2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00042

# IDIO: Network-Driven, Inbound Network Data Orchestration on Server Processors

Mohammad Alian[1], Siddharth Agarwal[2], Jongmin Shin[3], Neel Patel[1], Yifan Yuan[2], Daehoon Kim[3], Ren Wang[4]
Nam Sung Kim[2]
[1]*University of Kansas,* [2]*University of Illinois, Urbana-Champaign,* [3]*DGIST,* [4]*Intel Labs*

***Abstract*—High-bandwidth network interface cards (NICs), each capable of transferring 100s of Gigabits per second, are making inroads into the servers of next-generation datacenters. Such unprecedented data delivery rates impose immense pressure, especially on the server's memory subsystem, as NICs first transfer network data to DRAM before processing. To alleviate the pressure, the cache hierarchy has evolved, supporting a direct data I/O (DDIO) technology to directly place network data in the last-level cache (LLC). Subsequently, various policies have been explored to manage such LLC and have proven to effectively reduce service latency and memory bandwidth consumption of network applications. However, the more recent evolution of the cache hierarchy decreased the size of LLC per core but significantly increased that of mid-level cache (MLC) with a non-inclusive policy. This calls for a re-examination of the aforementioned DDIO technology and management policies.**

**In this paper, first, we identify three shortcomings of the current static data placement policy placing network data to LLC first and the non-inclusive policy with a commercial server system: (1) ineffectively using large MLC, (2) suffering from high rates of writebacks from MLC to LLC, and (3) breaking the isolation between application and network data enforced by limiting cache ways for DDIO. Second, to tackle the three shortcomings, we propose an intelligent direct I/O (IDIO) technology that extends DDIO to MLC and provides three synergistic mechanisms: (1) self-invalidating I/O buffer, (2) network-driven MLC prefetching, and (3) selective direct DRAM access. Our detailed experiments using a full-system simulator — capable of running modern DPDK userspace network functions while sustaining 100Gbps+ network bandwidth — show that IDIO significantly reduces data movement (up to 84% MLC and LLC writeback reduction), provides LLC isolation (up to 22% performance improvement), and improves tail latency (up to 38% reduction in 99$^{th}$ latency) for receive-intensive network applications.**

***Keywords*-Non-inclusive Cache; DDIO; Datacenter Network;**

## I. Introduction

The evolution of networking technology has led to network bandwidth in servers approaching memory bandwidth and proper handling of network traffic across memory hierarchy can noticeably affect overall performance. Direct Cache Access (DCA) [19] was introduced to address this challenge by allowing the NIC to directly write inbound network data into the cache hierarchy instead of the main memory (DRAM) first [24]. As such, DCA not only reduces service latency and memory bandwidth consumption of network applications but also improves the performance of co-running applications with reduced interference between the network and co-running applications at the memory subsystem. The current implementation of DCA by Intel and ARM CPUs are known as Data Direct I/O (DDIO) [1] and Cache Stashing [7], respectively; AMD also implemented a DCA technology in its Zen 4 microarchitecture [30] and they will be collectively referred to as DDIO in this paper.

DDIO generally works well, but it has shortcomings arising from the fact that all applications using the network share the same limited number of LLC ways [35], [41], [36]. Although the number of ways allocated for DDIO is dynamically configurable [41], the precious shared LLC space should be conserved for the co-running applications. Furthermore, starting with the Skylake microarchitecture, Intel Xeon CPUs, used predominantly in server platforms, introduced drastic changes to the cache hierarchy, especially for datacenter applications. Compared to the previous microarchitecture (Broadwell), Skylake decreased the size of shared LLC from 2.5MB to 1.375MB per core in favor of increasing the size of the private middle-level cache (MLC) from 256KB to 1MB per core. As the LLC's role was demoted to being a victim cache for large MLCs, its inclusion policy changed from inclusive to non-inclusive, and its associativity shrunk from 20 to 11 ways. Although this drastic change provides stronger inter-core isolation and higher performance for some classes of datacenter applications such as database [11], it increases pressure on the shared LLC by applications and mechanisms like DDIO that make heavy use of the shared LLC space.

Through detailed analysis of the cache activity using a commercial system based on the Intel Skylake microarchitecture, we identify three other **S**hortcomings for the DDIO technology in a non-inclusive cache hierarchy. **(S1)** The baseline DDIO — or even dynamic DDIO policies [41] — does not take advantage of the large MLC. **(S2)** The non-inclusive cache hierarchy suffers from a high rate of writebacks from MLC to LLC by dead cachelines in high bandwidth network processing. **(S3)** The non-inclusive cache hierarchy can break the isolation enforced by limiting the DDIO ways between I/O and application data, causing *DMA bloating* phenomenon.

To alleviate the aforementioned three shortcomings, (S1)–(S3), we propose **I**ntelligent **D**irect **I/O** (IDIO) technology, a next-generation DDIO technology that implements three **M**echanisms. **(M1) Self-invalidating I/O buffers:**

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00042

We extend the current cache maintenance instructions to implement a cache invalidate instruction that drops the cacheline without writing it back to a lower level cache. This instruction is executed by the application after a DMA buffer data is consumed by the application (software stack). **(M2) Network-driven MLC prefetching:** We propose a network-driven prefetcher that is triggered by monitoring the network activity and synergistically works with self-invalidating buffers to minimize the LLC pressure. **(M3) Selective direct DRAM access:** Although DCA improves system performance by eliminating DRAM accesses for network RX, it fails to provide any benefit when the use distance of RX data is high, resulting in premature eviction of the RX buffers to DRAM. We propose a selective direct DRAM access mode, where DCA is disabled for the payload of the RX packets of different application classes.

We enabled the gem5 simulator [27] to run modern DPDK userspace network functions and sustain 100Gbps+ network bandwidth. Our detailed experiments using full-system gem5 show that IDIO significantly reduces on- and off-chip data movement (up to 84% MLC and LLC writeback reduction), provides LLC isolation (up to 22% performance improvement), and improves tail latency (up to 38% reduction in 99$^{th}$ latency) for receive-intensive network applications.

## II. Background

### A. Network Stack and Memory Hierarchy Evolution: Data Movement Perspective

**Network data movement in a computer system:** Computer systems traditionally use direct memory access (DMA) technology with memory-mapped I/O (MMIO) to transfer network data between NIC and CPU. The CPU allocates DMA ring buffers in the main memory for data to be received/transmitted from/to the NIC. The NIC's DMA engine copies data from the buffer to the NIC (TX) or from the NIC to the buffer (RX) without CPU involvement. When the DMA copies are completed, the NIC sends interrupts to notify the CPU of RX and TX completion. The notification can be implemented using a polling mode driver (PMD) as well. In the case of RX, the CPU reads the newly arrived data in the DMA ring buffer for processing.

**Data direct I/O.** Since memory bandwidth becomes a performance bottleneck in high-speed networking technologies [9], direct cache access (DCA) [19] has been proposed to improve the network performance by allowing the CPU to write/read data directly to/from the LLC, bypassing DRAM. Most modern CPUs from different vendors support DCA. For example, Intel's Xeon CPUs employ a DCA implementation called data direct I/O (DDIO) technology. DDIO directly uses the CPU's LLC for data communication between I/O devices and CPUs instead of going through DRAM, reducing both access latency and memory bandwidth consumption considerably [6], [21], [35], [22], [15], [14], [41], [39], [10], [28], [32], [5].

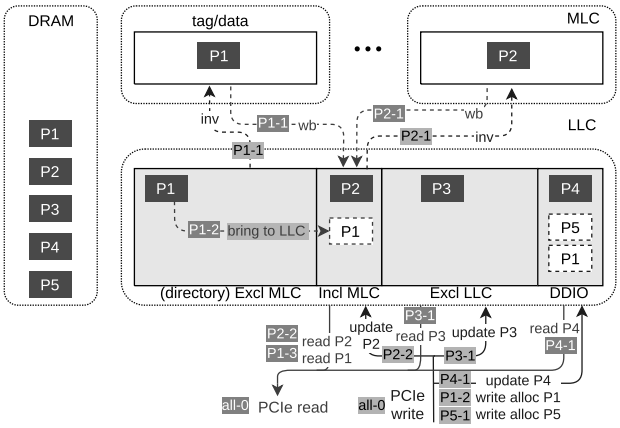



Figure 1: Data movement within a non-inclusive cache hierarchy for PCIe write and PCIe read. 'P' denotes packet.

**Recent evolution in server CPU cache hierarchy:** Starting from the Skylake microarchitecture, Xeon CPUs, used predominantly in server platforms, introduced drastic changes to the cache hierarchy. Compared to the previous microarchitecture (Broadwell), Skylake radically re-allocated on-chip cache resources by decreasing shared LLC capacity (from 2.5MB to 1.375MB per core) in favor of quadrupled private L2 capacity (256KB to 1MB). As the LLC's role was demoted to being a victim cache for large L2s, *its inclusion policy changed from inclusive to non-inclusive*, and its associativity shrunk from 20 to 11 ways. Although this drastic change facilitates inter-core isolation, it increases pressure on workloads and mechanisms (like DDIO) that heavily use shared LLC space.

**DDIO data movement in the non-inclusive cache hierarchy:** Figure 1 shows four parts of the LLC in a non-inclusive Skylake CPU series [36], [40]. *Excl MLC* is the directory that holds the tags of the valid MLC-resident cachelines, used to filter coherence activity sent to MLCs. Two out of 11 LLC ways store cachelines with a valid MLC copy (*Incl MLC* in Fig.1). The *Excl LLC* and *DDIO* ways in the figure store cachelines exclusively in LLC. In the default configuration for PCIe devices, like NICs, 2 out of 11 ways of LLC are used by DDIO to write-allocate device writes that miss in the LLC.

A PCIe read or write request may find the target address in five locations in the memory hierarchy. Blue rectangles P1–P5 in Fig.1 represent packets that reside in these five locations. *P1* is exclusively in the MLC. *P2* is in both MLC and LLC. P3 is exclusively in the non-DDIO ways of LLC. P4 is exclusively in DDIO ways of LLC. P5 is not cached. Note that all packets are backed by DRAM.

On the data egress path, when a PCIe read request is received (step all-0 in Fig.1), if the cacheline is in MLC (i.e., P1 or P2) and dirty, the cacheline will be written back to LLC ( P1-1 and P2-1 steps) and then sent to the requesting device [36]. If the requested cacheline is in LLC (i.e., P3 or

P4), the data will be read from LLC and sent to the device ( P3-1 and P4-1 steps). If the cacheline is not cached (i.e., P5), the data is read directly from DRAM, like conventional DMA.

On the data ingress path, when a full cacheline PCIe write request is received[1] (step all-0 in Fig.1), if the cacheline is in MLC (i.e., P1 or P2), it will be invalidated (P1-1 and P2-1 steps). Next, if the cacheline was exclusively in MLC (i.e., P1), then a cacheline is allocated in the LLC's DDIO ways and gets updated with the PCIe write data (step P1-2). A cacheline already present in LLC (i.e., P2 or P3) is directly in-place updated (steps P2-2 and P3-1). Finally, if the write address is not cached (i.e., P5), it will be write-allocated in the DDIO ways (step P5-1).

### B. Demystifying Network Applications

The efficiency of inbound network data placement policies highly depends on how the network applications consume the RX data. Although most network applications use a circular ring buffer to communicate with the NIC [35], [16], from the I/O device's standpoint, the reusability of shared CPU/NIC buffers depends on how the application recycles these buffers. We classify I/O buffer recycling into three **M**odes. **(M1) Copy**: The consuming application (or network stack) copies the received packets from the ring buffer to the application space and processes the copied packets later. This is how the Linux software stack works. **(M2) Re-allocate :** The consuming application stashes the pointers to buffers holding received but not yet processed packets and replenishes the ring buffer by updating its pointers to different DMA buffers. The application later uses the stashed pointers to process the packet contained in the corresponding buffers. This mode is used within the Linux kernel to reduce memory copies for large packets. **(M3) Run to completion:** The application processes the received DMA buffers in place and only frees them after application-level processing is completed. This prevents context switches and memory copies and achieves more predictable latency for latency-sensitive, network-intensive applications (e.g., Network Functions). The DMA buffers are reused throughout application execution. Latency-sensitive applications using DPDK often follow this approach.

To study the life cycle of a DMA buffer in the memory hierarchy while it is being processed, let's consider the application domain of Network Functions (NFs) as a prime example of latency-sensitive, network-intensive applications. A significant high-level distinction in packet processing characteristics across applications in that family is whether the NF performs *shallow* or *deep* packet inspection. The former category represents NFs that only inspect and operate on each packet's header (e.g., L2/L3 forwarders, basic firewalls, NAT, load-balancers) [17]. The latter category represents NFs that perform deep packet inspection (e.g., for intrusion detection or general client-server networking applications) and thus access each packet in its entirety.

[1]We only consider full PCIe writes in Fig.1 as the DMA write requests are mostly full cacheline writes.

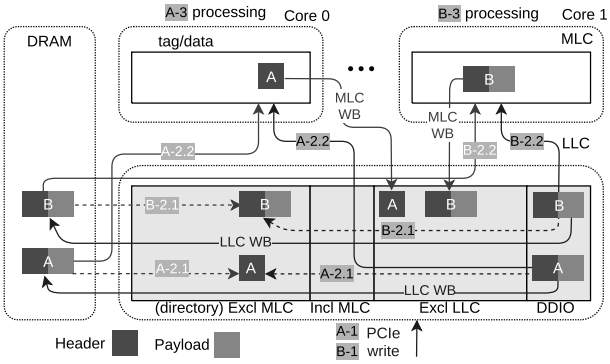



Figure 2: Data movement within a non-inclusive cache hierarchy for application operating on data received from network.

Fig.2 demonstrates the different data movement patterns the two application categories result in. Applications **"A"** and **"B"** refer to the shallow and deep category, respectively. We assume that the DMA buffer is already in the LLC's DDIO ways once a PCIe write arrives. A packet arrival results in a PCIe write that updates the DMA buffer's corresponding cachelines in LLC (A-1 and B-1 in Fig.2). If the time between the arrival of PCIe write and demand miss from the MLC is long, then the cachelines might get evicted to DRAM due to LLC contention ("LLC WB" in Fig.2). In case of an LLC WB, once a demand miss is received for application A, the packet header will be fetched from DRAM, its tag will be allocated in the inclusive LLC directory (step A-2.1) and its data will be allocated in core 0's MLC (step A-2.2). For application B, both header and payload will be requested by core 1 for processing (steps B-2.1 and B-2.2). If there is no LLC WB, then the data will be read from LLC (steps A-2.2 and B-2.2) and its tag will be moved to the directory (steps A-2.1 and B-2.1). Next, the applications process the fetched data (steps A-3 and B-3). Once the DMA buffer is processed by the core, it will stay in MLC until it is evicted to LLC/DRAM ("MLC WB" in Fig.2) or invalidated upon reuse by the NIC (i.e., another PCIe write to the same address). By the time a cacheline's MLC WB occurs, it is likely that the cacheline has already been evicted from the LLC. In that case, the cacheline evicted from the MLC will be allocated inside the non-DDIO ways of LLC, effectively increasing the footprint of DMA data in the LLC. We call this effect *DMA bloating*.

### C. Ethernet Flow Director

Intel Ethernet Flow Director [2] is a feature of Intel NICs that directs incoming packets from the NIC to the core on which the consuming application is running, avoiding

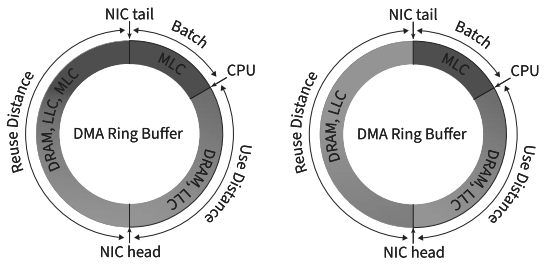



Figure 3: DMA buffer residency in a non-inclusive memory hierarchy throughout the buffer's life cycle when running (left) a general network application; (right) a zero-copy shallow network function. Red indicates buffers residing in the MLC.

unnecessary packet indirections. Flow Director comes in two flavors: Externally Programmed (EP) and automated Application Targeting Routing (ATR). EP mode allows users/programmers to manually set the flows, therefore comes in handy when an application can be pinned to a physical core. On the other hand, ATR dynamically learns the target core by populating a Filter Table with destination core numbers. The Filter Table has up to 8k entries in modern Ethernet adapters. Flow Director calculates the Filter Table index value by hashing the incoming packets' headers and accessing the corresponding Filter Table entry to retrieve each packet's destination core. IDIO builds on top of Flow Director's mechanism.

## III. DDIO Limitations

DDIO employs a static data steering policy in which it always places incoming PCIe writes in the LLC, as detailed in Sec.II-A. This static data steering has two drawbacks: (1) it does not leverage the exclusive MLC space to reduce the LLC contention, (2) regardless of the application access pattern (c.f., Sec.II-B), the entire packet is always placed in LLC, which can cause interference in the LLC (both for the network application itself and co-running applications). These drawbacks result in leaky DMA and latent contender issues detailed in prior work [35], [41], [36]. In this section, we discuss a new problem: the writeback of consumed DMA cachelines from MLC to LLC and DRAM. These cachelines are dead and there is no point in keeping their data inside the memory hierarchy. We show that these writebacks are a side effect of DDIO implementation, application access pattern to DMA buffers, non-inclusive cache hierarchy, CPU processing rate, and network RX rate.

**Observation (1): MLC-resident DMA buffers are invalidated upon reuse by the NIC.** Fig.3 shows the residency of DMA buffers in the memory hierarchy throughout their life cycle. For illustration purposes, we assume that descriptors and DMA buffers are one structure. In reality, the ring buffer holds 128-Byte descriptors where each descriptor stores metadata of RX packets, including a pointer to a

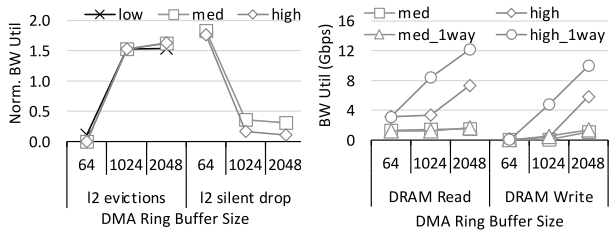



Figure 4: MLC and DRAM Leaks at various load levels and DMA ring buffer size.

DMA buffer. Another caveat is that DMA buffers are not necessarily allocated at consecutive physical memory addresses, as might be perceived from Fig.3. However, this simple illustration is useful to clarify the memory hierarchy's behavior while running network applications.

As shown in Fig.3 and detailed in Sec.II-A, received DMA buffers initially reside inside the LLC. In the figure, the most recently received DMA buffer is tracked by the *NIC head* pointer. Then, each RX DMA buffer will stay inside LLC or be written back to DRAM until the CPU demands it for processing (CPU pointer in Fig.3). To amortize the interrupt/polling overhead and improve cache locality, network applications process RX packets in batches. For example, DPDK's default batch size is 32 packets. In a run-to-completion software stack, the entire batch will be inside MLC for processing. After processing a batch, the CPU moves the *NIC tail* pointer, freeing up the processed DMA buffers. Note that in zero-copy shallow network functions, after processing a batch, the cachelines will reside in the LLC because PCIe reads from the NIC on the egress (TX) path will invalidate MLC copies and bring them back to LLC (Fig.3(right)). In a general network application, DMA buffers will remain inside MLC after being processed by the CPU until they get evicted to LLC (and from there get evicted to DRAM if there is LLC contention) (Fig.3(left)).

**Observation (2): Non-inclusive memory hierarchy can cause a high rate of MLC writebacks.** To emulate the general DMA buffer usage illustrated in Fig.3(left), we run 10 instances of the `TouchDrop` DPDK application on a server with 2×100Gbps Ethernet ports at various load levels. `TouchDrop` receives 1514-byte packets, touches their entire data, and drops them. Refer to Sec.VI for more information about our experimental setup.

In an ideal scenario, an RX buffer remains in LLC until the core consumes it and remains in MLC until being reused by the NIC. Fig.4(left) shows MLC writebacks, and MLC invalidations (due to a PCIe write) rate normalized to the RX network bandwidth at various DMA ring buffer sizes and load levels. Fig.4(right) shows the DRAM bandwidth numbers. *low*, *med*, and *high* load levels correspond to 8Mbps, 1Gbps, and 20Gbps steady RX rates, respectively. As expected, when the ring buffer size is 64, the normalized MLC writeback rate is low, and we have a high rate of MLC

invalidations. However, with DPDK's default ring buffer size of 1024 entries, regardless of load level, the MLC writeback rate increases to ∼1.52× of the RX network BW. As shown, the MLC writeback rate linearly increases with the network BW. Note that writebacks from the MLC cause interference in the LLC since the evicted cachelines need to be allocated in the non-inclusive LLC. MLC writebacks are inevitable once the DMA ring buffer size exceeds MLC size. For instance, when receiving 1514-Byte packets, ring buffers sized larger than 692 exceed the 1MB MLC size (1MB $<$ 692 $\times$ 1514B) and will inevitably experience such writebacks. MLC writebacks take place regardless of the network RX bandwidth and linearly increases with network rate.

**Observation (3): Non-inclusive cache hierarchy increases the aggregate cache capacity for I/O data.** Interestingly, LLC writebacks (when the same network application causes eviction of the previously received data to DRAM) are rarely seen in a non-inclusive cache hierarchy. As shown in Fig.4(right), we do not observe LLC writebacks even if the aggregate DMA ring size of all NFs exceeds the capacity of the DDIO ways (6.5MB DDIO ways vs. 15MB DMA aggregate ring buffer size for 10 NFs with 1024 ring size). There are two reasons: First, in a non-inclusive cache hierarchy, DMA data is split between MLC and LLC, increasing the effective cache capacity of the processor (red and gray parts of Fig.3). Second, as discussed in Fig.2, after an MLC writeback, the cacheline is no longer classified as I/O data and can be allocated in any of the LLC's ways, including non-DDIO ways. As a result, I/O data can now occupy the entire LLC, a phenomenon we call *DMA bloating*. We confirm DMA bloating by using LLC way-partitioning to limit `TouchDrop` applications to a single LLC way (*_1way* configuration in Fig.4(right)) and see 12.3× and 1.7× higher DRAM write BW at high load for 1024 and 2048 ring buffer sizes, respectively.

**Observation (4): Writeback of consumed DMA buffers causes unnecessary on-chip traffic and LLC evictions.** Another interesting observation from Fig.4(right) is that LLC writebacks are more pronounced at high load levels. Note that the working set size of our simple `TouchDrop` application does not change with load level as it is always equal to the DMA ring buffer size. The only thing that changes with load is the distance between NIC and CPU pointers in the ring buffer (Fig.3). Our hypothesis is that at higher loads, as the CPU lags behind the NIC for processing packets, the *Use Distance* increases, and thus LLC pressure mounts. This results in more LLC writebacks at higher loads, especially on bursty request arrivals at NIC.

Figure 5 shows MLC and LLC writebacks when a burst of packets is received. Results are based on a simulation setup detailed in Sec.VI, as it is not possible to perform such fine-grained analysis on a real hardware setup. Note that we scale down the LLC size in gem5 to 3MB and run only

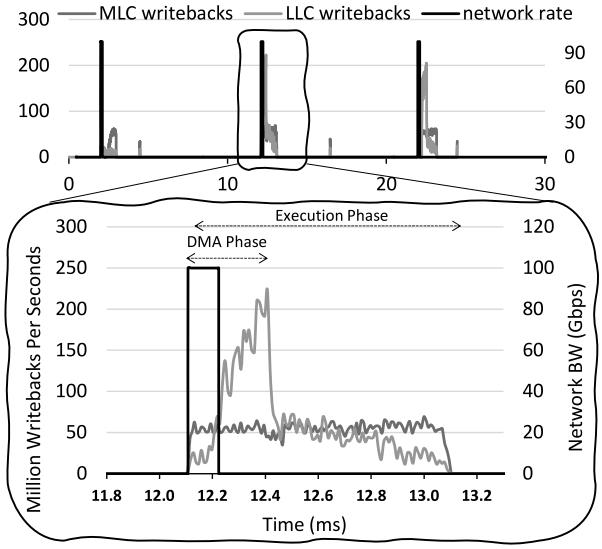



Figure 5: MLC and LLC writebacks when processing bursty network traffic using `TouchDrop`, with 1024 DMA ring buffer size receiving 1514 Byte packets.

two `TouchDrop` instances. The top graph shows writebacks over a timeline of 30 ms, and the bottom one zooms into the second burst. The writeback timeline illustrates two phases in processing RX packets from the DMA buffer:

*DMA Phase: Burst is received at NIC, and NIC starts the DMA transfer to LLC.* There is a lag of several µs from the start of the RX burst, and experiencing high misses in the LLC. In this phase, the NIC head pointer moves much faster than the CPU pointer, and the *Use Distance* in Fig.3 increases. Therefore, after some DMA transfers, the LLC DDIO ways become full, and we start experiencing high LLC writebacks (DMA leak).

*Execution Phase: Packet processing starts on the CPU.* When the CPU starts processing RX packets, the recently received packets are brought into MLC for processing. In this phase, the gap between the CPU pointer and NIC head pointer (Fig.3) reduces. The overlap between DMA and processing phases depends on the network RX rate, batch processing mechanism in the software stack, and CPU processing rate. The primary reason for LLC writebacks during the processing phase in Fig.5 is the writebacks caused by replacing consumed DMA buffers in the MLC. The data written back from MLC to LLC (and possibly DRAM) is dead and not used by `TouchDrop` applications anymore. As we move towards the end of burst processing, the number of LLC writebacks decreases due to the DMA bloating phenomenon discussed earlier. This observation motivates our proposal for self-invalidating I/O buffers detailed in Sec.IV-A.

**Observation (5): RX DMA buffer residency is correlated to the network rate.** Figure 5 reveals a strong correlation between MLC/LLC writebacks and network RX rate. We

leverage this correlation to develop a dynamic network-driven DMA buffer steering policy in IDIO to proactively prefetch cachelines into MLC to prevent the high LLC miss rates encountered in the DMA phase of processing bursty RX network traffic (Sec.V).

## IV. Mitigating MLC/LLC writebacks and DMA bloating

Leveraging the observations made in Sec.III, we propose three techniques for mitigating MLC/LLC writebacks and the detrimental effects of DMA bloating on performance.

### A. Self-Invalidating I/O buffers

DMA buffers are allocated based on the MTU packet size, which is 1514 Bytes for Ethernet; thus, the minimum DMA buffer entry size is often 2KB. Once an RX DMA buffer is processed, it becomes a dead buffer and sits inside the MLC until it gets evicted into LLC and DRAM. Such evictions are wasteful because DMA buffers will be overwritten by the next received packet using several full/partial PCIe write transactions. Note that even when there is a partial PCIe write transaction, the non-overwritten part of the cacheline is still part of the current DMA buffer and is never used. Therefore, it is safe to just invalidate the DMA buffers after they are consumed by the software stack instead of writing them back to LLC and/or memory.

The challenge lies in determining when a DMA buffer is dead and thus safe to invalidate. The hardware cannot determine that moment in time on its own, as it solely depends on the buffer usage behavior of the running software stack. For example, if the RX DMA buffers are copied to a new buffer before processing them, then it is safe to invalidate the cachelines that belong to the DMA buffer after the first touch by a core or copy engine. However, if the running application is a zero-copy network function, then the hardware cannot know the exact time that the NF is finished using the RX DMA buffers.

We propose a cross-layer solution to address this challenge. Since there is no existing hardware knob that the software can use to invalidate a cacheline without writing it back to memory, we propose a new instruction with these semantics. It is the software's responsibility to explicitly invalidate the RX DMA buffers after using them. As full PCIe cacheline write latency is lower when the cacheline is not present in any of the MLCs, we want the software stack to self-invalidate right after the RX data is consumed.

### B. Network-Driven MLC Prefetching

As detailed in Sec.III, there is a strong correlation between network RX rate, demand misses, and MLC and LLC writebacks. We propose a network-driven MLC prefetching technique that synergistically works with self-invalidating I/O buffers to alleviate the LLC pressure at the DMA phase of RX packet processing (Fig.5). More specifically,

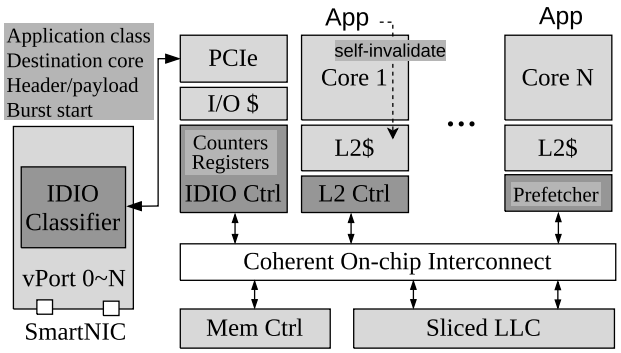



Figure 6: IDIO overall architecture.

we augment the NIC with logic that detects the arrival of a burst and relays the information to the IDIO cache controller to enable MLC prefetching. MLC prefetching is disabled once the DDIO ways and MLC ways reach an equilibrium state, and prefetching is not useful anymore (Sec.V-B). With self-invalidating I/O buffers, as the NIC tail pointer in Fig.3 moves, the MLC-resident buffers get invalidated, freeing up MLC capacity. Once a burst is received, our MLC prefetching uses this available MLC capacity to bring recent RX packets to MLC and free up LLC (i.e., DDIO ways) space.

### C. Selective Direct DRAM Access

Although Direct Cache Access (DCA) improves system performance by eliminating DRAM accesses for network RX, it fails to provide any benefit when the *use distance* of RX data is high, resulting in premature eviction of the RX buffers to DRAM. In such application scenarios, DCA causes contention in LLC and can hurt the performance of collocated applications. We propose a hardware/software solution for selective direct DRAM access mode, where DCA is selectively disabled for the payload of RX packets belonging to different application classes (Sec.V-A). We always keep DCA enabled for the header part of the RX packets as the header of each packet is typically processed by the CPU upon reception, resulting in short use distance.

## V. IDIO Architecture

We design IDIO to address the limitations of modern DDIO technology. IDIO makes dynamic decisions about each individual packet's placement in the memory hierarchy, improving DDIO's flexibility in two dimensions. First, instead of applying a fixed policy for all incoming packets, IDIO applies differential treatment based on the per-packet quality of service targets. Second, IDIO considers three potential placements in the memory hierarchy: DRAM, LLC, and MLC. IDIO continuously monitors network RX rate and data movement in the memory hierarchy and periodically adapts its placement decisions per traffic flow accordingly.

Figure 6 shows IDIO's high-level architecture. The colored components are the ones that IDIO modifies. IDIO

introduces two new components: IDIO classifier and IDIO controller. IDIO also enhances MLC's prefetcher to support prefetches upon receiving hints from the IDIO controller and extend cache maintenance instructions to implement a multi-cacheline invalidate instruction. IDIO classifier resides in the NIC and implements a logic to identify application class, per-packet destination core, header versus payload, and the start of an RX burst. The on-chip IDIO controller collects the information embedded in each DMA transaction from IDIO classifier and monitors per-core MLC eviction statistics determine the best placement for each traffic flow. In the rest of this section, we explain IDIO's main components and how the dynamic policy governing the IDIO controller operates.

### A. IDIO Classifier

IDIO classifier resides in the NIC and implements logic to (1) identify the application class of each incoming packet, (2) identify the DMA transfer that contains the first byte of each RX packet, (3) identify the destination core for the RX packet, and (4) detect RX bursts destined for the same core. The IDIO controller uses the classification outcome mentioned above to steer RX packets in the memory hierarchy intelligently (Sec.V-B).

We assume that the sending application includes information about the application class in the header of the packets it sends. For example, for TCP/IP packets, applications can leverage the 8-bit differentiated services field (DS field) [3] in the IP header for classification purposes. The 6-bit differentiated services code point (DSCP) field can be set by the `setsockopt` function for each socket connection and updated on the fly. DSCP can be used to distinguish packets coming from different applications with different DMA buffer *use distances*. We define two application classes; class 0 are applications with short *use distance*, and class 1 are applications with long use distance or applications whose payload is rarely used/processed. For instance, a Denial of Service (DoS) detention firewall application is a class 1 application as inspection of headers is mostly sufficient for making a drop or pass decision, and further inspection into the packet payload is rarely required. Such applications can benefit from direct DRAM access for the payload to reduce LLC contention.

As the header size of packets in all the well-known network protocols is less than 64 Bytes, the DMA transaction that transfers the very first cacheline of the RX packet contains the protocol header. IDIO classifier marks the first DMA transactions carrying RX data to CPU as the cacheline that includes the header.

As IDIO supports network traffic steering to the MLCs, the destination core for each packet should be known to IDIO controller to determine which MLC to steer the packet to if MLC steering is deemed beneficial. IDIO's packet classifier builds on existing NIC support to determine each

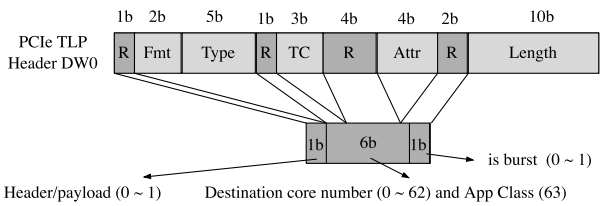


Figure 7: PCIe TLP reserved bits used by IDIO.

packet's destination core. We leverage SR-IOV and Ethernet Flow Director to create several virtual NIC ports (vPort) and pin them to network sockets created on each core using Application Device Queue (ADQ). In general, the purpose of ADQ is to map RX/TX queues directly to the application, so there is no DMA buffer or OS scheduler contention in a multi-programmed server. With ADQ, the application sets a hint (i.e., NAPI_ID) and uses this hint to map a socket to specific RX/TX queues (so those queues would go to this particular socket directly). Meanwhile, the NIC is configured with rules based on 5-tuple so that the traffic can be directed to certain RX/TX queues (using Flow Director's perfect match Filter Table, as introduced in Sec.II), which in turn match particular sockets corresponding to an application. The classifier also keeps a 32-bit *burst counter* per physical core to keep track of received bytes for each core. The *burst counters* are reset every 1$\mu$s. If the value of a counter exceeds a threshold (`rxBurstTHR`), the classifier notifies IDIO controller of a burst arrival.

To transfer the metadata extracted by the classifier on the NIC to the on-chip IDIO controller, we embed them within each DMA request by leveraging the reserved bits inside the PCIe's Transaction Layer Packet (TLP) headers ("R" bits in Fig.7). The target core number is encoded in 6 bits of the PCIe TLP header's reserved bits (bits 23, [19:16], and 11). As we will discuss in Sec.V-B, when the application class is 1, regardless of the core number, IDIO directly writes the data to DRAM. Application class 1 is identified by IDIO controller when these 6 bits are set to 1. Using this encoding, IDIO supports up to 63 cores. Header/payload and burst information are encoded into the TLP header's reserved bits at offset 31 and 10, respectively.

### B. IDIO Controller

The IDIO controller is tightly coupled with the PCIe root complex (*PCIe* in Fig.6) on the CPU chip. IDIO controller makes steering decisions based on the algorithm outlined in Alg.1, using per-packet information received from the classifier and per-core MLC writeback statistics monitored within the CPU chip. IDIO controller maintains one 32-bit counter, two 32-bit registers, and one 1-bit status register per physical core: the 32-bit `mlcWB` counter counts MLC writebacks at 1$\mu$s intervals. The 32-bit `mlcWBAcc` register accumulates 8192$\times$ consecutive samples of `mlcWB`. As shown in Alg.1 (lines 20–24), the 32-bit `mlcWBAvg` stores the average number of MLC writebacks at 1$\mu$s intervals

**Algorithm 1:** IDIO data plane and control plane

```
1  Data Plane @ IDIO controller
2  DMA [appClass, isHeader, isBurst, destCore] write
     request is received
3  fsmState[destCore]= isBurst?
     0:fsmState[destCore]
4   if isHeader then
5   |  Send prefetch-hint to destCore
6  else if appClass == 1 then
7   |  Direct DRAM write
8   else if status[destCore] == MLC then
9   |  Send prefetch-hint to destCore
10  else
11  |  Write-allocate or -update inside LLC
12
13 Control Plane @ IDIO controller
14 Every 1 μs:
15  for i in (0, number of cores) do
16  |  mlcPress= mlcWB[i]> (mlcWBAvg[i]+ mlcTHR)?
    |   high:low
17  |  update fsmState (shown in Fig. 8)
18  |  mlcWBAcc[i]+= mlcWB[i]
19 end
20 Every 8192μs:
21  for i in (0, number of cores) do
22  |  mlcWbAvg[i]= mlcWbAcc[i]/ 8192
23  |  mlcWbAcc[i]= 0
24 end
```

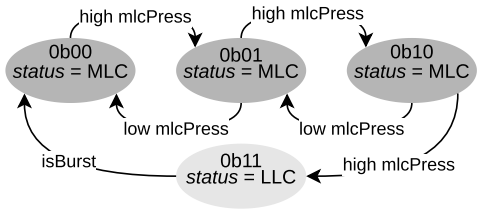



Figure 8: Per core FSM for setting *status* register in IDIO controller.

over the past 8192$\mu$s. We chose these intervals as they experimentally work well (Sec.VII); however, all the time intervals are configurable. Lastly, the one-bit `status` register indicates the destination of incoming DMA requests as follows: 0→LLC, 1→MLC.

If the DMA carries a header, regardless of its application class, it will be prefetched to MLC (Alg.1, lines 4–5). The rationale is that the header size is small, and the use distance of the header is usually short. If the application class (`appClass`) is 1, then we disable DDIO for that transaction and directly write the data into DRAM (lines 6-7). If `status` bit of the destination core is 1 (i.e., MLC), the data will be prefetched to MLC (lines 8–9). Otherwise, the DMA stays in LLC (lines 10–11).

IDIO controller updates per-core status bit using the FSM illustrated in Fig.8. The FSM implements a 2-bit saturating counter to switch the `status` bit from MLC to LLC. That is, by default, the MLC prefetching for a physical core is disabled (state `0b11`). Once a burst is identified for a physical core (Sec.V-A), the FSM transitions to state `0b00` (line 3 in Alg.1). Every 1$\mu$s, the IDIO controller measures the MLC pressure by comparing the number of MLC writebacks during the past 1$\mu$s interval (`mlcWB`) to the average writebacks over the past 8192$\mu$s (`mlcWBAvg`). A difference of `mlcWB` and `mlcWBAvg` exceeding a threshold (`mlcTHR`) indicates high MLC pressure (`mlcPress` in Alg.1 and Fig.8) and the saturating FSM counter is incremented, otherwise it is decremented (saturating at `0b00` and `0b11` as shown in Fig.8).

### C. MLC Controller

The MLC controllers implement a simple queued prefetcher logic that queues prefetch hints received from IDIO controller for specific cache blocks and send prefetch requests to the LLC accordingly. IDIO employs these prefetch hints to steer incoming network data to MLCs. The default MLC prefetcher queue size is 32 requests.

### D. Buffer Invalidation

Modern ISAs support several cache maintenance instructions for cleaning and invalidating cachelines. We extend the cache invalidate operation and introduce a new cache maintenance operation that invalidates a cacheline from private dcache and MLC, regardless of the dirty bit value. That is, the invalidation does not result in a writeback. The network application will use the instruction to explicitly invalidate the DMA buffer after it is consumed by the software stack, as shown in Fig. 6. Note that invalidate without flush instructions has already been implemented in several ISA. For example, Data Cache Invalidate by Modified Virtual Address (DCIMVAC) operation in arm_v7 ISA [8] and Data Cache Block Invalidate (DCBI) instruction in PowerPC [20] invalidate the cacheline that includes a virtual address.

Making such instructions available to a userspace application has some security implications. For example, imagine that after a process dies, the OS zeroes one of the physical pages used by that process and maps that page to a new process. If the new process uses the invalidate without flush instruction on the newly mapped page, it can observe the old content, breaking data privacy between processes. To mitigate the privacy issue, we introduce a special *Invalidatable* buffer that the kernel can allocate for a userspace application. We take a bit in the Page Table Entry (PTE) to mark a page as *Invalidatable* (currently, the Linux kernel has four reserved bits in PTE [25]). When a userspace application requests such a buffer, the kernel first flushes it to DRAM and then allocates it for the application. When an invalidation instruction is issued, the PTE bit is checked to ensure it is *Invalidatable*.

## VI. Methodology

The results are either collected from a physical or simulated two-server setup. In our physical setup, each server

Table I: Simulation configuration.

| Parameters | Values |
|---|---|
| Core ISA, freq: | aarch64, 3GHz |
| Superscalar | 3 ways |
| ROB/IQ/LQ/SQ entries | 384/128/128/128 |
| Int & FP physical registers | 128 & 192 |
| Branch predictor/BTB entries | BiMode/2048 |
| I/D/L2/L3 (per core size, assoc) | 32KB, 2/64KB, 2/1MB, 8/1.5MB, 12 |
| I/D/L2/L3 (latency, MSHRs) | 1CC, 2/2CC, 6/12CC, 16/24CC, 32 |
| DRAM/mem size | DDR4-3200/4GB |
| Operating system | Linux Linaro (kernel 5.4.0) |
| Network HW | 2x100Gbps Ethernet NICs |
| Network SW | DPDK 21.11.0, 1514-byte packets |

Table II: Functions used for evaluation. `TouchDrop` and `L2Fwd` are DPDK-based network functions.

| Function | Description |
|---|---|
| `TouchDrop` | receive packets, touch data, drop packets. |
| `L2Fwd` | receive packets, forward packets based on Ethernet header. |
| `LLCAntagonist` | allocate a variable size buffer and randomly access elements |

is equipped with an Intel Xeon Gold 6242 CPU, 96GiB DDR4-3200 DRAM, 3 memory channels, and one Mellanox ConnectX-5 Dual 100Gb Ethernet card. We use the Linux Perf tool [12] to collect performance counter values. Table I shows our gem5 configuration. We enabled gem5 in full-system mode to run userspace networking functions developed on the DPDK framework.

Since this work focuses on efficient inbound network data steering, we use three RX-intensive DPDK network functions to evaluate IDIO. Table II shows the description of the network functions as well as `LLCAntagonist` that is used to create LLC interference at various degrees. Before we collect stats with `LLCAntagonist`, we warm up caches by initializing the allocated buffer. To ensure that `LLCAntagonist` generates enough LLC pressure and is sensitive to LLC contention, we set the MLC size of the core running `LLCAntagonist` to 256KB.

We generate network traffic using DPDK pktgen [13] and a hardware load generator model for the physical system and gem5 experiments, respectively. The load generator model enables simple network benchmarking in gem5 without simulating multiple system nodes. Using hardware load generators is ubiquitous in the industry for evaluating the performance of the network [31]. To measure network latency, we annotate the DPDK applications in Table I with gem5 pseudo instructions to get the timestamps when a packet is completely processed. For example, in `TouchDrop`, a pseudo instruction is executed once a full packet is touched.

We generate steady and bursty network traffic with different packet and DPDK ring buffer sizes. We define a packet burst with three parameters: burst period, burst length, and burst rate. The *burst period* is the time between the start of two consecutive bursts, the *burst length* is the time between the first and last packet generated within one burst, and the *burst rate* is the packet rate (bits per second) during the bursts. We fix the burst period to 10ms and set the burst rate to either 10Gbps, 25Gbps, or 100Gbps. Based on the burst rate and packet size, we then set the burst length to receive exactly ring-buffer-size number of packets in each burst. Unless stated otherwise, we set the packet size to 1514 bytes, which is equal to Ethernet maximum transmission unit size. For instance, if the ring buffer size is 1024, then the burst length for 10Gbps, 25Gbps, and 100Gbps burst rates are 1.155, 0.231, and 0.115 ms, respectively. Therefore, the burst length depends on the value of the burst rate, packet size, and ring buffer size. The rationale for setting the burst length proportional to ring buffer size is to prevent packet drops within a single burst, as we experience packet drops as soon as the ring buffer becomes full.

We experimentally set the `rxBurstTHR` and `mlcTHR` threshold values to 10Gbps and 50 million writeback transactions per second (MTPS), respectively. We perform a sensitivity analysis to threshold values in Sec.VII.

## VII. Evaluation

In this section, we evaluate the effectiveness of IDIO in reducing MLC/LLC writebacks, contention in LLC, and tail-latency while processing NFs at various configurations.

**DMA leak and unnecessary writebacks mitigation:** Figure 9 compares MLC writeback and LLC writeback rates while processing one burst in `TouchDrop` for DDIO and IDIO at 100Gbps and 25Gbps burst rates. To show the synergy between techniques, we include Invalidate (Fig.9c and 9d) and Prefetch (Fig.9e and 9f) configurations that only enable self-invalidating I/O buffers (Sec.IV-A) and network-driven MLC prefetching (Sec.IV-B) techniques. The Static (Fig.9g and Fig.9h) and IDIO (Fig.9i and Fig.9j) configurations enable both techniques. However, the Static configuration always enables MLC prefetching for *appClass* 0 (by hardcoding *status* register in Alg.1 to MLC), but IDIO dynamically enables and disables MLC prefetching based on the FSM explained in Fig.8. We also plot DMA request rate of `TouchDrop` application to show different phases of the burst processing. Note that since `TouchDrop` only receives packets, all the DMA requests are write. The execution phase starts $\sim$1.9µs after the first DMA transaction. This delay is the time it takes for NIC to writeback the used descriptors to the CPU after the DMA-transfer of the RX data to the CPU is completed. Only after the descriptors are updated, the DPDK polling mode driver can detect packet arrival and start the execution phase (cf. Fig.9). Our sampling interval for calculating the rates in Fig.9, Fig.11, and Fig.13 is 10$\mu$s.

At first glance, two things stand out in Fig.9: 1) IDIO significantly reduces the LLC writebacks at all load levels, and 2) IDIO reduces the processing time of a burst. Moreover, Figures 9c$\sim$9f clearly show the synergy between self-invalidating and MLC prefetching techniques. As evident in the figures, self-invalidations significantly reduce both MLC and LLC writebacks, while MLC prefetching reduces the burst execution time by increasing the aggregate residency of RX network data in the cache hierarchy. Fig.10 compares

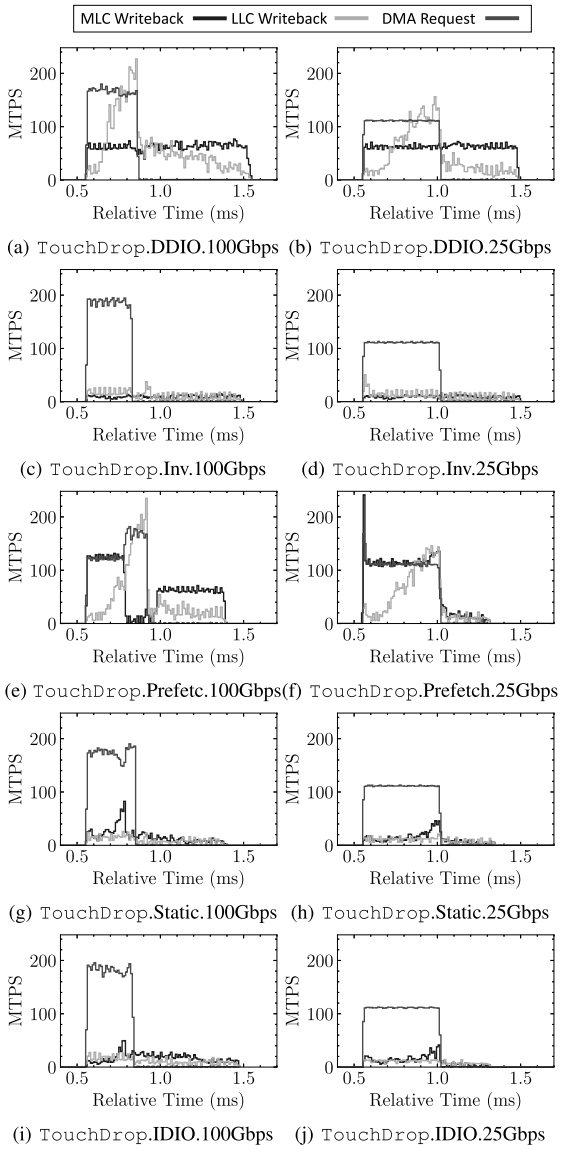



Figure 9: Two TouchDrop processes running with 1024 ring buffer size and 1514 bytes packets. MTPS is Million Transactions Per Second.

the number of MLC writebacks, LLC writebacks, DRAM read, and DRAM write transactions during the burst shown in Fig.9, normalized to that of DDIO. Exe Time in the figure is the burst processing time (i.e., start of DMA phase till the end of the execution phase) of IDIO normalized to the burst processing time of DDIO. The MLC writebacks at 100Gbps, 25Gbps, and 10Gbps are reduced by 73.9%, 83.7%, and 63.8% compared to DDIO, respectively. Likewise, IDIO significantly reduces LLC writebacks and DRAM bandwidth utilization. In fact, IDIO almost eliminates DRAM write bandwidth. Such data movement reductions in the memory subsystem result in 18.5% and 22.0% improvement in burst

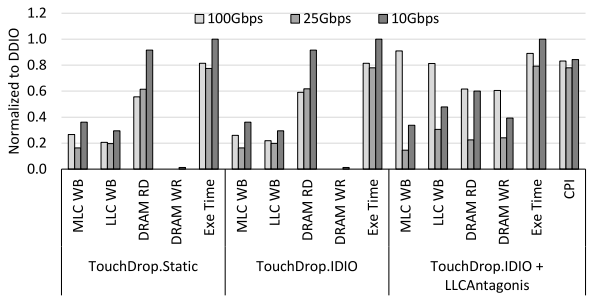



Figure 10: Normalized number of MLC writeback, LLC writeback, DRAM read, and DRAM write transactions as well as burst processing time (Exe Time) of Static and dynamic IDIO for the configurations illustrated in Fig. 9 and a co-running scenario. Lower values are better.

processing time at 100Gbps and 25Gbps, respectively.

Although IDIO significantly reduces the number of MLC and LLC writeback transactions at all burst rates, Fig.10 suggests that IDIO proves the most useful at 25Gbps compared with 100Gbps or 10Gbps burst rates. The reason is that at high burst rates, the MLC-prefetching mechanism quickly fills up MLC and starts experiencing high MLC writebacks, and gets disabled early on. However, at medium burst rates, while IDIO prefetches RX data to MLC, the core consumes data at a comparable rate, and thus the self-invalidating mechanism in IDIO frees up MLC space for new prefetches. Such timely prefetch-invalidate is realized when IDIO prefetches at the same rate as the CPU consumes data. Although our simple queued prefetcher performs adequately well at all burst rates, a more sophisticated prefetcher that follows the CPU pointer in the ring buffer to regulate the MLC prefetching rate will likely provide more benefit. Because at lower burst rates, the CPU processes data as soon as the packets arrive at NIC, there is no room for IDIO to prefetch RX data into MLC. However, the self-invalidating mechanism is beneficial at any burst rate. Note that burst processing time is not improved at the 10Gbps rate because packets are not queued up in the ring buffer, and therefore improvement in per-packet processing time does not improve the burst processing time. However, we still see tail latency reduction even at 10Gbps (as we will discuss later in Fig.12)

Interestingly, even the Static IDIO policy for MLC prefetching provides most of the benefits of the dynamic IDIO policy. The difference between Static and dynamic IDIO configurations in Fig.9g and Fig.9i is where Static configuration lets MLC writeback rate exceed 50 MTPS, but IDIO regulates MLC writeback rate by disabling MLC prefetching when MLC writeback rate exceeds mlcTHR (i.e., 50 MTPS). For lower burst rates like 25Gbps, there is no difference between Static and IDIO since the CPU processing rate of DMA buffers is comparable to the DMA write rate, and thus the self-invalidating mechanism frees up space in the MLC for new MLC prefetches without

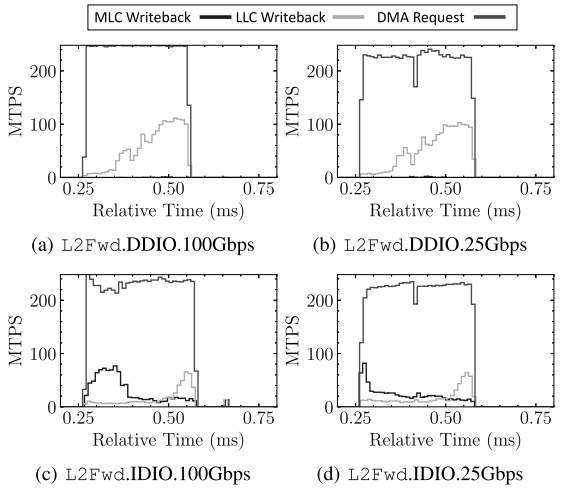



Figure 11: Two L2Fwd processes running with 1024 ring buffer size and 1024 bytes packets. MTPS is Million Transactions Per Second.

introducing MLC pressure.

To summarize, here are the main takeaways from Fig.9: (1) IDIO significantly reduces MLC, and LLC writebacks, (2) IDIO improves packet processing rate, (3) IDIO's efficiency is not sensitive to the threshold values due to the seamless synergy between MLC prefetching and self-invalidating buffer at various burst rates.

**Experimenting with shallow NFs:** Figure 11 shows the MLC and LLC writeback rate timeline for L2Fwd with 1024 bytes packets with DDIO and IDIO configurations. L2Fwd implements a zero-copy run-to-completion buffer recycling model and uses the RX DMA buffer for forwarding the packet back to the network. Therefore, a DMA buffer is consumed only after the forwarding is completed. In the baseline DDIO, the payload remains in the LLC or leaks to DRAM, and only the header is used in L2Fwd for processing. Since the header size is small (even a full 1024 size ring buffer only takes 64KB), as shown in Fig.11a and Fig.11b, there is almost no MLC activity in the DDIO configuration. However, the LLC writeback rate gradually increases as more data is received from the network. These writebacks can be DMA leaks (not consumed DMA buffers) or unnecessary writebacks of consumed DMA buffers. In contrast, IDIO significantly reduces the LLC writebacks by (1) effectively utilizing the unused MLC space to *admit data* to the non-inclusive MLC and reduce the LLC contention, and (2) invalidating consumed LLC-resident buffers after the forwarding is completed. IDIO explores an interesting data steering option, and that is *data admission* to higher level memory versus *data eviction* to lower level memory. Such data steering has not been an option in inclusive cache hierarchies and needs to be further explored in non-inclusive cache hierarchies.

IDIO also supports direct DRAM access for application

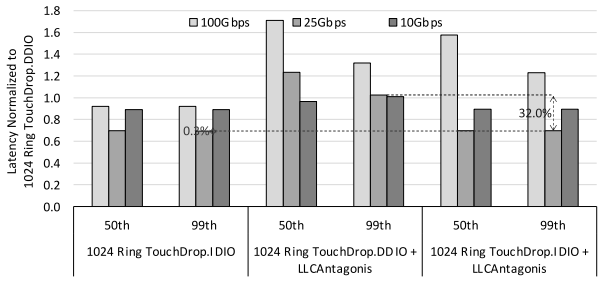


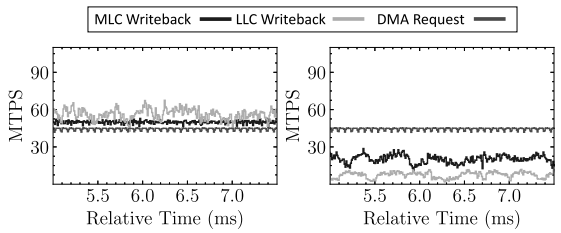

Figure 12: 50th and 99th percentile latency for TouchDrop with 1514 byte packet size.

classes with high use distance of the RX payloads. L2Fwd does not fit into this class as the payload is quickly used for transmission. We evaluate the direct DRAM access feature of IDIO by running a variant of L2Fwd where the application drops the payload after processing the header. As explained in Sec.V-A, each packet carries the class information of the sending application and, in the RX server, IDIO directly transfers the payload to DRAM. In this scenario, the LLC writeback rate and DRAM write bandwidth are the same as network RX bandwidth.

**LLC contention mitigation:** To quantify the benefit of less LLC interference, we co-run LLCAntagonist and TouchDrop with 1024 ring buffer size and 1514 byte packets at various burst rates. As illustrated in Fig.10 (TouchDrop.IDIO + LLCAntagonist configuration), IDIO is effective in reducing MLC and LLC writebacks and DRAM bandwidth utilization even when co-running an NF with an LLC-intensive application. More importantly, co-running with IDIO improves burst processing time by 10.9% and 20.8% for 100Gbps and 25Gbps compared with baseline DDIO, respectively. The CPI of the LLCAntagonist is also improved by 16.8%, 22.1%, and 15.7%, respectively.

**Tail-latency mitigation and performance isolation:** Figure 12 compares the 50th and 99th percentile latency of packets processed in TouchDrop using 1024 ring buffer sizes when running solo and co-run with LLCAntagonist. We normalized all the data points to DDIO's solo run. IDIO reduces TouchDrop's 99th latency by 7.9%, 30.5%, and 10.9% when running solo, and 6.1%, 32.0%, and 8.2% when co-running at 100Gbp, 25Gbps, and 10Gbps, respectively. As shown, IDIO also provides isolation between the network function and LLCAntagonist at 25 and 10Gbps rates. At higher network rates, the network function becomes too sensitive to LLC interference, and more sophisticated mechanisms are required to provide performance isolation [26].

**Experimenting with steady network traffic:** So far, all the results assumed bursty network traffic. Fig.13 illustrates the effectiveness of IDIO in reducing MLC and LLC writebacks where each TouchDrop receives steady network traffic at 10Gbps rate (total 20Gbps). Note that we experience packet drops at network rates higher than 12Gbps for each




(a) `TouchDrop`.DDIO.10Gbps (b) `TouchDrop`.IDIO.10Gbps

Figure 13: Two `TouchDrop` processes running with 1024 ring buffer size and 1514 bytes packets at steady load. MTPS is Million Transactions Per Second.

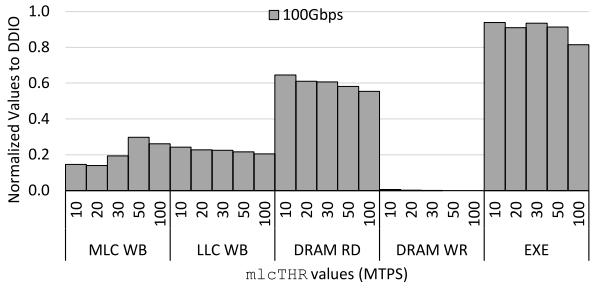



Figure 14: Sensitivity of IDIO to `mlcTHR` threshold value.

core. Although the LLC writeback rate is not as significant as when a burst is received, Fig.13a shows that DDIO experiences consistent MLC and LLC writebacks at a steady RX rate. In fact, the MLC writeback rate is the same as the bursty traffic. The reason is that most of the MLC writebacks belong to the consumed DMA buffers, and since the packet processing rate on the CPU is the same as when a burst of packets is received, DDIO experiences the same MLC writeback rate in both steady and bursty traffic. Self-invalidating DMA buffer mechanism in IDIO removes most of the MLC writebacks and significantly reduces LLC writebacks.

**Sensitivity to threshold values:** Lastly, we show that IDIO is not overly sensitive to the value of `mlcTHR` threshold. Fig.14 compares the statistics reported in Fig.10 when sweeping `mlcTHR` value from 10 MTPS to 100 MTPS. Note that we set `mlcTHR` to 50 MTPS for all the previously reported results. As illustrated in the figure, IDIO consistently improves the reported statistics regardless of the threshold value. We only show the sensitivity analysis for 100Gbps burst rate because as the burst rate decreases, the sensitivity to the `mlcTHR` also decreases.

## VIII. Related Work

**On-chip communication acceleration:** The trend of increasing numbers of cores on the same chip and higher I/O device bandwidth demands fast and efficient on-chip communication. CAF [38] proposed a hardware-assisted core-to-core queuing mechanism to reduce the coherence traffic and also enable fine-grained core-to-core communication. HyperPlane [29] designed a hardware coherence-assisted notification mechanism for the multi-core software data plane. MOPED [18] proposed extensions to the directory-based coherence protocols to offload the message synchronization and data copying to the hardware for accelerating MPI messages on a CMP. IDIO is orthogonal and compatible with them.

**Network data placement:** Data Direct I/O (DDIO) technology [1] injects I/O data directly to CPU's LLC instead of detouring to DRAM, and several enhancements have been proposed for the default static DDIO. For example, IAT [41] implements a dynamic DDIO policy by re-configuring DDIO LLC ways based on runtime monitoring of system stats to mitigate LLC writebacks. CacheDirector [14] improves default DDIO to steer the header of each network packet into the LLC tile closest to the core that will process the packet, with the goal of reducing the processing latency for fine-grained network functions. However, due to the limited flexibility of the current commercial hardware, they are not able to fine-tune the destination of the inbound data and still suffer from the penalty of a high MLC writeback rate. IDIO proposes a more comprehensive and fine-grained (both spatially and temporally) control mechanism for the inbound I/O traffic, which is especially important for the tail latency performance of the latency-critical NFs. DMA Cache [34] identifies the different characteristics of DMA versus CPU data and introduces a cache structure specifically used for DMA data. One of IDIO's side benefits is to enforce isolation between DMA and CPU data. NEBULA [33] proposed selective network data steering to private L1 data caches. IDIO steers data to large MLCs and mitigate cache trashing concerns while capturing similar latency and traffic isolation benefits. NetDIMM [4] enables memory modules to directly access network. IDIO can leverage NetDIMM to directly move data to DRAM when selective direct DRAM access is deemed beneficial.

**Dynamic self invalidation:** Dynamic Self Invalidation (DSI) [23] opportunistically invalidates cachelines in a shared memory multi-processor to reduce the overhead of cache coherency. However, DSI does not drop the cacheline like IDIO as the cacheline data is still alive and used by other processors. Wang et al. [37] proposed a NIC-triggered eviction policy for used network buffers. In contrast, the network buffer invalidation in IDIO is triggered by the software stack that has full knowledge about the liveliness of the network buffers.

## IX. Conclusion

In this paper, we started with a detailed explanation of data movement in a non-inclusive cache hierarchy in the context of network applications. Then we made three key observations about the data movement in a non-inclusive hierarchy by running carefully crafted network experiments

on real hardware and a full-system simulator: (1) MLC space is not efficiently utilized by DMA buffers, (2) packet processing software stacks suffer from high rates of writebacks from MLC to LLC that results in LLC contention, and (3) the LLC share of DMA buffers is often bloated, resulting in breaking the isolation between I/O and other applications. We introduced IDIO that implements three synergistic ideas for resolving the observed issues: (1) self-invalidating I/O buffers, (2) network-driven MLC prefetching, (3) and selective direct DRAM access. Our detailed experiments using a full-system simulator — capable of running modern DPDK userspace NFs and sustaining over 100Gbps network bandwidth — show that IDIO is effective in reducing on-chip data movement and providing isolation for shared LLC when running various NFs.

## ACKNOWLEDGEMENT

This work was supported in part by grants from National Science Foundation (CNS-1705047), National Research Foundation of Korea grant funded by the Korean Government (NRF-2020R1C1C1013315), the Institute of Information & communications Technology Planning & Evaluation grant funded by the Korean government (No.2018-0-00503), Samsung Electronics, and Intel Corporation. Nam Sung Kim has a financial interest in Samsung Electronics and NeuroRealityVision Corporation.

## REFERENCES

[1] "Intel Data Direct I/O Technology (Intel DDIO): A Primer." [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html

[2] "Intel® Ethernet Flow Director and Memcached Performance," accessed: 03/14/2020. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf

[3] "Type of service," https://en.wikipedia.org/wiki/Type-of-service.

[4] M. Alian and N. S. Kim, "NetDIMM: Low-latency near-memory network interface architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 699–711.

[5] M. Alian, J. Shin, K.-D. Kang, R. Wang, A. Daglis, D. Kim, and N. S. Kim, "IDIO: Orchestrating inbound network data on server processors," *IEEE Comput. Archit. Lett.*, vol. 20, no. 1, p. 30–33, jan 2021. [Online]. Available: https://doi.org/10.1109/LCA.2020.3044923

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

[7] ARM, "Arm DynamIQ Shared Unit Technical Reference Manual r3p0," https://developer.arm.com/documentation/100453/0300/.

[8] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, "Cache maintenance operations AR_v7," https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-/Functional-grouping-of-VMSAv7-system-control-registers/Cache-maintenance-operations--functional-group--VMSA?lang=en#BEIEFEFI.

[9] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proc. VLDB Endow.*, vol. 9, no. 7, p. 528–539, mar 2016. [Online]. Available: https://doi.org/10.14778/2904483.2904485

[10] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM conference (SIGCOMM'21)*, Virtual Event, Aug. 2021.

[11] I. Cutress, "I Keep My Cache Private," https://www.anandtech.com/show/11550/the-intel-skylakex-review-core-i9-7900x-i7-7820x-and-i7-7800x-tested/4.

[12] A. C. De Melo, "The new linux perf tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[13] DPDK, "Pktgen - Traffic Generator powered by DPDK," https://github.com/pktgen/Pktgen-DPDK.

[14] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostic, "Make the most out of last level cache in intel processors," in *Proceedings of the Fourteenth EuroSys Conference*, 2019.

[15] A. Farshin, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, "Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks," in *Proceedings of 2020 USENIX Annual Technical Conference (ATC'20)*, Virtual Event, Jul. 2020.

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

[17] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. I. Seltzer, "Parking packet payload with P4," in *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020*, D. Han and A. Feldmann, Eds. ACM, 2020, pp. 274–281. [Online]. Available: https://doi.org/10.1145/3386367.3431295

[18] J. Gu, Y. Sun, S. S. Lumetta, and R. Kumar, "MOPED: Accelerating data communication on future cmps," *IEEE Micro*, vol. 31, no. 4, pp. 42–50, 2011.

[19] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network I/O," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE, 2005, pp. 50–59.

[20] IBM, "dcbi (Data Cache Block Invalidate) instruction," https://www.ibm.com/docs/en/aix/7.2?topic=set-dcbi-data-cache-block-invalidate-instruction.

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

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

[23] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 23, no. 2, p. 48–59, may 1995. [Online]. Available: https://doi.org/10.1145/225830.223995

[24] E. A. Leon, K. B. Ferreira, and A. B. Maccabe, "Reducing the impact of the memory wall for I/O using cache injection," in *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*, 2007, pp. 143–150.

[25] Linux, "Page Table Types," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/include/asm/pgtable_types.h.

[26] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462.

[27] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

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

[29] A. Mirhosseini, H. Golestani, and T. F. Wenisch, "HyperPlane: A scalable low-latency notification accelerator for software data planes," in *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*, Virtual Event, Oct. 2020.

[30] L. NEIO Systems, "DDIO — oh oh," https://latency-matters.medium.com/ddio-oh-oh-e0099754b7d9.

[31] K. Pandit, B. Bian, V. M. Prasad, A. Kwatra, P. Lu, M. Riess, W. Willey, H. Xie, and G. Xu, "Modeling the impact of cpu properties to optimize and predict packet-processing performance."

[32] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir, "The benefits of general-purpose on-NIC memory," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, Lausanne, Switzerland, Feb. 2022.

[33] M. Sutherland, S. Gupta, B. Falsafi, V. J. Marathe, D. N. Pnevmatikatos, and A. Daglis, "The NeBuLa rpc-optimized architecture," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 199–212. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00027

[34] 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*, 2010, pp. 1–12.

[35] 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)*. USENIX Association, 2018.

[36] M. Wang, M. Xu, and J. Wu, "Understanding I/O direct cache access performance for end host networking," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, feb 2022. [Online]. Available: https://doi.org/10.1145/3508042

[37] R. Wang, S. Gobriel, C. Maciocco, T.-Y. C. Tai, B.-Z. Friedman, H. T. Nguyen, N. N. Venkatesan, M. A. O'hanlon, S. M. Shah, S. Jain *et al.*, "Technologies for network packet cache management," Jun. 5 2018, uS Patent 9,992,299.

[38] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, "CAF: Core to core communication acceleration framework," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2016, pp. 351–362.

[39] X. Wei, X. Xie, R. Chen, H. Chen, and B. Zang, "Characterizing and optimizing remote persistent memory with RDMA and NVM," in *Proceedings of the 2021 USENIX Annual Technical Conference (ATC'21)*, Virtual Event, Jul. 2021.

[40] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 888–904.

[41] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, "Don't forget the I/O when allocating your LLC," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 112–125.