DDIOSim: A Microarchitecture Simulator for Data Direct I/O Technology

Hari Sharan, Mythili Vutukuru, Biswabandan Panda
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India
{harisharan, mythili, biswa} @cse.iitb.ac.in

***Abstract*—This paper presents DDIOSim, a cycle-accurate microarchitecture simulator that simulates Data Direct I/O-based network packet processing. Our open-source simulator consists of a front-end trace generator that generates traces of CPU instructions, memory accesses, and network I/O events across multiple networking applications, and a cycle-accurate backend simulator that processes these traces by simulating DDIO operations along with the entire CPU and cache/memory hierarchy. Our simulator can be used to explore various DDIO design and configuration options, and its interactions with other microarchitecture optimizations like cache hierarchy, hardware prefetchers, and DRAM schedulers.**

I. Introduction

Recent advancements in networking technology have led to a rise in network link speeds to 400Gbps and beyond [1]. A faster network link demands faster packet processing at the CPU. Traditionally, data from network I/O devices is transferred to the main memory through direct memory access (DMA) first, and the CPU later fetches the data from DRAM to process. However, this method is inefficient for high throughput network I/O devices (100s of Gbps) due to high DRAM latency and limited DRAM bandwidth. Intel's Data Direct IO (DDIO) [2] is a recent technology that directly transfers I/O data from the network interface card (NIC) into the last-level cache (LLC), resulting in lower access latency and higher throughput for network packet processing.

In order to explore the hardware-level design options for technologies like DDIO, the research community needs a microarchitecture simulator that can simulate the network packet processing done by the OS kernel, along with *faithfully* simulating the behavior of hardware, including DDIO-enabled NICs, CPU, caches, and DRAMs. While there has been a proliferation of academic and commercial simulators [3]–[6], there is no *open-source* simulator that can simulate the end-to-end packet processing of DDIO, across the hardware and software layers.

To help fill this void, we present DDIOSim, a cycle-accurate simulator that simulates DDIO-enabled packet I/O across both the NIC hardware and OS network stack, along with a faithful simulation of DDIO interaction with the processor and memory hierarchy. DDIOSim is built over two existing simulators, QSim [7] and ChampSim [4]. QSim is a tracing wrapper around the QEMU virtual machine that can be used to generate CPU and memory traces, while ChampSim is a microarchitecture simulator that simulates user-level instruction traces generated by binary instrumentation tools like Intel Pin [8]. We extend QSim to work as a front-end trace generator, to generate traces of CPU, memory, and I/O activity for user-level and kernel-level packet processing performed by applications running in the QEMU virtual machine. We then process these traces in the backend, which is an extension of ChampSim that is modified to simulate DDIO-related processing within the processor and cache hierarchy.

Our design choices are informed by our desire to use DDIOSim to explore various DDIO-related microarchitecture design options across a variety of real-life applications. Because DDIOSim is built over a trace-based simulator like ChampSim, we have the benefit of high-speed simulations, along with the flexibility of adding state-of-the-art microarchitecture optimizations in the form of cache management policies, data prefetchers, DRAM schedulers and so on. We can also generate traces for, and simulate a variety of applications by running them within the QEMU virtual machine of QSim. Experiments with DDIOSim can help us revisit microarchitecture decisions and memory hierarchy configurations for various applications, while keeping DDIO in mind. We believe that DDIOSim is a useful tool for computer architecture and networking researchers that want to include DDIO-based microarchitecture optimizations in their research.

II. Background & Related work

A. Intel Data Direct I/O (DDIO)

Intel DDIO, introduced with Intel Xeon processor E5 family [2], makes the last level cache (LLC) the primary destination and source of network I/O data rather than main memory, helping to deliver increased bandwidth, lower latency, and reduced power consumption. Traditionally, on the arrival of an incoming packet at the NIC, it writes the data to main memory (DRAM) using direct memory access (DMA). When the packet is scheduled for processing, it is fetched into the cache hierarchy from DRAM. With Intel DDIO, the NIC can directly write data to LLC, avoiding multiple accesses to main memory, reducing latency and memory bandwidth demand.

DDIO is limited to 10% of LLC. The LLC space used by DDIO cannot be further partitioned using Intel CAT [9] and is shared among the multiple co-running workloads. DDIO has two operating modes for an incoming I/O write operation. In Write Update mode (destination address is an LLC hit), the cache line is overwritten with new data whereas in Write Allocate (destination address is an LLC miss) mode, a cache line is allocated in LLC and no trips to memory are needed.

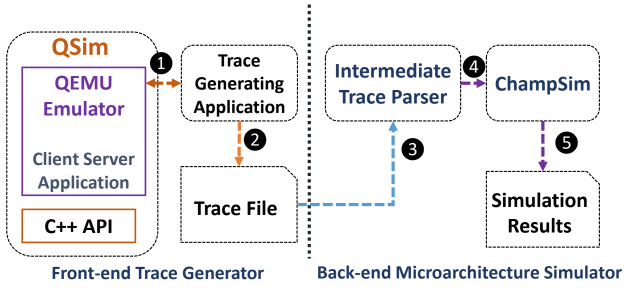
QSim
QEMU Emulator
Client Server Application
C++ API
1
Trace Generating Application
2
Trace File
Intermediate Trace Parser
3
4
ChampSim
5
Simulation Results
Front-end Trace Generator
Back-end Microarchitecture Simulator

Fig. 1: Overview of DDIOSim

The default configuration for Intel DDIO uses two LLC ways. However using the Model Specific Register (MSR) - "IIO LLC WAYS", more LLC ways can be assigned to DDIO [10].

B. QSim and ChampSim

QSim [7] is a thread-safe multicore emulation library based on the QEMU emulator [11], which provides instruction-level control of the emulated environment. It boots a modified Linux kernel inside QEMU emulator and provides callbacks for extracting detailed information about the executing instruction stream from QEMU. It is implemented as a library with a C++ API, and manages a collection of instances of a modified QEMU CPU emulator. Each instance contains its own set of global variables, including the translation cache and CPU state, but shares a common host process, guest RAM state, and QSim callback pointers. The API allows control of the QEMU CPU emulator at an instruction level using callbacks that are invoked when an event of interest occurs in QEMU.

QSim modifies QEMU helper functions to invoke callbacks registered by the user, which can be used to log instructions, memory accesses, interrupts etc. The instructions callbacks provide opcode, CPU mode, decoded instruction and register values. The memory callbacks provide access type(read/write), guest virtual, and guest physical / host virtual addresses. The key advantage of using QSim over other front-end emulators such as Intel PIN tool [8] is that it executes a Linux kernel and allows tracing of OS kernel code execution along with user code execution, which is essential to capture network stack processing that is critical in understanding DDIO performance.

ChampSim [4] is a recent trace-based microarchitecture simulator that has been used extensively for microarchitecture research and ISCA championships [12], [13]. It models a high-performance out-of-order (OoO) core and memory hierarchy, and enables evaluating microarchitectural ideas for improving problems of microarchitectures. It is written in C++ and uses an instruction trace of an application to simulate the microarchitecture level processing. At the end of simulation, ChampSim provides a number of metrics and statistics to evaluate the microarchitectural ideas and performance such as IPC (Instructions per Cycle), cache hits, and so on. The trace of application execution can be automatically obtained from the Intel PIN tool, or can be taken from other front-end emulators after suitable conversion into the ChampSim format.

C. Related work

Recent research has studied the performance benefits of Intel DDIO using real hardware, and suggested approaches for improving the network performance. Farshin et al. [10] propose tuning of DDIO LLC ways for improving throughput, while Tootoonchian et al. [14] propose tuning the number of RX descriptors. Yifan et al. [15] propose a mechanism to adaptively allocate more/less LLC ways to DDIO depending on the performance status of workloads. All of these papers focus on tuning the configuration parameters of existing DDIO-enabled hardware but do not provide ways to explore the hardware design of DDIO itself. Alian et al. [16] propose to change the DDIO mechanism by writing the packets up to L2C for some class of applications. However the simulator used has not been made public.

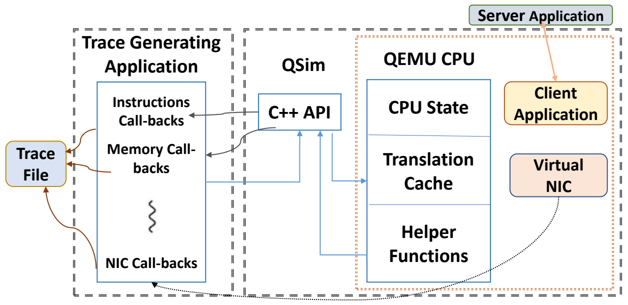

Fig. 2: DDIOSim Front-End Trace Generator

III. DDIOSim Design

We present DDIOSim, a microarchitecture simulator that simulates the behavior of DDIO for different workloads, allowing researchers to explore design options. The simulator has two main components, as shown in Figure 1: a front-end trace generator based on QSim that generates instruction-level traces for network applications, and a backend microarchitecture simulator that runs cycle-accurate hardware simulations of DDIO on the traces generated by the front-end.

A. Front-End: Trace Generator

The front-end trace generator, shown in Figure 2, is based on the QSim trace generator. We use QSim primarily because it allows tracing of OS kernel instructions pertaining to packet processing (in addition to user application instructions), and allows running multiple different types of networking applications inside the Linux OS of the QEMU emulator.

We extend the functionality of the existing QSim simulator to trace the network events in the QEMU virtual NIC, in addition to the CPU and memory events traced already. We modified the QEMU virtual NIC implementation to invoke a QSim callback when a received packet is stored in DRAM, and when a packet to be sent is being read from DRAM. The NIC callbacks provide information about the packet size, packet data, and the address where the packet will be stored in DRAM, which is essential to simulate DDIO behavior. The trace-generating application, written in C++ using the QSim APIs, initializes Qsim and its QEMU emulator and sets up the different callbacks. We run one endpoint of a client-server network application whose network I/O we wish to trace inside the QEMU emulator, with its corresponding peer running outside the emulator in the host. When the tracing

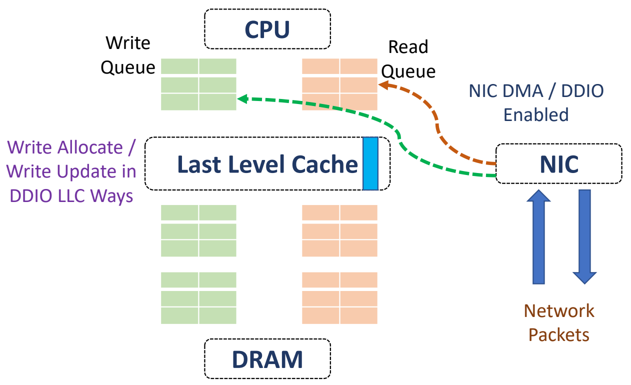

Fig. 3: DDIO implementation in ChampSim

mechanism is enabled, the different callbacks triggered by the network application in QSim are received by the trace-generating application, which then writes them to a trace file.

B. Backend: Microarchitecture Simulator

Our backend simulator is an extension of the open-source ChampSim simulator. ChampSim is a trace-based simulator that ingests an input trace containing a lot of different information about the executing instruction, such as branch/non-branch, source/destination registers, and memory references. Therefore, we write an intermediate trace parser in C++ to first convert the trace file generated by QSim into the format acceptable to ChampSim. This intermediate parser parses the QSim trace, and classifies each entry into branch/non-branch instruction, memory access, or network packet read/write. For branch instructions, it determines if the branch is taken, which is useful for the branch predictor of ChampSim. It extracts the registers and memory addresses and then generates the complete memory reference addresses in case memory reference addresses have to be evaluated from register values. Network packet read/write events generated by our NIC callback are translated into equivalent ChampSim instructions by extracting the relevant data such as the physical address of the location where NIC does the DMA read/write, size of data, etc. A special instruction id differentiates between CPU instructions and network DMA instructions in the ChampSim simulation. As ChampSim simulates memory accesses on virtual addresses, our intermediate parser translates the physical address of memory accesses into virtual addresses using the information present in memory callbacks. The final trace in ChampSim-compatible format is then given as input to ChampSim.

We now modify the ChampSim simulator to simulate NIC DMA and DDIO behavior, as shown in Figure 3, while ensuring backward compatibility with the original non-networking simulator features. A virtual NIC implementation was added for simulating the NIC DMA process. When an instruction is read from the trace, the network DMA instructions (identified by a special id) are passed on to the virtual NIC instead of going through the CPU pipeline. The virtual NIC parses the NIC DMA instructions of the trace and then DMA's the network packets to LLC or to DRAM, depending on whether DDIO is enabled or disabled. It extracts the packet size and generates an equivalent number of memory references to the read/write queue of LLC/DRAM. The network packet identifiers such as packet number and DMA address are maintained separately to identify accesses to network packets as the packets are being processed by the CPU.

The cache implementation in ChampSim is modified to implement the DDIO mechanism. We reserve two default LLC ways for storing incoming network packets and identifying the network packets in the read/writeback queue. On arrival of packets from virtual NIC when DDIO is enabled, we do "Write Update" or "Write Allocate" [2] operation for storing the packet data in LLC without going to DRAM, and invalidate the address in L1D/L2 caches if it exists. Similar changes were done in the DRAM implementation to identify network packet read/write requests. Considerable changes were also done in the handling of miss status holding registers (MSHR) to handle scenarios where concurrent requests to the same addresses arrive at MSHR, one from CPU and another NIC DMA request from virtual NIC. In the read/write mechanism for both cache and MSHR, the memory accesses from CPU and NIC had to be handled differently as both have different semantics. This is because, for memory access originating from CPU, data has to be returned back to CPU, whereas for an NIC DMA write update/allocate, no such action is required.

The processing of read/write memory references across the memory hierarchy (cache and DRAM) was instrumented to identify memory accesses to network packets when they are stored and then later moved across the memory hierarchy during the CPU packet processing. Identifying a memory access for network packet data was essential to measure the performance parameters of DDIO, e.g., the average memory access latency, and the number of hits/misses for memory accesses to network data. The network throughput for the simulated networking application was calculated based on the number of network packets processed in the simulation and the time taken for completion of the simulation.

The design of our simulator meets requirements that are not met by existing simulators. For DDIO-specific microarchitecture optimizations like designing a better cache hierarchy or prefetchers, we need simulators that can provide cycle accurate simulations of DDIO, processor, and the memory system. Further, all the recent state-of-the-art ideas on caching, prefetching, and cache hierarchy have been proposed using the ChampSim simulator as the platform, and all such source code is publicly available. Therefore, building upon ChampSim allows a fair comparison with prior work for all kinds of workloads, including network intensive workloads through DDIOSim. In terms of speed, our preliminary experiments show that DDIOSim is faster (in terms of simulation speed) than other execution-driven simulators like gem5. For example, gem5 simulates 0.1 MIPS (millions of instructions per second) whereas DDIOSim simulates 1 MIPS. Another reason for using a trace-based but more accurate simulator for microarchitecture study is the availability of industry-generated server traces [17], [18], which cannot be run on gem5 as gem5 needs the source code/binaries.

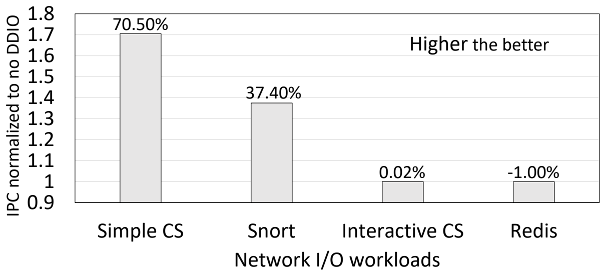

Fig. 4: Speedup with DDIO over no DDIO.

IV. Evaluation

A. Trace Generation

We generated traces for four different types of networking workloads running inside QSim, as described below. These workloads have different network I/O characteristics and were chosen to study the performance gains of DDIO across different classes of applications.

Simple Client Server (CS): A simple TCP server running in the host OS sends data to a TCP client running inside QSim.

Snort: Snort IDS [19] runs in monitoring mode with the above simple client-server application. Snort runs in the background, monitoring the network packets received by the client.

Interactive Client Server (CS): The server running in the host OS waits for clients response before sending next set of data, resulting in a lower rate of packets than with the simple CS.

Redis: Redis server [20] runs inside QSim, receiving bulk GET / SET requests from an application in the host OS.

Metric	Simple CS	Snort	Inter. CS	Redis
NW Tput (Mbps)	27022	18315	761	132
LLC MPKI	15.4	8.24	0.72	2.55
IPC	0.87	1.06	0.94	0.98
Avg DRAM BW (MB/s)	2325	1545	91	363
Avg mem access time (cycles) for all / only loads	60.2 / 26.6	38.9 / 19.1	5.5 / 5.7	10.3 / 8.7

TABLE I: Characteristics of simulated network applications

Core	Out-of-order, bimodal branch predictor, 4GHz with 6 issue width, 4 retire width, 352 entry ROB
TLBs	64-entry ITLB/DTLB, 1536-entry STLB, LRU
L1I cache	32KB 8-way (4 cycles), 8 MSHRs, LRU
L1D cache	48KB 12-way (5 cycles), 16 MSHRs, LRU
L2 cache	512KB 8-way (10 cycles), 32 MSHRs
LLC	2MB 16-way (20 cycles), 64 MSHRs
DRAM	4GB, 2666.6 MT/s, 1 channel
DRAM Controller	1 controller, 64 read/write queues, FR-FCFS

TABLE II: ChampSim simulation parameters

Table I shows the microarchitecture-level characteristics of these applications when simulated without Intel DDIO. The first two workloads can be termed as network I/O intensive applications as they have a high network throughput, while the next two workloads are less I/O intensive with a lower network throughput. These traces, which are representative of different types of network applications, were run through DDIOSim, with the simulation parameters shown in Table II.

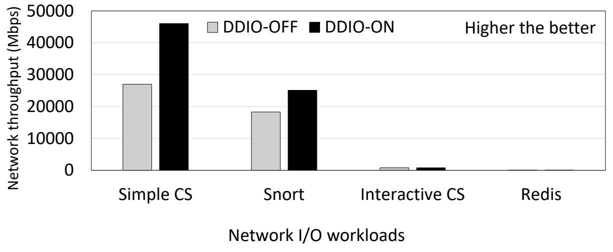

Fig. 5: Network throughput with and without DDIO.

The correctness of implementation of network packet DMA process and DDIO behavior in ChampSim was verified using different statistics generated by ChampSim. The number of packets being processed while generating the trace matched the number of packets being processed while simulating that trace in ChampSim. Similarly, the number of memory accesses to network packets during packet processing approximately matched the number of accesses performed during Linux network stack processing. We also match memory dependencies, register dependencies and timing information between instructions. The incoming network packets are stored in two LLC ways identified for DDIO. The maximum throughput achieved (46 Gbps with DDIO) is comparable to throughput achieved on a multicore server system running Linux [21].

B. Results: Performance Gains with DDIO

We now describe the performance gains observed for the various network applications when DDIO is enabled, as compared to when DDIO is disabled during simulation. The broad conclusions match those reported by DDIO in prior work. These results showcase the usefulness of our simulator in faithfully simulating DDIO mechanisms.

Figure 4 shows the IPC (normalized to baseline case without DDIO) of different network I/O workloads with DDIO. There is significant IPC improvement of up to 70% for network I/O intensive workloads (Simple CS and Snort). However, this increase is not significant in the less network I/O intensive workloads (Interactive CS and Redis) as expected. A similar trend can be seen with network throughput in Figure 5, where the more network I/O intensive applications see significant throughput gains with DDIO enabled, as the performance gains accruing from CPU accessing network data faster from LLC with DDIO matter more for network I/O intensive applications that perform more network packet accesses per unit time.

Figure 6 shows the LLC misses per kilo instruction (MPKI) of different network I/O workloads with and without DDIO, and we see that DDIO reduces LLC MPKI significantly for network I/O intensive workloads. Similar improvements have been observed for DRAM memory bandwidth utilization. These results are expected because DDIO stores the incoming network packets in LLC instead of DRAM, which ensures that the memory accesses for network packet data can be serviced from LLC, leading to a reduction in LLC MPKI and DRAM memory bandwidth utilization.

Figure 7 shows the system-wide average memory access time (AMAT) for memory accesses (including accesses at all

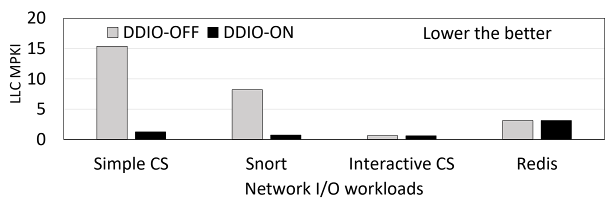

Fig. 6: LLC MPKI with and without DDIO.

levels of memory hierarchy) to network I/O data with and without DDIO enabled. Servicing the network packet data for CPU processing from LLC instead of DRAM leads to a reduction in AMAT as data can be fetched from LLC with lesser latency. We also see that DDIO improves the AMAT for load requests significantly from 61 cycles to 13 cycles for network-intensive applications.

We also ran simulations with increasing LLC cache sizes, and we observed that increasing the size of LLC for the network I/O intensive workloads leads to more performance gains with DDIO. We observed improvements of around 4.3% in throughput/IPC and 33% in AMAT for the Simple CS trace for an LLC size of 16 MB over an LLC size of 2 MB. Improvement in performance with higher LLC size occur because there is more space for DDIO to store incoming packets, and more of the application's working set can fit into the LLC. However, we observe that there is no significant further improvement for LLC sizes beyond 16MB, as the application's working set is able to fit within 16MB.

In summary, our simulation results from DDIOSim show that DDIO improves various performance metrics like IPC, application throughput, AMAT, and cache hit rates, especially for network I/O intensive applications. These results match what is expected of DDIO, as seen from prior published works. These results showcase the correctness of the modeling of DDIO behavior in DDIOSim, and make DDIOSim a useful tool to experiment with various design and configuration options for DDIO in a microarchitecture simulator.

V. Conclusion and Future work

DDIO plays an important role in improving the performance of networking applications. While prior work has characterized the various performance gains of DDIO, there are no open-source simulators available today to experiment with various DDIO design options or understand its interactions with other microarchitecture optimizations. Our work seeks to address this lacuna by developing DDIOSim, a microarchitecture simulator that builds upon QSim and ChampSim to provide a cycle-accurate simulation of DDIO and its interactions with the CPU and cache hierarchy.

Our results show that the simulator helps to characterize the performance gains of DDIO for different types of applications. In the future, we plan to use our simulator to study and test new microarchitecture ideas for further improving the DDIO performance gains. For example, we can study the impact of running multiple CPU and memory-intensive applications along with networking applications, improving existing DDIO mechanisms by writing packets to various levels of cache, interactions between DDIO and hardware prefetchers, and so on. We plan to open-source the simulator code upon publication, in order to benefit the wider research community.

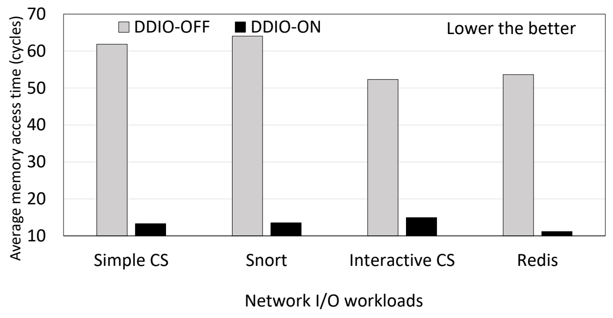

Fig. 7: Average memory access time with and without DDIO.

VI. Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments and suggestions. Special thanks to Parth Sangani and Pulkit Agrawal for their help.

References

[1] "Ethernet roadmap," https://ethernetalliance.org/technology/ethernet-roadmap/, [Accessed 25-Jun-2023].
[2] "Intel Data Direct I/O (DDIO)," https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html, [Accessed 25-Jun-2023].
[3] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.
[4] "Champsim, a trace-based simulator," https://github.com/ChampSim/ChampSim, [Accessed 25-Jun-2023].
[5] R. Ubal *et al.*, "Multi2sim: A simulation framework for cpu-gpu computing," in *PACT 2012*.
[6] D. Wang *et al.*, "Dramsim: A memory system simulator," *SIGARCH Comput. Archit. News*, 2005.
[7] C. D. Kersey *et al.*, "A universal parallel front-end for execution driven microarchitecture simulation," in *RAPIDO 2012*.
[8] "Intel Pin tool," https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html/, [Accessed 25-Jun-2023].
[9] "Intel Resource Director Technology (RDT)," https://www.intel.in/content/www/in/en/architecture-and-technology/resource-director-technology.html, [Accessed 25-Jun-2023].
[10] A. Farshin *et al.*, "Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks," in *USENIX ATC 2020*.
[11] F. Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, 2005.
[12] "2nd cache replacement championship," https://crc2.ece.tamu.edu/, [Accessed 25-Jun-2023].
[13] "3rd data prefetching championship," https://dpc3.compas.cs.stonybrook.edu/, [Accessed 25-Jun-2023].
[14] A. Tootoonchian *et al.*, "ResQ: Enabling SLOs in network function virtualization," in *15th USENIX Symposium, NSDI 2018*.
[15] Y. Yuan *et al.*, "Don't forget the I/O when allocating your LLC," in *ACM/IEEE ISCA 2021*.
[16] M. Alian *et al.*, "IDIO: network-driven, inbound network data orchestration on server processors," in *MICRO 2022*.
[17] "Google workload traces," https://dynamorio.org/google_workload_traces.html, [Accessed 25-Jun-2023].
[18] "Championship value prediction traces," https://perscido.univ-grenoble-alpes.fr/datasets/DS382, [Accessed 25-Jun-2023].
[19] "Snort, network intrusion detection & prevention system," https://www.snort.org/, [Accessed 25-Jun-2023].
[20] "Redis, open source in-memory key-value store," https://redis.io/, [Accessed 25-Jun-2023].
[21] Q. Cai *et al.*, "Understanding host network stack overheads," in *ACM SIGCOMM 2021 Conference*.