performance degradation. We discuss the impact of load imbalance on DDIO performance in the next section.

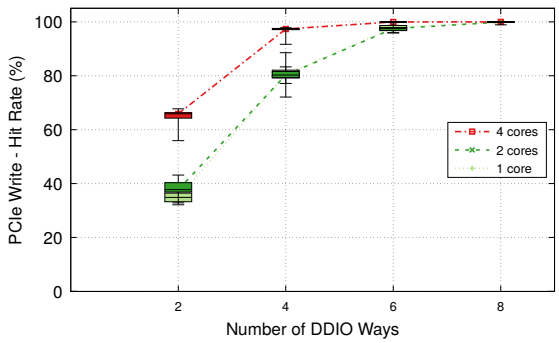


Figure 11: Increasing the number of DDIO ways can have a similar *positive* effect as increasing the number of processing cores, while forwarding 1500-B packets at 100 Gbps with a total of 4096 RX descriptors. PCIe read hit rate shows the same behavior as PCIe writes.

6 Application-level Performance Metrics

The previous section focused on the PCIe read/write hit rates and showed that increasing link speed & packet size and the number of descriptors & cores could degrade these metrics. PCIe read/write hit rates represent the percentage of I/O evictions (i.e., the performance of DDIO), but also *indirectly* affect application performance. The correlation between PCIe metrics and meaningful performance metrics (e.g., latency and throughput) depends on an application's characteristics. For instance, a low PCIe write hit rate can severely affect an application that requires the whole DMA-ed data. Conversely, the impact is much less for an application that needs only a subset of the DMA-ed packet. Fig. 2 showed one example of this correlation for the latter case, where the application only accessed the packet header. These results showed that *even* when an application does not require the whole DMA-ed data, increasing the number of descriptors (i.e., causing a reduction in PCIe hit rate metrics) could negatively affect the 99^{th} percentile latency. Note that we observed the same effect at median latency. This section further elaborates this impact in two scenarios where a stateful network function is processing a realistic workload* via 18 cores with a run-to-completion model [38, 93]. The benefits of increasing cache performance are not limited to this model and could be even greater for a pipeline model where fewer cores handle the I/O.

Stateful service chain. To evaluate the effect of increasing DDIO capacity, we chose a stateful service chain composed of a router, a network address port translator (NAPT), and a round-robin load balancer (LB) as a suitable chain to exploit hardware offloading capabilities of modern NICs while still keeping state at the processor. In this case, we offload the routing table of the router to the NIC and only handle the *stateful tasks* (i.e., NAPT + LB) and the basic functionality of the router in software. We generated 2423 IP filter rules for the campus trace using the GenerateIPFlowDirector element in Metron [38] and use DPDK's Flow API technology [31] to offload them into a Mellanox NIC. To examine the impact of load imbalance, we generate two different sets of rules with different load imbalance factors. One distributes the rules among 18 cores in a round-robin manner while the other is load-aware and tries to reduce the flow imbalance in terms of bytes received by every core. We calculated the number of packets received by each core for both cases and the maximum imbalance ratio of a core is 2.78× for the load-aware technique, while the round-robin technique causes 1.69× maximum load imbalance. The load-aware method has a higher load imbalance because we generate rules for the whole trace, but only replay a subset of it. Fig. 12 shows the 99^{th} percentile latency of this chain for different load balancing methods (with different load imbalance ratio), specifically increasing DDIO capacity reduces the 99^{th} percentile latency by ~21% when the load imbalance is higher. However, when the load imbalance is lower, these improvements reduce to ~2%. A higher load imbalance factor means that a core receives more packets than others, some of which could be evicted while enqueued in the LLC. Hence, it is crucial to realize a good balance to get the most out of DDIO. Furthermore, load imbalance is the root cause of many other performance degradations and is hard to prevent [8, 10].

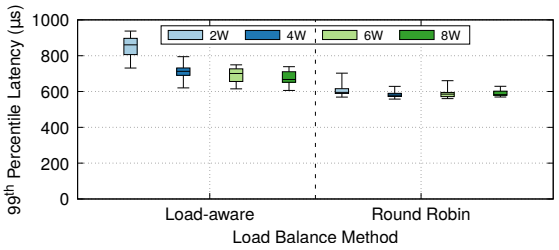


Figure 12: DDIO should be *carefully* tuned when the load imbalance factor is higher. The results shows 99^{th} percentile latency of a stateful network function while 18 cores are processing mixed-size packets at 100 Gbps. The throughputs were 94 & 97 Gbps for load-ware (higher imbalance) & round-robin (lower imbalance) experiments, respectively.

7 Is DDIO Always Beneficial?

The previous section showed that performance could be improved by tuning DDIO for I/O intensive network functions operating at ~100 Gbps. However, these results *cannot* be generalized, as the improvements are highly dependent on the application's characteristics. Moreover, there may be some applications that do not benefit from DDIO tuning. To investigate this, we measure the sensitivity of different applications to DDIO by enabling/disabling it (see §4.4). Table 2 shows the results for four applications/benchmarks:

*We replay the first 400 k packets of a 28-minute campus trace fifty times. The full trace has ~800 M packets with an average size of 981 B.

Table 2: DDIO sensitivity changes for different applications.

Application \ DDIO	Enabled				Disabled				Sensitivity
	Throughput	Median (µs)	Avg (µs)	99^{th} (µs)	Throughput	Median (µs)	Avg (µs)	99^{th} (µs)	
Memcached (TCP)	1003058 TPS	N/A	477.62	N/A	994387 TPS	N/A	481.62	N/A	Low
Memcached (UDP)	638763 TPS	N/A	750.12	N/A	631354 TPS	N/A	758.75	N/A	Low
NVMe (Full Write)	4427.2 MiB/s	44879.4	44437.6	46452.4	4434.2 MiB/s	44827	44374.68	46452.4	Low
NVMe (Random Read)	3372.4 MiB/s	582	589.67	765.7	3233.7 MiB/s	601.8	614.46	805.7	High
NVMe (Random Write)	1498.3 MiB/s	1307.8	1324.73	1991.2	1499.9 MiB/s	1309.5	1323.38	1971.4	Low
L2 Forwarding	98.01 Gbps	500.82	662	1055.98	87.02 Gbps	1058.15	862	1229.62	High
Stateful Service Chain (without offloading)	63.92 Gbps	665	657	923	63.25 Gbps	672	666	931	Low
Stateful Service Chain (with round-robin offloading)	97.35 Gbps	499	505	595	87.46 Gbps	531	924	1981	High

(i) DPDK-based implementation of Memcached developed by Seastar [5], (ii) an NVMe benchmarking tool (i.e., fio [4]), (iii) L2 forwarding application, (iv) a stateful service chain, used in §6, which performs IP filtering in software rather than offloading it to the NIC, and (v) the stateful service chain with round-robin offloading used in §6. We define sensitivity as "Low" if the maximum impact on the performance of an application is $\leq 5\%$. For Memcached, we use the method recommended by Seastar [2] with 8 instances of memaslap clients running for 120 s and a Memcached instance with 4 cores. For NVMe benchmarks, we tested a Toshiba NVMe (KXG50PNV1T02) with 4×1024-GB SSDs according to [3], where we report the average of 10 runs. The L2 forwarding application forwards mixed-size packets, while using 4 cores with a total of 4096 RX descriptors. The stateful service chain without offloading uses RSS to distribute packets among 18 cores (to increase the throughput) with 18×256 RX descriptors. The results demonstrate that different applications have different levels of sensitivity to DDIO, which can be exploited by system developers to optimize their system in a multi-tenant environment, where multiple I/O applications co-exist, see §8.1. The most sensitive application is L2 forwarding, which is the most I/O intensive application among these applications and can run at line rate. Some applications (e.g., Memcached) experience less benefit from DDIO, as their performance may be bounded by other bottlenecks. A more detailed sensitivity analysis of different applications remains as our future work.

8 Future Directions for DCA

Tuning DDIO occupancy was shown to substantially improve the performance of some applications. However, increasing the portion of the cache used for I/O is only a temporary solution for two reasons: (i) I/O is only a part of packet processing and (ii) to achieve suitable performance many networking applications require a large amount of cache memory for *code/data*. Moreover, many network functions would benefit from performing in-cache flow classification [92]; hence, there is a trade-off between allocating cache to I/O vs. code/data and this trade-off depends on the application's characteristics & cache size. Additionally, since DDIO is way-based, the granularity of partitions is quite coarse in recent Intel processors, due to low set-associativity. Therefore, it is harder to partition the cache fairly between code/data & I/O. These reasons, together with the recent trend in Intel processors of decreasing per-core LLC, eventually make the current implementation of DCA a major bottleneck to achieving low-latency service times. Hence, DCA needs to deliver better performance even with a small fraction of the cache. This makes it necessary to *rethink* the current DCA designs with an eye toward realizing network services running at multi-hundred gigabits per second. Some possible directions/proposals for future DCA are: ① Fine-grained placement: adopting CacheDirector [15] methodology (i.e., sending packets to the appropriate LLC slices) and only sending the relevant parts of these packets to the L2 cache, L1 cache, or *potentially* CPU registers [26]; ② Selective DMA/DCA: only DMA relevant parts of the packet (as required by an application) to the cache and buffer the rest in either main memory, the NIC, or Top-of-Rack switch; and ③ I/O isolation: extend CAT to include I/O prioritization in addition to Code and Data Prioritization (CDP) technology [60] to alleviate I/O contention. These ideas could be simulated in a cycle accurate simulator (e.g., gem5 [6, 12]), which remains as our future work. Next, we examine one potential solution in the current systems to better take advantage of DDIO.

8.1 Bypassing Cache

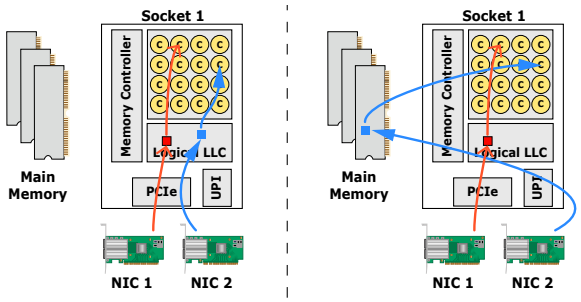
§3.1 explained that one way to prevent unnecessary memory accesses and the leaky DMA problem is to reduce the number of descriptors. However, this could increase packet loss and generate more PAUSE frames at high link rates. Unfortunately, both can have a severe impact on the service time as they postpone the service time by at least a couple of microseconds. Taking these consequences into account, we believe future DCA technologies should perform cache injection more effectively: *DMA should not be directed to the cache if this would cause I/O evictions*; thus, buffering packets in local memory (at a cost of only several hundreds of nanoseconds) is preferable to dropping or enqueuing packets in previous nodes. Additionally, bypassing cache would be beneficial in a multi-

tenant scenario where performance isolation is desired. For instance, low-priority and/or low-DDIO-sensitive applications could bypass cache to make room for high-priority and/or high-DDIO-sensitive applications. In addition, one could prioritize [7] different traffic flows, thus only a subset of received traffic (and hence cores) would use cache for I/O. Implementing a system to prioritize DDIO for different flows either in a programmable switch or modern NICs (e.g., Mellanox Socket Direct Adapters) remains as our future work.

Evaluation. To evaluate the benefits of bypassing the cache, we use two methods: (i) disabling DDIO and (ii) exploiting DMA via a remote socket (see §4.3). We set up a 200-Gbps testbed, see Fig. 13. We first connect two 100-Gbps NICs to the same socket. Next, we connect one of these NICs to a remote socket. We run two instances of L2 forwarding application located on the first socket, each of which uses 4 cores and one NIC to forward mixed-size packets. We chose four cores per NIC because our earlier experiments (see Fig. 10) showed that DDIO can achieve an acceptable performance while receiving 1500-B packets with four cores. To reduce the contention for cache and memory bandwidth, we apply CAT & MBA to each application (similar to ResQ [83]). We assume that one of the applications has a higher priority, and we measure its latency in five different scenarios: **(i)** without the presence of the low-priority application, **(ii)** when the low-priority application pollutes the cache via 2-way DDIO (see Fig. 13a), **(iii)** when the low-priority application pollutes the cache via 4-way DDIO, **(iv)** when the low-priority application bypasses the cache by DMA-ing packets via a remote socket (see Fig. 13b), **(v)** when the low-priority application bypasses the cache via disabled DDIO. Fig. 14 shows the 99^{th} percentile latency of the high-priority application–other percentiles show a similar trend with a smaller difference. These results demonstrate that bypassing cache via a remote socket (i.e., case iv) achieves the same latency as when there is no low-priority application (i.e., case i). However, when both applications are receiving traffic via DDIO (i.e., case ii), the 99^{th} percentile latency degrades ~30%. We observe that bypassing cache has the same benefits as increasing DDIO capacity (i.e., case iii vs. case iv). Furthermore, comparing cases (iv) and (v) indicates that disabling DDIO *slightly* pollutes the cache (as opposed to bypassing via a remote socket). We speculate that disabling DDIO only affects the packets, not the descriptors. Therefore, we conclude that bypassing cache can result in less variability in performance and potentially better performance isolation. Additionally, it is clearly necessary to tune DDIO capacity when moving toward 200 Gbps.

9 Lessons Learned: Optimization Guidelines

This section summarizes our key findings, which could help system designers/developers to optimize DDIO for their applications. Furthermore, our study should inspire computer architects to improve DCA's performance by



(a) Through local socket. (b) Through remote socket.

Figure 13: Receiver setup to achieve 200 Gbps. On the right setup, the second NIC is connected to the remote socket. It sends packets through UPI link directly to the main memory.

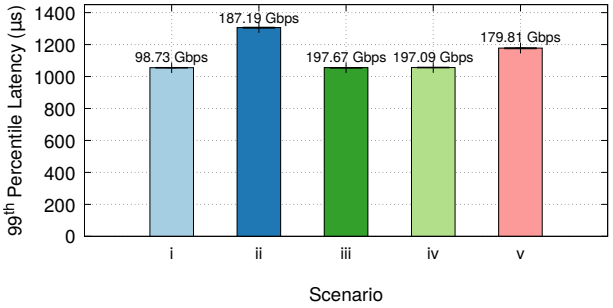


Figure 14: Bypassing cache and tuning DDIO at 200 Gbps mitigate I/O contention and improve the tail latency of the high-priority application up to 30%. Scenarios: (i) 100 Gbps with no contention; (ii) contention at 200 Gbps; (iii) tuning DDIO at 200 Gbps; (iv) bypassing cache via a remote socket; and (v) bypassing cache via disabled DDIO. The total achieved throughput of the receiver is written on the bars.

offering increasing control. Although we focused on packet processing, our work is not limited to network functions. Our investigations could be equally useful in other contexts (e.g., HPC) that require high-bandwidth I/O when transferring data via RDMA and processing with GPUs. We showed that current approaches to avoid DDIO becoming a bottleneck are only temporary solutions and they are inapplicable to multi-hundred-gigabit network applications. We proposed a benchmarking method to understand the unknown & little-discussed details of DDIO. Later, we characterized the performance of DDIO in different scenarios and showed the benefits of bypassing the cache. We concluded that there is *no* one-size-fits-all approach to utilize DDIO. Our study reveals:

- The locations of LLC to which DDIO injects data (§4.1).
- Co-locating an application's code/data with I/O in the cache could adversely impact its performance (§4.2).
- The way that DDIO behavior changes for different system parameters and workload characteristics (§5).
- If an application is I/O bound, adding excessive cores could degrade its performance (Fig. 10).
- If an application is I/O bound, *carefully* sizing the DDIO capacity can improve its performance and could lead to the

same improvements as adding more cores (Fig. 11).
- If an application starts to become CPU bound, adding more cores can increase its throughput, but then it has to balance load among cores to *maximize* DDIO benefits (Fig. 12).
- If an application is *truly* CPU/memory bound, DDIO tuning is less efficient (Fig. 9). However, it can be beneficial to buffer in DRAM incoming requests/packets which cannot be processed in time, rather than having the NIC issue PAUSE frames or drop packets.
- Going beyond ~75 Gbps can cause DDIO to become a bottleneck (Fig. 8). Therefore, it is essential to *bypass cache* to realize performance isolation. Bypassing cache could be done for low-priority traffic or applications that do not benefit from DDIO (§8.1).
- Different applications have different levels of sensitivity to DDIO (§7). Identifying this level is essential to utilize system resources more efficiently, provide performance isolation, and improve performance.

10 Related Work

The most relevant work to our study is ResQ [83], which we discussed thoroughly in §3.1 and §8.1. This section discusses other efforts relevant to our work.

Injecting I/O into the cache. The idea of loading I/O data directly to the processor's cache was initially proposed using cache injection techniques [52, 63]. Later, it was used to enhance network performance on commodity servers and was referred to as DCA [25]. Amit Kumar et al. [42] investigated the role of coherency protocol in DCA. Their results indicated that the benefit of DCA would be limited when the network processing rate cannot match the I/O rate. In addition, [75] showed that DCA could cause cache pollution; hence they proposed an alternative cache injection mechanism to mitigate the problem. A. Kumar et al. [43] characterized DCA for 10-Gbps Ethernet links. Other works have discussed that DCA is insufficient due to architectural limitations [40, 46, 71]. For example, the work in [46] proposed a new I/O architecture that decouples and offloads I/O descriptor management from the NIC to an on-chip network engine. Similarly, the work in [40] proposed a flexible network DMA interface which can support DCA. Last but not least, Wen Su et al. [71] proposed an improvement to combine DCA with an integrated NIC to reduce latency.

Efforts toward realizing 100 Gbps. Many have tried to tackle challenges to achieve suitable performance for fast networks, mostly in the context of NFV [49] and key-value stores [19, 45]. Some research has exploited new features in modern/smart/programmable NICs (e.g., [38, 47, 84, 94]) & switches (e.g., [36]) or proposed new features (e.g., [70]) to offload costly software processing. A number of works investigate packet processing models (e.g., [9, 39, 93]). CacheBuilder [80] and CacheDirector [15] have discussed the importance of cache management in realizing 100-Gbps networks. HALO [92] exploited the non-uniform cache architecture (NUCA) characteristics of LLC to perform in-cache flow classification. Last but not least, IOctopus [68] proposed a new NIC design and wiring for servers to avoid non-uniform DMA penalties. Our work is complementary to these works.

Cache partitioning. Many have tried to overcome cache contention by performing cache partitioning [53]. These efforts can be split into two main categories: (i) software techniques and (ii) hardware techniques. The former group principally relies on physical addresses to partition cache based on sets [22, 48, 67] or slices [15]. This way of cache partitioning does not require any hardware support, but it is not very commonly used, due to its drawbacks (e.g., OS/App modification and costly re-partitioning). The latter group mostly exploits way-partitioning (e.g., CAT) to partition the cache among different applications [13, 18, 62, 89, 90, 91]. In addition to these techniques, Wang et al. [85] proposed a hybrid approach that combines both techniques to achieve finer granularity for partitioning. To the best of our knowledge, there are only two works (ResQ [83] and CacheDirector [15]) that have specifically tried to exploit cache partitioning techniques to improve packet processing. ResQ proposes to isolate a percentage of LLC that is used for I/O and CacheDirector exploits the NUCA used in Intel processors to distribute I/O more efficiently among different LLC slices. Our work is complementary to these works, as most of them do not consider I/O when partitioning the cache.

11 Conclusion

DCA technologies were introduced to improve the performance of networking applications. However, we systematically showed that the latest implementation of DCA in Intel processors (i.e., DDIO) cannot perform as needed with increasing link speeds. We demonstrated that better I/O management is required to meet the critical latency requirements of future networks. Our main goal is to emphasize that networking is, now more than before, tightly coupled with the capability of the current hardware. Consequently, realizing time-critical multi-hundred-gigabit networks is only possible by (i) increasingly well-documented control over the hardware and (ii) improved holistic system design optimizations.

Acknowledgments

We would like to thank our shepherd, Mark Silberstein, and anonymous reviewers for their insightful comments. We are grateful to Tom Barbette for helping us with his NPF tool. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The work was also funded by the Swedish Foundation for Strategic research (SSF). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 770889).

References

[1] Direct Cache Access (DCA), Oct 2010. ftp://supermicro.com/ISO_Extracted/CDR-X8-Q_1.02_for_Intel_X8_Q_platform/Intel/LAN/v16.3/PROXGB/DOCS/SERVER/DCA.htm, accessed 2019-08-05.

[2] Memcached Benchmark, 2015. https://github.com/scylladb/seastar/wiki/Memcached-Benchmark, accessed 2019-12-30.

[3] Benchmarking - Benchmarking Linux with Sysbench, FIO, Ioping, and UnixBench: Lots of Examples. https://wiki.mikejung.biz/Benchmarking, 2018.

[4] Flexible I/O Tester (fio). https://fio.readthedocs.io/en/latest/fio_doc.html, 2019.

[5] Seastar. http://seastar.io/, 2019.

[6] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. Data Direct I/O Characterization for Future I/O System Exploration. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020. https://yifanyuan3.github.io/publication/ddio_gem5, accessed 2020-05-20.

[7] Philip C Arellano and James A Coleman. Method, apparatus, and system for allocating cache using traffic class, March 30 2017. US Patent App. 14/866,862.

[8] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. RSS++: Load and State-Aware Receive Side Scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[10] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, Santa Clara, CA, February 2020. USENIX Association.

[11] Harsha Basavaraj. A case for effective utilization of Direct Cache Access for big data workloads. Master's thesis, UC San Diego, 2017. https://escholarship.org/uc/item/0fr3735b, accessed 2019-07-24.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[13] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, Feb 2018.

[14] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[15] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 8:1–8:17, New York, NY, USA, 2019. ACM.

[16] Financial Services Industry (FSI) - Frequently Asked Questions. https://software.intel.com/en-us/articles/financial-services-industry-fsi-frequently-asked-questions, accessed 2019-07-24.

[17] Intel Forum. Intel Ethernet X520 to XL710 - Tuning the buffers: a practical guide to reduce or avoid packet loss in DPDK applications. https://etherealmind.com/wp-content/uploads/2017/01/X520_to_XL710_Tuning_The_Buffers.pdf, accessed 2019-07-24.

[18] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-Driven LLC Allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 295–308, Denver, CO, June 2016. USENIX Association.

[19] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[20] Jeff Gilbert and Mark Rowland. The Intel Xeon Processor E5 Family: Architecture, Power Efficiency, and Performance, August 2012. https://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-8-DataCenter/HC24.29.827-Xeon-Rowland-Xeon-E5-2600-Disclaimer.pdf, accessed 2019-07-24.

[21] Andrew Herdrich, Khawar Abbasi, and Marcel Cornu. Introduction to Memory Bandwidth Allocation, March 2019. https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation, accessed 2019-07-24.

[22] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A Predictable Cache-Aware Memory Allocator. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 23–32, July 2011.

[23] How to disable Data Direct I/O (DDIO) on Intel Xeon E5? https://forums.intel.com/s/question/0D50P0000490NFhSAM/how-to-disable-data-direct-io-ddio-on-intel-xeon-e5?language=en_US, accessed 2019-07-24.

[24] Ted Hudek. Introduction to Receive Side Scaling, 04 2017. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, accessed 2019-12-29.

[25] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, June 2005.

[26] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The Case for a Network Fast Path to the CPU. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 52–59, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Information about PCM PCIe counters. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/543883, accessed 2019-07-24.

[28] Intel. Intel Data Direct I/O Technology Overview, 2012. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html, accessed 2019-07-26.

[29] Intel. Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring, July 2017. https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-uncore-performance-monitoring-manual.html, accessed 2019-07-26.

[30] Intel. Intel Arria 10 Avalon-ST Interface with SR-IOV PCIe Solutions User Guide, 2019. https://www.intel.com/content/www/us/en/programmable/documentation/lbl1415123763821.html#lbl1453336559194, accessed 2019-07-26.

[31] Intel Ethernet Flow Director and Memcached Performance, 2014. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf, accessed 2019-09-09.

[32] IO Issues: Remote Socket Accesses. https://software.intel.com/en-us/vtune-amplifier-cookbook-io-issues-remote-socket-accesses, accessed 2019-09-01.

[33] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge, 2012. https://bit.ly/2LKVfZr, accessed 2019-07-24.

[34] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[35] Muthurajan Jayakumar. Data Plane Development Kit: Performance Optimization Guidelines. https://software.intel.com/en-us/articles/dpdk-performance-optimization-guidelines-white-paper, accessed 2019-07-24.

[36] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[38] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Conference on Networked Systems Design and Implementation (NSDI 18)*, NSDI'18, pages 171–186, Renton, WA, 2018. USENIX Association.

[39] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr., and Dejan Kostić. SNF: synthesizing high performance NFV service chains. *PeerJ Computer Science*, 2:e98, November 2016.

[40] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. *SIGPLAN Not.*, 51(4):67–81, March 2016.

[41] Maciek Konstantynowicz, Patrick Lu, and Shrikant M. Shah. Benchmarking and Analysis of Software Data Planes. Technical report, Cisco, Intel Corporation, FD.io, Dec 2017. https://fd.io/wp-content/uploads/sites/34/2018/01/performance_analysis_sw_data_planes_dec21_2017.pdf, accessed 2019-07-24.

[42] A. Kumar and R. Huggahalli. Impact of Cache Coherence Protocols on the Processing of Network Traffic. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 161–171, Dec 2007.

[43] A. Kumar, R. Huggahalli, and S. Makineni. Characterization of Direct Cache Access on multi-core systems and 10GbE. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 341–352, Feb 2009.

[44] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *S&P*, May 2020. Intel Bounty Reward.

[45] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform. *ACM Trans. Comput. Syst.*, 34(2):5:1–5:30, April 2016.

[46] G. Liao, X. Znu, and L. Bnuyan. A new server I/O architecture for high speed networks. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 255–265, Feb 2011.

[47] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[48] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, Feb 2008.

[49] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of Performance Acceleration Techniques for Network Function Virtualization. *Proceedings of the IEEE*, 107(4):746–764, April 2019.

[50] Patrick Lu. Performance Considerations for Packet Processing on Intel Architecture, May 2017. https://fdio-vpp.readthedocs.io/en/latest/events/Summits/FDioMiniSummit/OSS_2017/2017_05_10_performanceconsideration.html, accessed 2019-07-24.

[51] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, Carlsbad, CA, October 2018. USENIX Association.

[52] V. Milutinovic, A. Milenkovic, and G. Sheaffer. The cache injection/cofetch architecture: initial performance evaluation. In *Proceedings Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 63–64, Jan 1997.

[53] Sparsh Mittal. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Comput. Surv.*, 50(2):27:1–27:39, May 2017.

[54] Jeffrey C. Mogul and John Wilkes. Nines are Not Enough: Meaningful Metrics for Clouds. In *Proc. 17th Workshop on Hot Topics in Operating Systems (HoTOS)*, 2019.

[55] David Mulnix. Intel Xeon Processor Scalable Family Technical Overview, Sep 2017. https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview, accessed 2019-07-24.

[56] NetApp. What is the potential impact of PAUSE frames on a network connection?, Nov 2017. https://ntap.com/2RpAx1Q, accessed 2019-07-24.

[57] Network Performance Framework. https://github.com/tbarbette/npf, accessed 2019-07-24.

[58] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018*

Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, pages 327–341, New York, NY, USA, 2018. ACM.

[59] Khang Nguyen. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family, Feb 2016. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, accessed 2019-07-24.

[60] Khang T Nguyen. Code and Data Prioritization - Introduction and Usage Models in the Intel® Xeon® Processor E5 v4 Family, 2016. https://software.intel.com/en-us/articles/introduction-to-code-and-data-prioritization-with-usage-models, accessed 2019-07-26.

[61] John Ousterhout. Always Measure One Level Deeper. *Commun. ACM*, 61(7):74–83, June 2018.

[62] Jinsu Park, Seongbeom Park, and Woongki Baek. 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*, EuroSys '19, pages 10:1–10:16, New York, NY, USA, 2019. ACM.

[63] Hazim Shafi Patrick Joseph Bohrer, Ramakrishnan Rajamony. Method and apparatus for accelerating input/output processing using cache injections , March 2004. US Patent No. US6711650B1.

[64] PCIe Bandwidth Drops on Skylake-SP. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/741386, accessed 2019-07-24.

[65] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[66] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros. Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, April 2016.

[67] Timothy Sherwood, Brad Calder, and Joel Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 155–164, New York, NY, USA, 1999. ACM.

[68] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. IOctopus: Outsmarting Nonuniform DMA. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 101–115, New York, NY, USA, 2020. Association for Computing Machinery.

[69] Splash-3 Benchmark Suite. https://github.com/SakalisC/Splash-3, accessed 2019-07-24.

[70] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, Boston, MA, February 2019. USENIX Association.

[71] W. Su, L. Zhang, D. Tang, and X. Gao. Using Direct Cache Access Combined with Integrated NIC Architecture to Accelerate Network Processing. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 509–515, June 2012.

[72] Roman Sudarikov and Patrick Lu. Hardware-Level Performance Analysis of Platform I/O, June 2018. https://dpdkprcsummit2018.sched.com/event/EsPa/hardware-level-performance-analysis-of-platform-io, accessed 2019-07-24.

[73] Supermicro. *1028UX-LL1-B8, 1028UX-LL2-B8, and 1028-LL3-B8 User's Manual*. https://www.supermicro.com/manuals/superserver/1U/MNL-1886.pdf, accessed 2019-07-24.

[74] Supermicro. *6028UX-TR4 User's Manual*. https://www.supermicro.com/manuals/superserver/2U/MNL-1706.pdf, accessed 2019-07-24.

[75] D. Tang, Y. Bao, W. Hu, and M. Chen. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010.

[76] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-Channel, 2019. https://arxiv.org/pdf/1909.04841.pdf, accessed 2019-09-15.

[77] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Observing Network Packets over a Cache Side-Channel. In *Proceedings of the 47th*

International Symposium on Computer Architecture, ISCA '20, New York, NY, USA, 2020.

[78] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions. *CoRR*, abs/1810.09360, 2018.

[79] Temporary PCIe Bandwidth Drops on Haswell-v3. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/600913, accessed 2019-07-24.

[80] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. Dark Packets and the End of Network Scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, pages 1–14, New York, NY, USA, 2018. ACM.

[81] Shelby Thomas, Geoffrey M. Voelker, and George Porter. CacheCloud: Towards Speed-of-light Datacenter Communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[82] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 21:1–21:16, New York, NY, USA, 2019. ACM.

[83] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 283–297, Renton, WA, April 2018. USENIX Association.

[84] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[85] 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 *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, Feb 2017.

[86] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization, Jan 2017. https://software.intel.com/en-us/articles/intel-performance-counter-monitor, accessed 2019-07-24.

[87] Henry Wong. Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, accessed 2019-07-24.

[88] Xeon E5 disable DDIO in OS? https://forums.intel.com/s/question/0D50P0000490VP0SAM/xeon-e5-disable-ddio-in-os?language=en_US, accessed 2019-07-24.

[89] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 13:1–13:15, New York, NY, USA, 2018. ACM.

[90] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 14:1–14:13, New York, NY, USA, 2018. ACM.

[91] M. Xu, L. Thi, X. Phan, H. Y. Choi, and I. Lee. vCAT: Dynamic Cache Management Using CAT Virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 211–222, April 2017.

[92] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 601–614, New York, NY, USA, 2019. ACM.

[93] Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. A Closer Look at NFV Execution Models. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, pages 85–91, New York, NY, USA, 2019. ACM.

[94] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, Sep. 2014.