Data Direct I/O Characterization for Future I/O System Exploration

Mohammad Alian, *UIUC*, malian2@illinois.edu
Yifan Yuan, *UIUC*, yifany3@illinois.edu
Jie Zhang, *KAIST*, jie@camelab.org
Ren Wang, *Intel Labs*, ren.wang@intel.com
Myoungsoo Jung, *KAIST*, mj@camelab.org
Nam Sung Kim, *UIUC*, nskim@illinois.edu

***Abstract*—I/O performance plays a critical role in the overall performance of modern servers. The emergence of ultra high-speed I/O devices makes the data movement between processors, main memory, and devices a major performance bottleneck. Conventionally, the main memory is used as an intermediate buffer between the processor and I/O devices and I/O devices cannot directly access processor side caches. Data Direct I/O (DDIO) technology aims to reduce the memory bandwidth utilization by enabling the I/O devices to leverage Last Level Cache (LLC) as the intermediate buffer. Our experimental results show that DDIO can completely eliminate memory bandwidth utilization while running network- or storage-intensive applications. However, when modeling the I/O subsystem using architectural simulators, DDIO is often ignored, which can result in inaccurate assessments about the I/O and memory subsystem of emerging and future large-scale computer systems. In this paper, we provide a detailed background on DDIO technology in Intel server processors. Then we present our cycle-accurate I/O subsystem model in gem5 simulator that can be configured to model DDIO. We verify our model against baseline gem5 and validate it by comparing its results against a physical computer system.**

I. Introduction

We are living in the data explosion era. The rate of data growth is continuously increasing with the widespread use of Internet of Things (IoT), social media, streaming video services, and many more emerging cloud services. Technology giants such as Google, Facebook deal with an unprecedented amount of data and number of users at their datacenters; As of early 2018, 3.5 billion daily Google web searches and 1.5 billion daily active Facebook users are reported [1]. Processing a single web search query requires crawling the web across thousands of servers [2]. Activities in social media contribute to the generation of new data, the processing of old data, and the movement of data across servers, racks, and even datacenters [3]. Looking closely at the data processing in such huge scale, the instruction execution of big-data applications is bottlenecked by the data movement between processing units, memory modules, network adaptors, and storage devices.

Because of the large footprint of modern big-data applications, their dataset neither fits in the on-chip caches nor the off-chip memory modules. Therefore, such applications experience frequent I/O accesses to fetch their data either from remote nodes or local storage devices [4], [5]. In conventional servers, to access the data of an I/O device, the data has to first be delivered to a DRAM buffer and then the DRAM buffer can be used by the processor. This causes multiple trips over the memory channels to access the data. This was not a concern for slow I/O devices, but with the advancement in I/O technology and emergence of ultra high bandwidth, low latency network and storage devices, memory utilization for I/O accesses became a bottleneck [6]. To alleviate the memory bandwidth bottleneck in high bandwidth I/O devices, direct cache access (DCA) has been proposed [7]. The current implementation of DCA technology on Intel Xeon processors, also known as Data Direct I/O (DDIO) technology, uses processor's last level cache (LLC) as the intermediate buffer between the processor and I/O devices. Traditionally, DDIO (from Intel Sandy Bridge architecture) mainly benefits network packet processing and GPUs. But recently, with the advancement in non-volatile storage technology, PCIe based high-speed storage devices (*e.g.*, SSDs) can also leverage this feature.

Although DDIO is shown to be useful in reducing memory bandwidth utilization and I/O access latency, it can show non-intuitive performance implications. For instance, Tootoonchian *et al*. [8] study the impact of DDIO on network function performance when co-running multiple network functions on a server. Specifically, they showed that DMA data can compete for the DDIO-assigned LLC space and result in more LLC miss, and thus performance degradation. Marinos *et al*. [9] observe that DDIO benefits cannot be exploited for video streaming applications due to the long reuse distance of storage data that is fetched into LLC. Kurth *et al*. [10] observe the latency benefit of DDIO over memory, which potentially leads to a security attack. These studies show that a DDIO enabled server can result in completely different memory access patterns and even application performance compared with a server without DDIO.

Computer architects widely use architectural simulators to realize their designs and assess their performance. gem5 [11] is the state of the art architectural simulator with large support from industry and academia. gem5 is capable of simulating processors, memory subsystem [12], off-chip devices [13]–[15], and multi-node systems [16]. Unfortunately, the I/O subsystem of gem5 does not accurately model a real system. The connection bridge between processor/memory and off-chip I/O devices is implemented with a focus on making it only functionally correct and does not accurately model the performance of a real I/O bridge.

In this paper, we first explain DCA implementation in

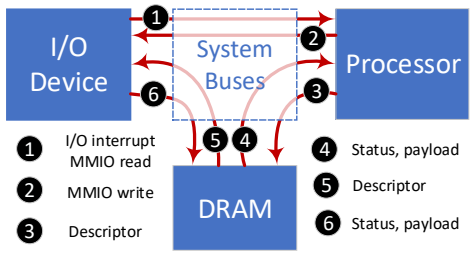


Fig. 1. I/O operations within a DMA enabled system.

Intel processors. Intel DDIO is a platform feature on all modern Xeon processors which implements DCA for all PCI-Express attached I/O devices, including network and storage I/O accesses. Then we characterize the behavior of several network and block I/O-intensive applications when DDIO is enabled and disabled. We study the impact of DDIO on memory utilization and application performance. Although the impact of DDIO on networking subsystem is well studied in the literature [7], [17]–[21], its impact on block I/O devices is often neglected or even ignored. To the best of our knowledge, this paper is the first work that characterizes DDIO performance for block I/O accesses. In the second part of this paper, we focus on enhancing I/O subsystem of gem5 to enable it to accurately model the performance and functionality of a server with DDIO enabled or disabled. We verify the correctness of our model with the baseline gem5 I/O model and validate its performance by comparing its behavior against physical hardware. Lastly, we discuss some interesting use-cases for our model that will open new research avenues that were not possible to explore without our detailed DDIO model in gem5.

II. Background

In this section, we provide the necessary background on the I/O subsystem interaction with memory and CPU, explaining performance bottlenecks during the data movement. We then explain Intel Data Direct I/O (DDIO), the Intel implementation of Direct Cache Access (DCA), a Xeon processor feature aiming to reduce I/O overhead and improve performance and energy consumption.

A. Basic Data Movement Operations

Data movement between IO adaptors and CPU/memory requires tight collaboration and interaction of multiple software and hardware components. At high level, a common operation involves memory-mapped I/O (MMIO), Direct Memory Access (DMA), and I/O interrupt method. Fig. 1 illustrates the typical I/O operation with a DMA enabled I/O devices, similar to the flow described in [7]. First, the processor configures a data buffer in its system memory for data reception (Rx) and transmission (Tx). The buffer information is conveyed to I/O adaptors via descriptors containing pointers to the allocated data buffer. Once the descriptors are created, the processor writes to an MMIO register owned by the adaptor to inform it of the readiness for Rx/Tx operations. The I/O adaptors, upon receiving the indication, fetch the corresponding descriptors and then can carry out the block data transfer via DMA operations.

DMA is a method that enables I/O adaptors to transfer data directly to or from system memory without the processor's involvement, hence allowing the processor to execute other tasks concurrently. Once the DMA operation is finished, the I/O adaptors update the status in memory to indicate data readiness and then issue an I/O interrupt to notify the processor. Upon receiving I/O interrupts, the processor enters the interrupt handling routine, figuring out the cause of the interrupt, and carry out corresponding actions accordingly. In the case of Rx operation, the processor reads the newly arrived data from memory for processing. In the case of Tx operation, the processor understands the I/O adaptor has DMAed data from memory so it can move forward.

Until recently, the basic communication model, along with PCI-Express as I/O interconnect and gigabit Ethernet NIC, worked reasonably well and supported various network-based applications. However, with the emergence of 10∼100Gb/s (and beyond) High-speed NICs and new requirements from software, I/O performance is becoming the bottleneck. New technologies, optimized implementations, and improved programming models are being developed to alleviate the I/O bottlenecks. For example, PCIe Gen5 [22] will double the throughput of Gen 4 to provide 128GB/s of raw throughput. Newer platforms allow data to be served from processor cache if present [23]. Poll Mode Driver (PMD) in DPDK [24] and Linux kernel allows software I/O drivers to directly access Rx/Tx descriptors without suffering the overhead of interrupt routines.

Even with these optimizations, the default DMA process for I/O transaction still poses a significant impact on the performance and efficiency of the processor. DDIO (or known as DCA) has been proposed and implemented to address this issue, as we explain next.

B. Data Direct I/O: Motivation and Implementation

As explained in detail in [7], looking at I/O interactions from a system memory/cache perspective reveals the potential performance bottleneck of the current DMA model. The data structures used for communication–status, descriptors, and payload–are within the coherency domain and cacheable in the processor. As a result, whenever an I/O adaptor issues memory read and write requests via DMA operations, the processor cache needs to be snooped for each cacheline to ensure the previous copy of the same cacheline is properly invalidated, in order to maintain data coherency. After DMA operations, the payload is stored in system memory. Typically, most applications or networking stacks (*e.g.*, TCP/IP) requires accessing payload soon, via issuing processor read requests. These memory accesses cause cache misses, and data need to be fetched from system memory (DRAM). Missing cache and reading from DRAM induces longer latency and higher memory bandwidth, resulting in sub-optimal performance and energy efficiency.

DCA [7] has been proposed to address this issue. DCA aims to bypass the DRAM, and write data directly to the cache on the Rx path, or read directly from cache on the Tx path. DCA provides two major benefits: 1) Largely eliminating memory transactions required by DMA, reducing DRAM bandwidth and energy consumed due to DRAM access. 2) Significantly lowering latency from the process since data can be found in and served from the cache. This helps in improving application performance.

There are various ways to implement DCA. Intel DDIO technology is a platform feature on all modern Xeon processors. DDIO uses Last Level Cache (LLC) as the primary destination and source for I/O and processor data communication instead of DRAM. With DDIO, I/O adaptors directly read from and write to LLC without making a detour to DRAM, reducing both access latency and memory bandwidth. Specifically, DDIO behavior can be described as following [25].

Tx Path: I/O adaptor reads data from the processor. Without DDIO, a read request from the I/O adaptor causes data payload to be forwarded from the cache if present. After forwarding, corresponding cachelines are to be evicted from the cache. The rationale of this flow, which potentially causes multiple DRAM trips, is based on the assumption that I/O devices were in general slow, implying low temporal locality. The timely eviction may have improved cache utilization with slow I/O. However, this assumption is often not true anymore with the fast development of high-speed I/O, where cachelines are being reused frequently. As a result, on newer platforms, cachelines are not evicted to DRAM after serving an I/O read, making the default Tx behavior similar to that of DDIO as described below.

With DDIO, the cachelines corresponding to data are not evicted after Tx operations. Hence, they normally stay in cache. This leads to 1) Data access from the software creating required data are normally served from the cache, and CPU

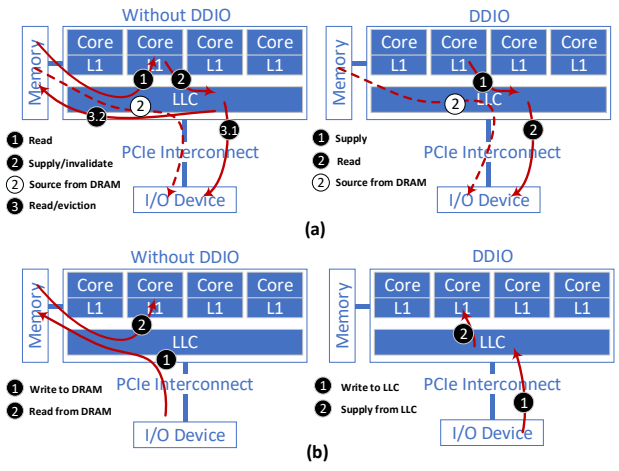
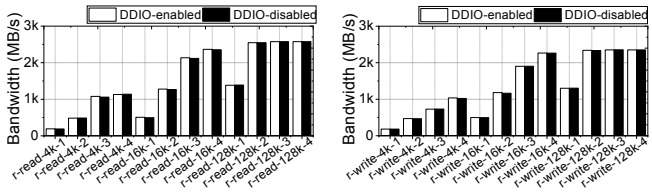


Fig. 2. Data movement between processor caches, system memory, and I/O devices in a system with and without DDIO enabled. (a) Tx Path; (b) Rx Path



(a) SSD read bandwidth. (b) SSD write bandwidth.

Fig. 3. SSD performance measured in non-NUMA systems with enabled/disabled DDIO.

does not need to fetch data from DRAM; and 2) Read request from I/O is usually satisfied from the cache as well without causing extra evictions. As a result, Latency and DRAM accesses are both reduced on the Tx path with DDIO technology. Fig. 2 compares the difference with and without DDIO on the Tx path where I/O reads data from the processor.

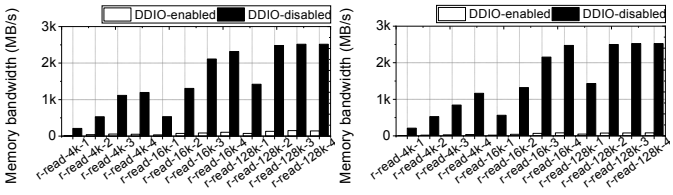
Rx Path: I/O adaptor writes data to the processor. Without DDIO, a write I/O operation causes the I/O adaptor to transfer (usually using DMA) to host system memory (DRAM). If the data existed in the processor cache, it is invalidated. When the processor needs the data and issues data accesses, they will miss the cache entirely, and the data has to be brought in from DRAM, causing long latency and many DRAM accesses.

With DDIO, most memory access on the Rx path can be eliminated via allocating data directly into LLC without having to make a detour to the system memory. This is accomplished by using hardware flows: "Write Update" or "Write Allocate" depends on whether or not the memory address of the data exists in the processor's cache. 1) When the memory addresses are in the cache hierarchy, I/O writes causes in-place updates of the cachelines in LLC, eliminating trips to the system memory. 2) when the memory addresses do not exist in the processor cache, new cachelines are allocated in LLC to accommodate incoming data, and no need to bring in any data from DRAM. Since the newly written data are kept in Modified state and will be Write-back to memory eventually if needed, the data coherency is maintained. Fig. 2. compares the difference with and without DDIO on the Rx path where I/O writes data to the processor.

Studies show that DDIO can reduce latency, save energy, and improve performance for various workloads [25]. DDIO usually is restricted to use a certain number of cache ways in LLC to be conservative and leave enough LLC space to applications running on the same processor. The default settings of the number of DDIO ways vary depending on the type of platforms.

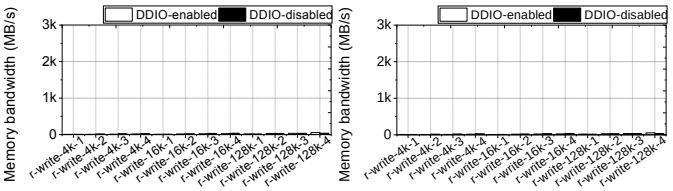
III. DDIO Characterization

As discussed in Sec. II, DDIO technology can reduce the data movements between processor and memory for I/O operations. To limit potential cache contentions between I/O and cores, by default, Intel assigns 2 LLC ways as I/O ways to servers, or 4 LLC ways to certain storage SKUs (short



(a) Memory read bandwidth. (b) Memory write bandwidth.

Fig. 4. Memory bandwidth utilization measured in non-NUMA systems with enabled/disabled DDIO (SSD read).



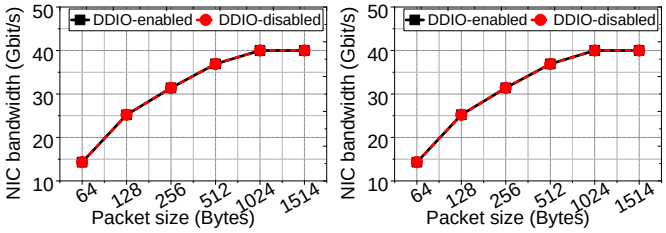
(a) Memory read bandwidth. (b) Memory write bandwidth.

Fig. 5. Memory write bandwidth utilization measured in non-NUMA systems with enabled/disabled DDIO (SSD write).

for "Stock Keeping Units"). Note that the current DDIO implementation does not distinguish between different I/O devices, and all LLC ways for DDIO are used by all the I/O devices.

We set up a real hardware platform to evaluate how DDIO technology impacts I/O-intensive and memory-intensive workloads. Please refer to Section VI-A for the details of experimental setup. The key observations of DDIO's characteristics are summarized as follows:

Observation 1: DDIO decreases memory bandwidth utilization. We execute a storage microbenchmark and compare the SSD performance and memory utilization between the systems that enable and disable DDIO in Figs. 3, 4 and 5. Note that we only show the results of random read/write, and the sequential read/write results have the same characteristics. "Xk-X" in the x-axis indicates the I/O block size and I/O depth. As shown in the figure, while employing DDIO does not improve the I/O throughput, it can significantly reduce the memory bandwidth utilization in the host. Specifically, the memory bandwidth utilization by SSD random read operation is up to 2.5 GB/s (both memory read and write) in the DDIO-disabled systems, whereas the memory bandwidth utilization decreases to less than 100 MB/s by enabling DDIO. This is because DDIO technology places the data of the underlying SSD to LLC and directly serves the requests of CPU from LLC rather than accessing the main memory. The memory bandwidth utilization of SSD write operations is low even when disabling DDIO. This is in contrast with the description of a DDIO disabled system in [25] As also described in Sec. II, after some investigation, we found out that Xeon processor generations newer than Sandy Bridge-EP no longer evict an LLC cacheline after a read access from a PCIe device, regardless of DDIO being enabled or disabled. Moreover,

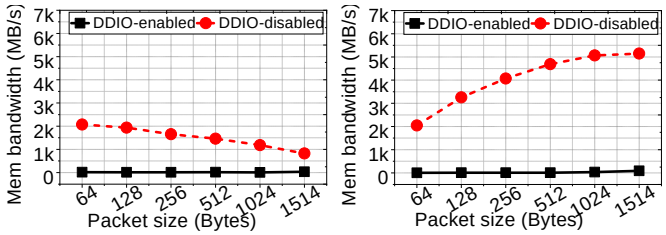


(a) NIC Rx. (b) NIC Tx.

Packet size (Bytes)	64	128	256	512	1024	1514
DDIO-enabled Rx	27.9	24.6	15.4	9.1	4.9	3.3
DDIO-disabled Rx	27.9	24.6	15.4	9.1	4.9	3.3
DDIO-enabled Tx	27.9	24.6	15.4	9.1	4.9	3.3
DDIO-disabled Tx	27.9	24.6	15.4	9.1	4.9	3.3

(c) Packet rate (Mpps).

Fig. 6. NIC performance measured in non-NUMA systems with enabled/disabled DDIO.



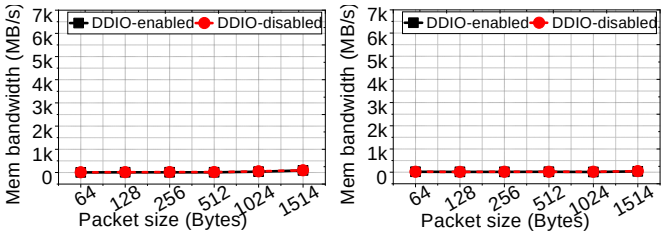
(a) Memory read bandwidth. (b) Memory write bandwidth.

Fig. 7. Memory bandwidth utilization measured in non-NUMA systems with enabled/disabled DDIO (NIC Rx).

because streams of data are written to SSD by the microbenchmark, without any temporal locality, enabling or disabling DDIO does not have any impact on the I/O throughput.

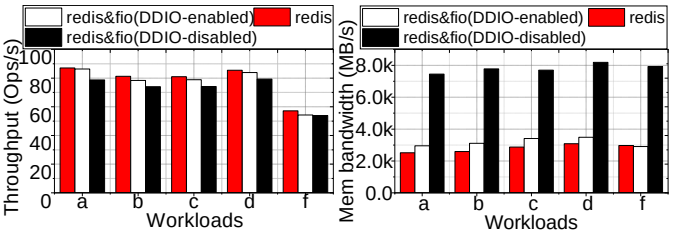
Fig. 6, Fig. 7 and Fig. 8 show the network RX/TX bandwidth, memory bandwidth utilization of RX, and memory bandwidth utilization of TX. In the case of Rx with DDIO disabled, the memory write bandwidth utilization is almost equal to the achieved network bandwidth (*i.e.*, up to ∼40Gbps). On the other hand, the memory read bandwidth decreases gradually as the packet size increases, and its value is smaller than that of the memory write bandwidth (*e.g.*, 822MB/s with the packet size of 1514B). This is because, in our case, *l3fwd* only deals with the packet headers. That is, while the NIC always writes all the data to the main memory, the CPU cores only fetch the data of the packet header (as well as descriptors and metadata of the *mbuf* data structure in DPDK) into LLC. In other words, the memory read bandwidth here is related to the packet rate, not the network bandwidth. On the other hand, when DDIO is enabled, the network traffic will introduce negligible memory bandwidth consumption to the system.

Observation 2: DDIO improves the performance of memory-intensive applications. We further evaluate the impact of DDIO on the memory-intensive workloads by configuring three different scenarios: executing redis alone, co-running redis and fio (SSD sequential read) in the DDIO-enabled system, and co-running redis and fio in the DDIO-



(a) Memory read bandwidth. (b) Memory write bandwidth.

Fig. 8. Memory bandwidth utilization measured in non-NUMA systems with enabled/disabled DDIO (NIC Tx).



(a) Performance. (b) Memory bandwidth.

Fig. 9. The impact of storage access and DDIO on memory-intensive workload.

disabled system. Fig. 9 shows the performance of different redis workloads in the three scenarios. As shown in the figure, I/O-intensive workloads do not impact the performance of memory-intensive workloads, when DDIO is enabled. In contrast, the performance of redis degrades by 7.4% when I/O-intensive workloads are running in a DDIO-disabled system. As the data of SSD needs to be stored in main memory, the I/O intensive workloads compete with memory-intensive workloads for the memory bandwidth resources. We also collect the memory bandwidth of the three scenarios, which is shown in Fig. 9b. "redis&fio (DDIO-disabled)" costs 2.8× and 2.5× more memory bandwidth than "redis" and "redis&fio(DDIO-enabled)", respectively.

IV. GEM5 I/O MODEL

Before we start explaining our DDIO model in gem5, we provide a quick background on the I/O subsystem modeling in current gem5 release with the classic memory model. gem5 classic memory model implements a MOSEI coherence protocol between different cache entities in the system. Memory bus (MemBus) is the point of coherency in the system. Coherent caches respond to snoop requests from other caches and forward them to other coherent components that are connected to their ports. Because the focus of this work is on DDIO modeling, we do not go to the details of coherency protocol implementation in the gem5 classic memory model and only explain the interactions between coherency protocol and I/O subsystem.

To give the I/O devices a coherent view of the memory subsystem and also match the I/O bus width with that of MemBus, gem5 passes all the I/O data through a special

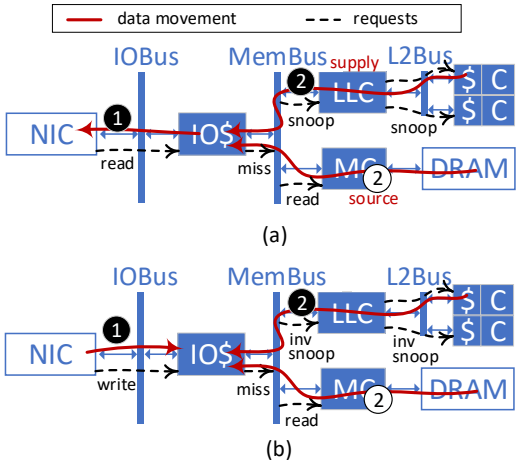


Fig. 10. gem5 baseline I/O data movement. (a) NIC data consumption, (b) NIC data delivery.

cache called IOCache. As depicted in Fig. 10, the CPU side of IOCache is connected to IOBus, and its memory side is connected to MemBus. IOBus is a non-coherent bus that connects the DMA port of all I/O devices to IOcache. IOBus resembles a PCI interconnect. Here we explain the data movement in gem5 memory subsystem when an I/O device reads/writes from/to a memory address.

Tx Path: I/O adaptor reads data from the processor. when a DMA read packet arrives at IOCache any of the following scenarios can occur:

(1) Memory address hits in IOCache: The data is served from IOCache without any activity on MemBus, LLC, and DRAM. Step ❶ in Fig. 10(a) illustrates this scenario. Forwarding data directly from cache instead of going to DRAM resembles a DDIO implementation. However, in DDIO, we access data in LLC, but here we do not have any activity on LLC.

(2) Memory address misses in IOCache: In this case, a miss request is created at IOCache and send to MemBus. MemBus will broadcast snoop requests to all its coherent slave ports. Depending on the location of the data we have the following cases: (2.1) Memory address is cached in the processor side caches (*e.g.* L1, L2, or LLC): The cacheline of the responding cache is downgraded to Owned or Shared state and is delivered to the IOCache. IOCache is filled with the delivered cacheline in Shared state. Note that gem5 does not invalidate the cached data in the processor side caches, which is not exactly the behavior that we expect from a non-DDIO implementation (see Sec. II). Step ❷ in Fig. 10(a) illustrates this case. (2.2) Memory address is not in the caching hierarchy: MemBus will send a memory read request to the memory controller, and the DMA read request is serviced from memory and cached in the IOCache in Exclusive state. Step ② in Fig. 10(a) illustrates this case. Having data cached in IOCache is neither a non-DDIO nor a DDIO implementation because, in both cases, we source data from DRAM without allocating it in a cache [25].

Rx Path: I/O adaptor writes data to the processor. when

a DMA write request arrives at IOCache, we can have the following cases:

(1) Data is cached in Exclusive or Modified state in IOCache: the data in IOCache is updated with the new DMA write request, and no further activity is needed. Step ❶ in Fig. 10(b) illustrates this case. This resembles the "Write Update" step in DDIO implementation. However, in DDIO, the updated block is stored in LLC, not in a separate cache (*i.e.*, IOCache).

(2) Data is cached in Shared or Owned state in IOCache: an Invalidate request is sent to the MemBus to be forwarded to all the processor side caches. After all the other (possible) shared instances of the cacheline are invalidated from the processor side caches, the cache block is updated with the DMA write request data, leaving the cache block in Modified state. The `inv` requests in Fig. 10(b) illustrates this step. Again this resembles the "Write Update" in DDIO with a difference that the "Update" happens inside IOCache instead of LLC.

(3) Data is not cached in IOCache: If the DMA write request is a whole cacheline write request[1], an Invalidate request is sent to the MemBus to be forwarded to all the processor side caches. After all the other (possible) shared instances of the cacheline are invalidated from the processor side caches, a cache block in IOCache is allocated, and its content is updated with the DMA write request data, leaving the cacheline in Modified state. The `inv` requests in Fig. 10(b) illustrates this step. On the other hand, if the DMA write request is not a whole cacheline write, IOCache first sends a read exclusive request (*i.e.*, `ReadExReq`) to the MemBus to fetch an exclusive copy of the whole cacheline that will be updated. `ReadExReq` invalidates any copy of the cacheline in the caching hierarchy after read (refer to Table. I for the explanation of related gem5 memory commands). Step ❷ in Fig. 10(b) illustrates this case when the address is cached in processor side caches, and step ② shows the case where the snoop requests do not hit in the processor side caches and an exclusive copy of the cacheline is read from DRAM and sent to IOCache. This case resembles "Write Allocate" in DDIO, but again the allocation takes place in IOCache instead of LLC.

In the next section, we explain how we enabled gem5 to accurately model the I/O subsystem of a server where DDIO is enabled or disabled.

V. DDIO Modeling in gem5

As discussed in Sec IV, the current gem5 memory model with IOCache neither simulate a system without DDIO, nor a system with DDIO: it is something in between. IOCache can lead to inaccurate and misleading experimental results for I/O intensive applications as it acts as a large buffer between I/O devices and processor. Ideally, instead of IOCache, we want to have an I/O controller module that provides coherency for I/O devices, matches the width of IOBus and MemBus while implementing the correct behavior of non-DDIO and DDIO processor features. In this section, we explain our I/O model

[1]A whole cacheline write request is a write request that its size is equal to the cacheline size and its address is cacheline aligned.

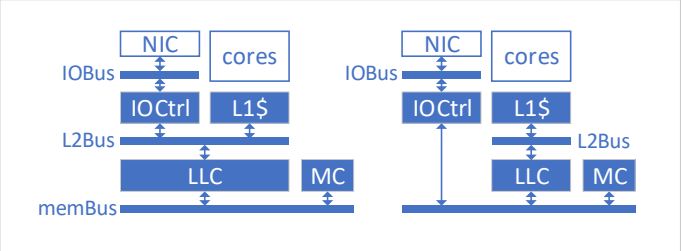


Fig. 11. IOCtrl placement when DDIO is enabled (left) and disabled (right).

for gem5 and its implementation that closely resembles the I/O subsystem of real hardware.

Instead of IOCache, we need a component which more accurately models the root complex (*i.e.*, chipset) in real systems. We call this component I/O controller (*i.e.*, IOCtrl). IOCtrl operates in two modes: DDIO and NON-DDIO. To model the correct DDIO and NON-DDIO behaviors, we modify IOCache to model IOCtrl and also enable LLC to identify I/O data and implement the correct actions upon receiving I/O data. The following subsections explain IOCtrl and LLC logic when operating in DDIO and NON-DDIO modes.

IOCtrl logic. IOCtrl is derived from Cache object in gem5. Similar to IOCache, IOCtrl is within the coherency domain of the processor side caches. IOCtrl implements a configurable fully associative buffer that stores read and write requests before it gets forwarded to LLC, DRAM, or I/O devices. The size of each buffer entry is equal to the cacheline size. IOCtrl uses configurable numbers of Miss Status Holding Registers (MSHR) to implement a non-blocking coherent I/O system. Fig. 11 depicts the connection between IOCtrl and the rest of the memory subsystem in the simulated environment when DDIO is enabled or disabled. Because all the I/O traffic needs to go through LLC when DDIO is enabled, we connect IOCtrl directly to L2Bus instead of memBus.

When a DMA read request arrives at IOCtrl (from a DMA device), first, the local buffer is searched for a match. If there is a match, then the read request is going to be satisfied from the IOCtrl buffer. Otherwise, if DDIO is enabled, IOCtrl tries to acquire a *shared copy* of the requested cacheline; if DDIO is disabled, then IOCtrl acquires an *exclusive copy* of the cacheline. That is, a miss packet with `ReadSharedReq` and `ReadExReq` command is created for the former and latter cases, respectively. Once MemBus or L2Bus receives the miss packet, it sends snoop requests to all of its coherent ports. If any of the caches is responsive (*i.e.*, the address is cached in them), then the data for the read request is going to be supplied by the responsive cache. Based on the MOESI protocol, by default, the new state of the cache block from the responsive cache would be Owned. However, if DDIO is not enabled, we should invalidate the cacheline if it is cached in the cache hierarchy. This is why we acquire an exclusive copy of the cacheline when DDIO is disabled. The state of the responsive cache is going to change to Owned and Invalid after `ReadSharedReq` and `ReqExReq` miss packets are serviced, respectively.

TABLE I
DESCRIPTION OF GEM5 MEMORY COMMANDS RELATED TO DDIO IMPLEMENTATION.

Commands	Description
$InvalidateReq$	Invalidate any copy of the cacheline in the cache hierarchy.
$WriteLineReq$	Full cacheline write req - always destined for memory if the source is IOCache
$ReadExReq$	Cacheline aligned read that will invalidate any copies of the cacheline in other caches.
$ReadShareReq$	Cacheline aligned that will read a shareble copy of cacheline from cache hierarchy or memory.

When a DMA write request arrives at IOCtrl (from a DMA device), if the address is found in Modified or Exclusive state in the local buffer, the write request is going to be performed inside the IOCtrl without any further action on MemBus or L2Bus. If the DMA write request misses in the IOCtrl or the address is found in Shared or Owned states, we have the following cases: If *DDIO is enabled*, we need to invalidate all the copies in processor side caches except for the LLC copy (if present) and send the write request directly to LLC. To minimize the changes in the gem5 memory subsystem, we do not exactly implement this in LLC but have an implementation that emulates the same behavior. We explain our approach later when we describe the changes in the LLC logic. If *DDIO is disabled*, we need to invalidate all copies in processor side caches, including any LLC copy if present. This is simple, and depending on if the write request is a whole cacheline or partial cacheline, an `InvalidateReq` or a `ReadExReq` should be issued, respectively.

Once all the copies of the write address are invalidated, and IOCtrl acquires an exclusive copy of the cacheline address, the DMA write request can be safely written into the IOCtrl. However, considering the large access granularity of the I/O devices and limited buffering of IOCtrl, shortly the dirty cachelines inside IOCtrl is going to be replaced. Before sending a writeback, IOCtrl appends an ID field called "ioDataID" to the packet carrying the writeback data. Each I/O device has a non-zero and unique "ioDataID". "ioDataID" is used by LLC to allocate different I/O data in the right LLC way. Also note that, if DDIO is disabled, the writebacks from IOCtrl buffers will be written to DRAM directly.

LLC logic. Alg. 1 illustrates how LLC handles I/O data and regular memory requests when DDIO is enabled. Note that LLC is just a regular cache object, and we only add two parameters, namely "isLLC" and "ddioEnabled", to distinguish the LLC from other caches and also implement the correct behavior when DDIO is enabled.

However, to avoid unnecessary changes in the gem5 memory subsystem, even when DDIO is enabled, we invalidate LLC copies but set a flag for the invalidate block called "ioInvalidated". Later, when a writeback is received at the LLC from IOCtrl, we first look for any block in a chosen LLC set that has "ioInvalidated" flag set. If there is any, that block is chosen to satisfy the writeback. Otherwise, based on the DDIO LLC allocation, we chose a victim way inside the chosen set and satisfy the writeback. If DDIO is disabled, as illustrated in Fig 11, the LLC is not in the path of the I/O data and all the IOCtrl writebacks are forwarded to the DRAM.

Algorithm 1: LLC logic operations when DDIO is enabled

```
1  switch request command do
2      case InvalidateReq OR ReadExReq do
3          Lookup cache;
4          if command is ReadExReq then
5              Supply the data;
6          end
7          if is cached then
8              Invalidate the block;
9              if ioDataID != 0 then
10                 Set ioInvalidated flag for the block;
11             else
12                 Reset ioInvalidate flag for the block;
13             end
14         end
15     case writeback AND ioDataID != 0 do
16         Lookup cache;
17         if is cached then
18             Update the cache block
19         else
20             dst_set = Find the destination set;
21             if There is an ioInvalidated block then
22                 Write into that block;
23                 reset ioInvalidated flag;
24             else
25                 Allocate a block in the assigned DDIO ways
26             end
27         end
28     case default do
29         Baseline cache operations;
30 end
```

VI. EVALUATION

A. Methodology

Hardware environment. We evaluate the impact of DDIO on a real hardware platform. Specifically, we configure the hardware platform with dual Intel Xeon Platinum 8260 CPUs [26], each has 24 physical cores and a 35.75MB LLC. We also configure a 96GB DDR4 DRAM-based main memory, which is shared by all cores. For the storage evaluation, our platform has an Intel Optane DC P4800X NVMe SSD [27], which can deliver up to 2.4GB/s read bandwidth and 2GB/s write bandwidth. For network-related evaluation, we equip the platform with one Intel XL710 40GbE NIC [28]. We use another platform with the same configuration as the network traffic generator and connect these two platforms with a 40Gb Ethernet cable.

Benchmarks. We leverage *fio* [29] to evaluate the performance of the storage with microbenchmark workloads, *e.g.*, sequential read (s-read), random read (r-read), sequential write

(s-write) and random write (r-write). fio adopts various I/O parameters. By default, we configure the I/O block size as 128KB, I/O depth as 4, and I/O engine as *libaio*. Such a set of configurations is sufficient to saturate the read/write bandwidths of the Optane SSD. We also evaluate the performance of a representative in-memory database (*redis* [30]) by examining various Yahoo! Cloud Serving Benchmark (YCSB) workloads [31], which reflects different query patterns [32][2]. By default, we set the database size as 8GB and the number of worker threads as 4. We directly collect SSD bandwidth and database throughput from the built-in counters of *fio* and YCSB, respectively.

For network-related evaluations, we use DPDK's *l3fwd* [33] with network traffic generated by *pktgen-DPDK* [34]. To separate the behavior of NIC TX and RX, we modify the default code of *l3fwd* to sink all the packets after the output port lookup (this means the program will only receive and process packets, but does not send them), so that we can measure the TX-related metrics with *DPDK-pktgen* and RX-related metrics with *l3fwd*. By default, we set up 2 Tx queues and 2 Rx queues (with 2048 descriptors) for *l3fwd*, and assign each Tx/Rx queue pair with a dedicated CPU core. Besides, we leverage Intel Performance Counter Monitor (PCM) [35] to get the DRAM bandwidth.

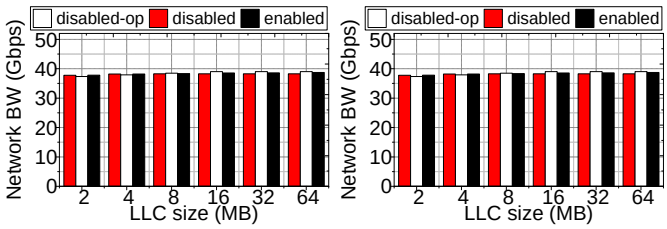
Note that, to make evaluation results precise, we leverage *numactl* utility to bind all the tested workloads to the same NUMA node. To guarantee a consistent CPU performance, we disable the DVFS governor in the Linux and configure all CPU cores to work with performance governor (*i.e.*, the frequency of the CPU cores are fixed to the highest frequency) . In addition, we completely write all blocks into the Optane SSD and flush the DRAM buffer in a warm-up phase before executing the experiments.

To show that our gem5 model can accurately model the behavior of the I/O subsystem of a modern server, we implement a NIC device driver in gem5 bare-metal and use it to generate I/O traffic. The size of TX and RX descriptor rings is 1024. We configure two gem5 nodes, one sender and one receiver to send and receive 1514Byte packets in one direction. We configure each gem5 node with an out of order double-core processor running at 2.4GHZ, two DDR4-2400 memory channels, one 40Gbps NIC with two hardware queues, and two-level cache hierarchy with LLC size and associativity set to 32MB and 16ways, respectively (unless stated otherwise). In Sec. VI-B, we discuss the results that verify our gem5 model. Note that we do not do validation against physical hardware because it needs careful gem5 calibration for cores, memory system, operating system, etc. which is outside the scope of this work.

B. Verification of the gem5 model

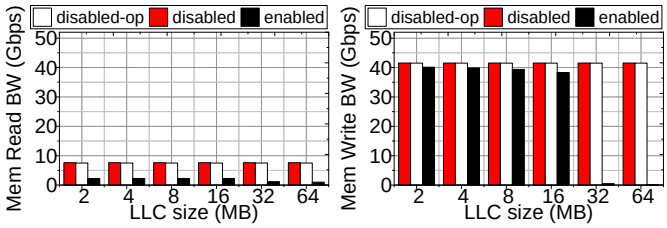
Fig. 12a and Fig. 12b show the achieved network bandwidth between two gem5 nodes with different LLC size, when three different I/O configurations are employed:

[2]We do not include workload-E since it is for range query, while others are for single query.



(a) RX. (b) TX.

Fig. 12. Network bandwidth.



(a) Memory read. (b) Memory write.

Fig. 13. Memory bandwidth in RX.

disabled-op: The default gem5 I/O model. An optimized version of DDIO disabled configuration where DMA reads do not invalidate LLC data (refer to Sec.III).

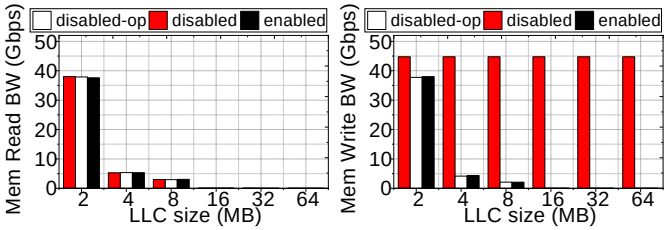
disabled: DDIO is disabled.

enabled: DDIO is enabled, and 10% of the LLC ways are assigned to DDIO data.

As shown, LLC size has minimal impact on the network bandwidth, and all the configurations achieve ∼40Gbps bandwidth.

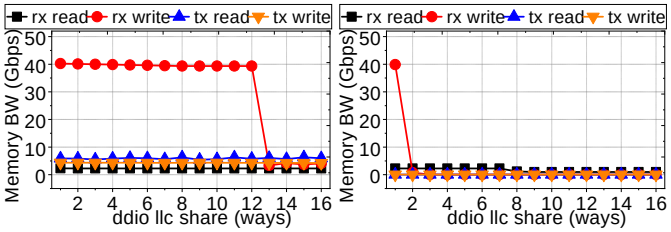
Fig. 13a shows the memory read bandwidth utilization at the receiver node. As expected, when DDIO is disabled, we only see the memory read bandwidth corresponding to the reading of headers from the memory, which exactly matches the behavior of Fig. 7a. Fig. 13b shows the memory write bandwidth at the receiver node. As expected, when DDIO is disabled, the memory write bandwidth matches the network bandwidth as all the network data has to be written to the memory. On the other hand, when DDIO is enabled, for LLC size less than 32MB, DDIO with 10% of LLC share fails to fit the network data, and the data leaks to the memory. Note that two full 1024 element Rx descriptor rings with a packet size of 1514 occupy more than 3MB of LLC. Considering the conflicts and the 10% LLC share for DDIO, Fig 13b suggests the LLC share, packet size, and descriptor ring size should be carefully configured to avoid DMA leaks to the memory [8]. Note that *disabled* and *disabled-op* shows similar behavior at the RX side, as we also see in the real hardware experiments (*c.f.* Sec. III).

Fig.14a shows the memory read bandwidth utilization at the sender node. For all different DDIO configurations, the memory read bandwidth utilization is similar across different LLC sizes. This is not surprising because there is no data reuse in the TX descriptor rings of our sender node as the node



(a) Memory read. (b) Memory write.

Fig. 14. Memory bandwidth in TX.



(a) 4MB LLC. (b) 32MB LLC.

Fig. 15. Memory bandwidth under different LLC ways.

always writes new data to the descriptor ring for transmission. When the LLC size is less than 3MB, the ring buffers do not fit inside the LLC and that causes cacheline conflicts and evictions of the earlier to-be-sent packets. This results in future NIC DMA reads to be satisfied from the DRAM instead of LLC. Increasing the LLC size results in lesser cacheline conflicts due to TX ring buffer fill ups and lesser evictions of the to-be-send packets. Fig. 14a shows this trend across different LLC size.

As explained in Sec. 2, when DDIO is disabled, DMA reads results in eviction of dirty cachelines from LLC to DRAM. This effect can be observed in Fig. 14b by looking at the data corresponding to *disabled* configuration. The large write bandwidth utilization in *disabled-op* and *enabled* configurations when LLC size is set to 2MB is because of write conflicts cache evictions. As explained earlier, because the whole TX ring buffer does not fit inside the 2MB LLC we have conflicting writes. For *disabled-op* and *enabled* configurations, we still see such conflicting writes even in 4MB and 8MB LLC but their frequency are significantly lower than 2MB LLC configuration.

C. Usecases

In this section, we use our DDIO enabled model and try to study its effectiveness when using different LLC sizes and various LLC way allocations. Fig 15a and Fig 15b show the memory read and write bandwidth utilization when restricting DDIO to use 1 to 16 (all of the ways) LLC ways with LLC size set to 4MB and 32MB, respectively. There is a sudden drop in the memory write bandwidth when increasing DDIO's share over 12 and 1 ways for 4MB and 32MB configurations, respectively. This shows that DDIO is effective in eliminating the memory write bandwidth utilization once the entire RX ring buffer can fit inside the DDIO's LLC portion.

The main takeaway from Fig. 15 is that an optimal system I/O configuration needs careful consideration about the underlying hardware and software features, such as DDIO's LLC sharing, ring buffer size, and network interface's maximum transmission unit size. We expect that more fine-grain tuning mechanisms for DDIO configuration, as well as for the network stack, will further benefit the performance. Our DDIO model paves the path for a new line of research for intelligent inbound I/O-data placement on the processor chip. Modern server processors statically use 10%-20% of the LLC space for I/O data. Can a dynamic LLC allocation based on the runtime demands of I/O intensive applications improve the overall systems performance? Can more aggressive injection of I/O data to other levels of processor's cache hierarchy improve applications performance? All these are interesting research questions and possible topics for future work.

VII. Conclusion

In this paper, we first explained Data Direct I/O (DDIO) feature in Intel processors in detail. We explained the detailed behavior of a cache coherent I/O system with DDIO enabled and disabled. We showed experimental results about how DDIO can impact the memory bandwidth utilization and also the performance of running applications. Considering the importance of architectural simulations in the field, we took gem5 and enhanced its I/O subsystem to model a DDIO enabled system. Lastly, we ran several experiments with our gem5 model to verify its correctness and show how this model can be useful when studying scale-out network performance.

VIII. Acknowledgments

We would like to thank the anonymous reviewers for their comments. We also thank Anil Vasudevan, Robert Blankenship, Bin Li and Charlie Tai, for their insightful feedback and technical support. This work is supported by funding from National Science Foundation (No. CNS-1705047).

References

[1] B. Marr, "How much data do we create every day? the mind-blowing stats everyone should read," https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#6ab0df8960ba, 2018.

[2] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE micro*, 2003.

[3] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787472

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, 2010.

[5] K. Shvachko, H. Kuang, S. Radia, R. Chansler *et al.*, "The hadoop distributed file system." in *MSST*, vol. 10, 2010.

[6] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proceedings of the VLDB Endowment*, vol. 9, 2016.

[7] 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)*. IEEE, 2005.

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

[9] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart, "Disk—Crypt—Net: Rethinking the stack for high-performance video streaming," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

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

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, 2011.

[12] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, "Simulating dram controllers for future system architecture exploration," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014.

[13] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, 2014.

[14] M. Jung, J. Zhang, A. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. Kandemir, "SimpleSSD: Modeling solid state drives for holistic system simulation," *IEEE Computer Architecture Letters*, vol. 17, 2017.

[15] M. Alian, K. P. Srinivasan, and N. S. Kim, "Simulating PCI-Express interconnect for future system exploration," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018.

[16] M. Alian, U. Darbaz, G. Dozsa, S. Diestelhorst, D. Kim, and N. S. Kim, "dist-gem5: Distributed simulation of computer clusters," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017.

[17] 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)*. Ieee, 2007.

[18] 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*. IEEE, 2009.

[19] 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*. IEEE, 2012.

[20] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015.

[21] G. Liao, "Accelerating I/O processing in server architectures," Ph.D. dissertation, UC Riverside, 2011.

[22] J. Hruska, "PCIe 5.0 arriving in 2019 with 4x more bandwidth than PCIe 3.0," https://www.extremetech.com/computing/250640-pci-sig-announces-plans-launch-pcie-5-0-2019-4x-bandwidth-pcie-3-0, 2017.

[23] Intel, "Intel Data Direct I/O (DDIO)," https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2018.

[24] Intel, "Data Plane Development Kit (DPDK)," https://www.dpdk.org, 2018.

[25] Intel, "Intel Data Direct I/O Technology (Intel DDIO): Technology Brief," https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html, 2012.

[26] Intel, "Intel Xeon Platinum 8260 processor," https://ark.intel.com/content/www/us/en/ark/products/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz.html, 2019.

[27] Intel, "Intel Optane SSD DC P4800X Series," https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-ssd-series/optane-dc-p4800x-series/p4800x-375gb-2-5-inch-20nm.html, 2017.

[28] Intel, "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, 2018.

[29] J. Axboe, "Flexible I/O tester (fio)," 2016.

[30] T. Macedo and F. Oliveira, *Redis Cookbook: Practical Techniques for Fast Data Manipulation*. " O'Reilly Media, Inc.", 2011.

[31] 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*. ACM, 2010.

[32] brianfrankcooper, "YCSB core workloads," https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2017.

[33] DPDK, "L3 forwarding sample application," https://doc.dpdk.org/guides/sample_app_ug/l3_forward.html, 2019.

[34] K. Wiles, "The Pktgen application," https://pktgen-dpdk.readthedocs.io/en/latest/, 2019.

[35] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor-A better way to measure CPU utilization," https://software.intel.com/en-us/articles/intel-performance-counter-monitor, 2012.