Patching up Network Data Leaks with Sweeper

Marina Vemmou
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
mvemmou@gatech.edu

Albert Cho
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
acho44@gatech.edu

Alexandros Daglis
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
alexandros.daglis@cc.gatech.edu

***Abstract*—Datacenters have witnessed a staggering evolution in networking technologies, driven by insatiable application demands for larger datasets and inter-server data transfers. Modern NICs can already handle 100s of Gbps of traffic, a bandwidth capability equivalent to several memory channels. Direct Cache Access mechanisms like DDIO that contain network traffic inside the CPU's caches are therefore essential to effectively handle growing network traffic rates. However, a growing body of work reveals instances of a critical DDIO weakness known as "leaky DMA", occurring when a significant fraction of network traffic leaks from the CPU's caches to memory. We find that such network data leaks cap the network bandwidth a server can effectively utilize.**

We identify that a major culprit for such network data leaks are evictions of already consumed dirty network buffers. Our key insight is that buffers already consumed by the application typically need not be written back to memory, as their next reuse will be a full overwrite with new network data by the NIC. We introduce Sweeper, a hardware extension and API that allows applications to mark such consumed network buffers. Hardware then skips writing marked buffers back to memory, drastically reducing memory bandwidth consumption and mitigating the performance penalty of network data leaks. Sweeper boosts a 24-core server's peak sustainable network bandwidth by up to 2.6× as compared to DDIO-based configurations.

I. Introduction

The rapid growth of data and application demands is driving rapidly increasing networking demands in modern datacenters. To illustrate, Google's internal datacenter network traffic reportedly doubled every 15 months over six years [50]. Networking latency is also a critical figure of merit, as the trend for fine-grained software decomposition places inter-server communication latency on the critical path [14]. The net result is that datacenter operators have been rapidly evolving the entire networking stack, ranging from bespoke fabrics and protocol stacks [15], [19], [20], [39], [50] to specialized hardware endpoints for accelerated networking functionality [7], [13], [55]. Unsurprisingly, NIC vendors are quickly ramping up their bandwidth offerings, with 400Gbps devices already commercially available [43]. From a server architecture design perspective, such data rates represent unprecedented network-associated data movement, raising implications on the memory system design. To put numbers into perspective, a 400Gbps endpoint corresponds to the bandwidth capacity of several DDR4 memory channels.

The implications of growing network bandwidth on a server's memory hierarchy performance necessitates revisiting the conventional DMA-based approach that moves data between the NIC and CPU through DRAM. Therefore, technologies like Intel's DDIO [26], which allow the NIC to directly move data through the CPU's last-level cache (LLC), are already becoming standard. Although a step in the right direction, recent studies have identified deficiencies in DDIO's data movement policies when heavily loaded networked systems start leaking network data from the LLC to memory at considerable rates [12], [57], [58]. We study this problem and find that such leaks can significantly hamper the maximum network rate a server can effectively sustain, degrading peak performance by up to 2.65×.

Prior work has proposed a range of approaches to address DDIO limitations. ResQ [57] and NeBuLa [53] propose limiting the aggregate capacity of network buffers to improve LLC residency. However, shallow versus deep buffer provisioning presents a tradeoff. While effective in certain scenarios, provisioning shallow buffers is not a panacea, as it reduces resiliency to packet arrival bursts that can result in undesirable packet drops, which could be avoided with deeper buffers. In addition, minimum buffer depth provisioning is driven by rapidly increasing NIC line rates [12]. Other approaches propose dynamic resizing of the LLC capacity allocated to DDIO [58], placing network data in higher-level caches in addition to the LLC [1], [53], or selective cache bypassing [12]. While many of these mechanisms mitigate performance problems by delaying the onset of network data leaks, they do not fundamentally tackle the leaks' root cause.

In this work, we study the origin and classify network data leaks in systems with highly provisioned network capabilities. We corroborate prior findings that memory bandwidth interference is the main culprit of performance degradation [53] and identify that dirty but *already consumed* network buffers that get eventually evicted from the LLC are the main contributor of such interference. This critical observation suggests that the crux of the problem is writebacks of useless data that can be omitted without violating correctness, because the subsequent reuse of a consumed network buffer is a complete overwrite with new network data by the NIC.

We propose Sweeper, a hardware extension in the cache hierarchy that mitigates the performance implications of network data leaks, by drastically mitigating memory bandwidth interference caused by network data churn between the LLC and memory. Sweeper skips writing consumed network buffers

back to memory when it is safe to do so. A new instruction and API allows software to indicate when the contents of a network buffer instance have been conclusively used—a concept very similar to the well-known `free()` operation used in high-level languages like C, that instructs reclamation of allocated memory. The cache hierarchy then ensures that cache blocks corresponding to such marked buffers are not needlessly written back to memory, thus conserving precious memory bandwidth resources. Our evaluation on a set of scenarios with heavy networking activity demonstrate that Sweeper conserves up to 1.3× of memory bandwidth, allowing a server to utilize a 2.6× higher peak network throughput compared to using DDIO alone.

We make the following contributions:

- We study the performance implications of network data leaks in server architectures with highly provisioned network bandwidth and find that dirty evictions of already consumed network buffers significantly contribute to increased memory bandwidth utilization and overall system performance degradation.
- We identify that writebacks of such consumed network buffers are wasteful and can typically be avoided for memory bandwidth savings. Based on that insight, we propose Sweeper, a hardware extension and API complementary to DDIO, that allows the cache hierarchy to avoid wasteful writebacks, after the application explicitly marks consumed network buffers.
- Our evaluation of scenarios with heavy networking activity shows that Sweeper improves memory bandwidth utilization efficiency by up to 1.3×, yielding throughput improvements of up to 2.6× over plain DDIO configurations.
- Sweeper breaks the current tradeoff between shallow buffer provisioning to reduce performance implications due to their large footprint and deep buffer provisioning for resilience to packet drops, by removing the performance bottlenecks associated with the latter.

Paper outline: §II provides a brief background on DDIO and the problem of network data leaks from the LLC to memory. §III briefly introduces the methodology we use throughout the paper, first to trace the problem of network data leaks to its sources in §IV's study and later to comprehensively evaluate Sweeper. §V introduces Sweeper's design. We evaluate key parameters affecting the impact of network data leaks and Sweeper's efficacy in eliminating such leaks in §VI. Finally, we discuss related work in §VII and conclude in §VIII.

II. Background

A. Data Direct I/O

Data movement between host memory and I/O devices, including Network Interface Controllers (NICs), has historically been handled by Direct Memory Access (DMA). DMA directly transfers data between the device and the host's main memory, bypassing the cache hierarchy and invalidating any cached blocks that fall into the range of the accessed address range. Although traditional DMA functionality has been sufficient for all forms of I/O for decades, it is becoming a bottleneck for modern high-bandwidth I/O devices, such as NVMe storage and NICs handling 100Gbps+ data rates. Network capabilities in particular have been ramping up rapidly, fueled by the growing needs of modern applications handling massive datasets, distributed across the memory of thousands of servers. Commercially available NICs already offer up to 400Gbps full-duplex data movement [43], equivalent to the capability of four to eight (depending on their frequency) DDR4 DRAM channels. Evidently, using traditional DMA to pump such volumes of I/O traffic through memory can quickly become a severe performance bottleneck.

To address the challenges stemming from the shrinking gap between I/O and memory bandwidth offerings, Data Direct I/O (DDIO) technology allows I/O devices to avoid involving main memory in their data transfers by using the multi-MB Last-Level Cache (LLC) of modern CPUs for network data placement. In this work, we specifically focus on the implications of DDIO on systems with high-bandwidth NICs and the data movement behavior on the ingress path (i.e., arriving packets from the network to the CPU host). The key difference between the ingress paths of traditional DMA and DDIO is that the latter directly writes incoming network traffic in the LLC. If the target address range's corresponding cache blocks are not already present in the LLC, they are directly write-allocated, without triggering a memory access. To prevent write-allocated incoming network data from thrashing the entire LLC, DDIO by default uses two LLC ways for NICs, with model-specific registers allowing the number to be configured [2]. DDIO technology thus achieves two benefits. First, it reduces access latency to network data for both the CPU and NIC to that of an LLC access instead of a memory access. Second, it reduces memory bandwidth pressure, as much (or all) of the network data movement is absorbed by the SRAM-based LLC's ample bandwidth, instead of consuming precious DRAM bandwidth resources. For further details on DDIO's operation, we refer the interested reader to prior work that extensively analyzes DDIO technology [2], [26].

Note that Intel's DDIO technology is a specific instance of the general approach of Direct Cache Access (DCA) [23] for network data, and other CPU vendors implement their own version of DCA (e.g., ARM's Cache Stashing [4]). All such DCA mechanisms are fundamentally similar. Although we use the term "DDIO" throughout this paper, our study is not tied to any artifacts of the specific implementation and our observations and proposed solution apply to all alternative mechanisms in the DCA family.

B. Network Data Movement Overheads

DDIO has proven to be a powerful tool in the efforts to boost the performance of network-heavy applications, but is not without caveats. Effective use of DDIO relies on successfully containing the majority of network traffic in a subset of the LLC, which is a shared resource of finite capacity. A plethora of prior work has identified scenarios that have been problematic for DDIO, resulting in severely underperforming

systems [12], [16], [40], [44], [53], [57], [58]. Performance problems have been broadly associated with failure to contain network data within the LLC, resulting in significant rates of data spilling to memory. The umbrella term "leaky DMA", coined by Tootoonchian et al. [57], has been broadly used to describe such problematic scenarios. Some prior work uses the term "leaky DMA" to specifically refer to network buffers that have been write-allocated in the LLC by the NIC, but were evicted to memory due to cache pressure before the CPU picked the network data up. In this paper, we use the term "**network data leaks**" to refer to all instances of network data movement from the LLC to memory. There are two semantically different classes of network data leaks, which we analyze in this paper. Understanding the distinction between these two classes is important, as they have different performance implications and require different solutions.

The first class of data leaks is what is usually referred to as "leaky DMA": incoming network data that is written to the LLC by the NIC, but is evicted to memory before the receiving application is done processing it. We call data leaks of this first class "**premature buffer evictions**". The implications of premature buffer evictions are two-fold. First, the CPU's latency to access the network data increases by a latency equal to a memory access. That added latency cost alone is only noticeable for the most latency-sensitive applications deployed on the most latency-optimized architectures [41], [53]. Second, such data leaks consume memory bandwidth, potentially even more than a traditional DMA, as an incoming network data block may evict another cache block to memory, get evicted due to capacity pressure, and then be brought back again upon a demand access from the CPU. At modern network line rates, a considerable fraction of network traffic leaking in that manner can represent a large fraction of a server CPU's total available memory bandwidth (c.f. §II-A). In turn, memory bandwidth oversubscription can introduce significant queuing delays, degrading overall system performance.

The second class of data leaks consists of network buffers containing data that have been already consumed by the application. We call data leaks of this second class "**consumed buffer evictions**". Unlike premature buffer evictions, consumed buffer evictions do not introduce latency concerns. The next time the same buffer is reused by the NIC, the newly arrived network data will be directly write-allocated in the LLC, without memory involvement. For the same reason, every cache block holding a buffer written by the NIC is dirty. Therefore, every consumed buffer eviction triggers a writeback to memory, consuming precious memory bandwidth. A key difference between writebacks for consumed and premature buffer evictions is that the former correspond to data that will never be *read* again. This distinction is critical, as it can lead to solutions that are uniquely applicable to performance problems caused by premature buffer evictions, as is the case for Sweeper.

In §IV, we study problematic scenarios with heavy network data leaks and distinguish between premature and consumed buffer evictions. Although we find that premature buffer evictions are present in extreme scenarios, the first bottleneck encountered is consumed buffer evictions.

C. Network Data Leaks in Userspace Network Stacks

A key parameter affecting the advent and severity of network data leaks is the aggregate number of receive (RX) buffers used by the network stack: the more the used buffers, the larger their memory footprint, and the more challenging it is to keep them LLC-resident. Based on this observation, ResQ proposes limiting the number of provisioned buffers, so that they can comfortably reside in the LLC DDIO ways [57]. NeBuLa proposes a specialized architecture for services with very stringent response latency SLOs, featuring a hardware mechanism that dynamically monitors the queue depth of pending incoming requests. When queuing conditions are heavy enough to result in application SLO violations, NeBuLa explicitly drops new incoming packets, thus implicitly tackling buffer bloat performance concerns [53]. While limiting the number of receive buffers works well in several cases, it is not a panacea, as it incurs a toll: shallow buffering is less resilient to packet arrival bursts and can lead to undesirable packet drops [12].

Userspace network stacks like a DPDK dataplane allocate a receive ring buffer per core, resulting in buffer sizes comparable to modern LLC capacities. To illustrate, with a default ring buffer size of 1024 entries for typical MTU-sized (1.5KB) packets and 20 cores, the total size of network buffers is ∼30MB. For networked systems implementing a Virtual Interface Architecture (VIA) [10], like RDMA, the buffer bloat can be even more pronounced, as high-performance, synchronization-free reliable communication requires allocating dedicated receive buffers not only per core, but also per communicating endpoint [9], [31], [42], [51]. Thus, it is possible for the aggregate size of allocated receive buffers in such cases to be in the range of 100MB, exceeding the entire LLC capacity of even high-end servers.

III. Methodology

We briefly introduce the most critical methodological aspects of the system used throughout this paper's studies and defer details to Appendix A. We employ microarchitectural simulation to model a server CPU with 24 cores, a shared 36MB 12-way LLC, and four memory channels. As we will demonstrate, memory bandwidth availability significantly affects performance degradation due to network data leaks. Our memory setup corresponds to a medium-range memory-bandwidth-per-core provisioning encountered in modern server CPUs. Current commercial offerings range from one DDR4 channel per eight cores (e.g., AMD EPYC [22]) to one per four cores (e.g., Xeon Gold 6342 [29]). We base our initial study in §IV on such medium provisioning and demonstrate the generality of our findings as a function of memory bandwidth availability in §VI-D.

Our evaluation does not cap the available network bandwidth. Instead, we investigate the peak network bandwidth the CPU can effectively handle in each system configuration,

and demonstrate that network data leaks directly affect this attainable peak. Without loss of generality in our study's observations, we model a network architecture similar to Scale-Out NUMA [41], which features an integrated NIC and implements a lightweight userspace and hardware-terminated network protocol that uses memory-mapped Queue Pairs (QPs) similar to RDMA. Our evaluation varies the total size of allocated buffers, as it is a critical parameter affecting the network-buffer-related pressure in the LLC and, by extension, the occurrence of network buffer leaks (c.f., §II-C). We report that size for each experiment.

Baseline configurations. We compare three baseline packet injection configurations:

- **DMA**: Conventional Direct Memory Access I/O that places incoming packets directly into DRAM.
- **DDIO**: LLC injection of incoming packets, to a specified number of ways (c.f., §II-A).
- **Ideal-DDIO**: An unrealistic system with a separate infinite LLC only for incoming network packets. The CPU always finds network packets in the LLC and there is zero memory traffic due to network data movements.

Workloads. We use two network-intensive applications: the high-performance MICA key-value store (KVS) and an L3 forwarder network function, both ported to the Scale-Out NUMA transport: MICA from its RDMA-based version (HERD [30]) and L3 forwarder adapted from its stock DPDK version [25]. We additionally employ the X-Mem tool [18] to represent collocated applications competing for LLC capacity and memory bandwidth.

IV. Tracing the Leak to its Source

In this section, we study scenarios of heavy networking load where the problem of network data leaks occurs. We distinguish between the two potential sources of such leaks introduced in §II-B: consumed and premature buffer evictions. We find that consumed buffer evictions are the predominant leak source, while leaks attributed to premature buffer evictions require operation under extreme conditions, and are therefore less prevalent.

A. Consumed Buffer Evictions

Under the ideal operation case for DDIO, all the receive buffers fit in the DDIO ways, and data transfers from the NIC to the CPU do not result in memory access. However, in many realistic scenarios, the aggregate receive buffer footprint exceeds the DDIO capacity, thus the cache blocks required to accommodate an incoming packet may not reside in the LLC at the time of packet arrival. A NIC write miss write-allocates the cache block in the DDIO ways, triggering an eviction. Such evicted cache blocks are most likely in a dirty state compared to main memory, as most blocks in the DDIO ways hold data written earlier by the NIC, and thus require a memory writeback. Compared to the ideal case where no memory trips are required, now, with high probability, every NIC write operation adds memory bandwidth pressure with a writeback.

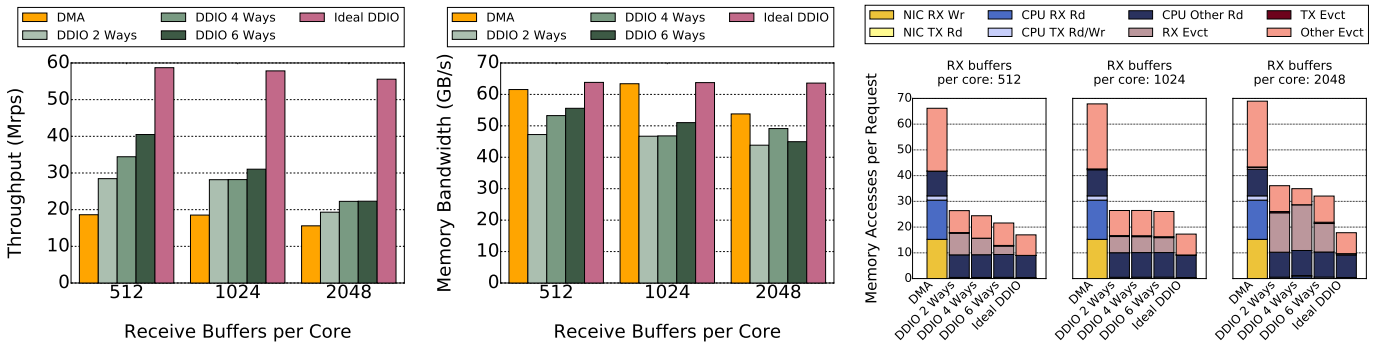
Figure 1 shows the performance (in terms of throughput) and key memory access statistics for our KVS running on all 24 cores of our simulated server. We evaluate a write-heavy workload with 1KB items (details in Appendix A), resulting in commensurate network packet size, and vary the number of allocated receive buffers per core (i.e., ring buffer size in an equivalent DPDK setting) from 512 to 2048. We compare conventional DMA packet injections against three DDIO configurations (2-, 4-, and 6-way; we evaluate up to 12-way DDIO in §VI-A) and ideal-DDIO (c.f., §III).

We start by noting that although Figure 1a may suggest that provisioning fewer buffers is always preferable, as it improves the peak sustainable throughput, *shallow buffering comes at a cost, as it is more vulnerable to undesirable packet loss in the event of packet arrival bursts*. Therefore, buffer provisioning presents a tradeoff: shallow buffering reduces buffer bloat and its performance implications, while deep buffering offers improved resiliency to packet drops. We later show that our proposed solution breaks this tradeoff, preserving high performance regardless of buffer size provisioning (§VI-A).

Figure 1a clearly shows DDIO's benefit, yielding up to 2.1× throughput gains over conventional DMA. DDIO decreases average memory access time (AMAT), which in turn results in reduced average service time and, equivalently, increased throughput. AMAT improves by finding network buffers in the LLC and by spending less time for every memory access, in the less loaded DRAM—we later show the considerable increase in memory access latency caused by DRAM pressure (§VI-B). DMA results in 9–35% higher memory bandwidth utilization than DDIO (Figure 1b), despite operating at a significantly lower application throughput.

Figure 1c's per-request memory access breakdown provides deeper insight into data movement and the resulting memory bandwidth utilization gap between DMA and DDIO. The breakdown attributes memory traffic to its sources: buffer accesses by the CPU and NIC, and application data accesses by the CPU. Figure 1c particularly focuses on providing a fine-grained breakdown among network buffer accesses. "CPU Other Rd" refers to direct CPU accesses to memory locations other than RX/TX buffers, while "Other Evct" refers to evictions of dirty application data from LLC to memory. As Figure 1c demonstrates, DDIO completely eliminates memory traffic directly generated by the NIC (writes on RX path, reads on TX path), as all NIC accesses are serviced from the LLC, reducing memory accesses per request by up to 70%.

Despite the significant performance improvement DDIO achieves over DMA, Figure 1a shows a significant gap between DDIO configurations and ideal-DDIO. As shown in Figure 1c, the main difference stems from the average number of memory accesses performed per application request: DDIO incurs a $1.3-2\times$ data movement premium over ideal-DDIO. Although allocating more ways to DDIO helps shrink that data movement gap, if the entirety of the network buffers cannot be contained in the LLC, there will be excess memory traffic, resulting in performance degradation. To illustrate, 512, 1024, and 2048 receive buffers per core for 1KB packets correspond



(a) Application throughput.

(b) Memory bandwidth utilization at each configuration's peak throughput.

(c) Breakdown of memory accesses per KVS request.

Fig. 1: KVS application demonstrating performance effect of network data leaks.

to 12MB, 24MB, and 48MB, which are equivalent to 33%, 67%, and 133% of the entire LLC capacity. For our 36MB 12-way LLC, 2-, 4- and 6-way DDIO restricts the capacity usable by the NIC to 6MB, 12MB, and 18MB, respectively. As a result, there is significant churn of network data between the LLC and memory, exhibiting itself in Figure 1c as RX buffer evictions (*RX Evct*). Unsurprisingly, increasing DDIO ways helps reduce such churn and improve performance, but even when the network buffers entirely fit in the allocated DDIO ways (e.g., 4- and 6-way DDIO with 512 RX buffers per core), detrimental data movement cannot be completely avoided.

Importantly, Figure 1c highlights the source of network data leaks that introduce performance problems: premature versus consumed buffer evictions. A premature buffer eviction would eventually cause the CPU to miss in the LLC and access memory for the RX buffer—hance, premature evictions are identified in Figure 1c as *CPU RX Rd*. As seen in the figure, even in the most space-constrained 2-way DDIO configuration, memory traffic attributed to premature buffer evictions is negligible. In contrast, *virtually all network data leaks are attributed to consumed buffer evictions* (*RX Evct* in Figure 1c).

B. Premature Buffer Evictions

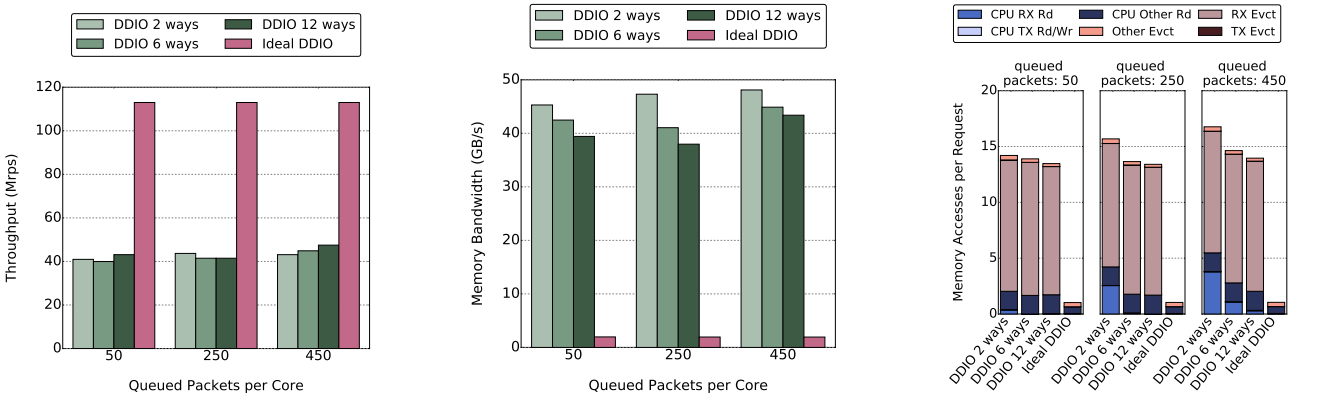
§IV-A demonstrated that consumed buffer evictions are the prevalent source of network data leaks. Unlike premature buffer evictions, consumed buffer evictions do not introduce bottlenecks directly on the application execution's critical path, but rather create system-level contention that introduces queuing effects at the memory. We expect that premature buffer evictions, when they occur, would have an even more profound performance degradation effect, as they result in even higher bandwidth consumption overhead.

A system must be operating under extreme conditions to continuously experience premature buffer evictions. Fundamentally, premature evictions require deep queue buildup of packets waiting to be serviced. As packets in such a queue wait for a prolonged amount of time, their probability of getting evicted from the LLC before being consumed by the CPU increases. On one hand, spurious queue buildups due to packet arrival spikes are possible, but the temporary nature of such events make them less of a concern, at least in terms of sustained throughput (they may still pose a tail latency concern). On the other hand, a steady state of premature buffer evictions would require packet arrival rates exceeding the server's sustainable service rate. Such pathological scenarios fundamentally correspond to unstable, ill-provisioned systems, which are beyond the scope of our study.

A typical modus operandi that can result in steady queue buildup by design is batching of packet processing. For example, that is a common option in the DPDK protocol stack—the protocol stack picks up arrived packets to process in batches of D. To emulate such operation under batching of degree D, we construct scenarios where the RX buffer always contains D unconsumed packets waiting to be processed. We achieve that by modifying our load generator to monitor and ensure that the number of unconsumed packets in each core's RX buffer is always at least D, by injecting new packets. We employ an L3 forwarder network function (L3fwd NF) handling 1KB packets and perform experiments with $D = \{50, 250, 450\}$. As a point of reference, DPDK's default batching degree is 32, but we push to considerably higher batching degrees to gauge the prevalence of premature buffer evictions under more extreme scenarios.

Figure 2 illustrates the same data for this L3fwd experiment as Figure 1 did for the KVS application. Instead of per-core RX buffer depth, the x-axis of Figures 2a and 2b sweeps the D parameter. The per-core RX buffer depth is fixed to 2048 and we evaluate 2-, 6-, and 12-way DDIO configurations.

The trends we observe generally align with those in §IV-A. A major difference is that ideal-DDIO consumes negligible memory bandwidth (Figure 2b), as L3fwd's minimal memory footprint rarely results in memory access. Hence, in this experiment, memory traffic is dominated by network buffer movement and any application data accesses represent a very small fraction of memory accesses per packet processed in Figure 2c. The key observation from Figure 2c aligns with our previous takeaway from Figure 1c: memory traffic is dom-



(a) Application throughput.

(b) Memory bandwidth utilization at each configuration's peak throughput.

(c) Breakdown of memory accesses per packet processed.

Fig. 2: L3 forwarder NF demonstrating performance effect of network data leaks.

inated by consumed buffer evictions. A key difference from Figure 1c is that in this experiment we start seeing a different type of evictions as well: premature buffer evictions, indicated by CPU RX buffer read misses (annotated as *CPU RX Rd* in the figure). However, the magnitude of premature buffer evictions is only considerable in the most constrained 2-way DDIO configuration, and even then, they are overshadowed by consumed buffer evictions.

In conclusion, § IV-A and IV-B's studies show that *consumed buffer evictions are the predominant source of network data leaks*. Although premature buffer leaks can become a problem under heavy queuing conditions, even in such scenarios, consumed buffer evictions are still significant network data leak contributors. Therefore, addressing this type of leaks will have a positive effect in all problematic scenarios.

V. Sweeper Design

Our exploratory analysis in §IV demonstrates that performance implications due to network data leaks mainly stem from consumed network buffer evictions, namely, RX buffers that have already fulfilled their purpose: the data placed in them by the NIC has been consumed by the application. Any reuse to these memory locations will start with a write by the NIC, when the buffer is reused to accommodate a newly arrived packet.

Our first critical insight is that *the data in the cache blocks comprising a consumed RX buffer are dead:* they have served their purpose from the application's perspective, and the CPU will never access the same data again. The memory location will most likely be accessed again by the CPU in the future, but only after the NIC has reused that same buffer location to store *different* data; in other words, the CPU would only access a *different instance* of the RX buffer. At the same time, such dead cache blocks corresponding to RX buffers are guaranteed to be *dirty*. Therefore, if the NIC does not reuse them in the immediate future, they will be evicted from the LLC and will—by default—be written back to memory, introducing the memory bandwidth interference that renders network data leaks a performance concern.

Given there exists a well-defined point in time when the contents of an RX buffer's instance will never again be read by any entity (CPU or NIC), writing this buffer's data back to memory is wasteful. Thus, our second key insight is that cache lines belonging to such a consumed buffer need not be written back to memory and can simply be dropped for bandwidth savings. From a purely hardware perspective, dropping a dirty cache line without writing it back to memory would appear to be a memory corruption. However, if the software *guarantees* that it will not access the contents of the same buffer instance, dropping dirty cachelines without writeback to memory does not raise correctness issues. A read access after such a guarantee has been declared would have undefined behavior, no different from software accessing a buffer it explicitly deallocated earlier by invoking `free()`. Additionally, a critically important observation is that the NIC's future access to write the next *instance* of the RX buffer will not be affected by the omission of the previous instance's writeback to memory, because the NIC directly write-allocates the new data at cache-block granularity, fully overwriting any previous data contents.

Based on these insights, we propose Sweeper, a software-hardware co-design aiming to alleviate memory bandwidth interference due to wasteful evictions of consumed network buffers. The software explicitly declares when the lifetime of a network buffer's instance has expired and exposes this information to the underlying hardware. The hardware then evicts this buffer from the cache hierarchy without triggering a writeback to memory, thus conserving memory bandwidth. Next, we discuss Sweeper's components, including its software API (§V-A) and the associated required ISA and microarchitectural extensions (§V-B). Finally, we walk through Sweeper's end-to-end operation with an example (§V-C).

A. Software Interface

We introduce a function that an application can use to indicate that it relinquishes access to the indicated buffer: `relinquish(buffer_address, size)`. The buffer's contents are considered to be lost after the execution of the function, as the cache blocks comprising the buffer are invalidated from the cache hierarchy without being written back to memory. A networking library that uses `relinquish` must always do so before recycling the buffer for reuse by the NIC, to avoid race conditions. In the most common use of the network stack where the packet is copied from the RX buffer to a local application buffer, the RX buffer can be trivially relinquished after that copy is performed. In systems utilizing zero-copy packet reception, the packet buffers should be relinquished right after the last use by the application, in a manner conceptually similar to freeing a dynamically allocated object as early as possible.

B. Hardware Support

An invocation to the `relinquish` function requires propagating new semantics to the processor and underlying cache hierarchy that are not readily available in current systems. Therefore, both ISA and microarchitectural extensions are required. We introduce `clsweep`, an unprivileged instruction, which takes a single register argument, holding the address of the cache block that must be *swept:* i.e., invalidated without being written back to memory. A `clsweep` triggers a `sweep` message that propagates through the cache hierarchy to invalidate every copy of the target cache block. A `sweep` is a variant of a classic `invalidate` message that sets a cache block's state to Invalid.

Several ISAs provide variants of instructions, similar to `clsweep`, that invalidate a cache block in the cache hierarchy without triggering a writeback. Although the introduction of such an instruction is not a novel contribution of its own, its use as a building block to eliminate detrimental performance effects in high-performance networked systems is new. Across ISAs that feature variants of such invalidation instructions, their common intended use is to support non-coherent producer-consumer communication that interact over DMA—for example, when the consumer wants to invalidate any data in its local cache hierarchy to force a fresh copy of the data to be fetched from the producer's non-coherent memory. Such communication typically involves at least one peripheral device and is orchestrated by the OS, therefore the related instructions are usually privileged.

x86 offers two instructions that are close to `clsweep`'s intended semantics, but neither of them is sufficient for our purpose. `CLFLUSHOPT` [27] invalidates the specified cache block from every level of the cache hierarchy, writing it back to memory if found dirty. The `INVD` [27] privileged instruction empties all caches without writing back dirty data. `clsweep` combines elements of `CLFLUSHOPT` and `INVD`.

Arm offers a rich collection of cache maintenance instructions [3, §C5.3]. `DC IVAC` allows invalidation without writeback of a specific virtual address, but can only be used in

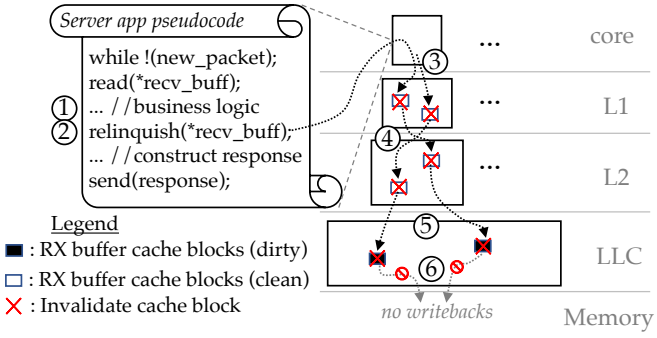


Fig. 3: Sweeper's buffer cleaning.

privileged mode. The same capabilities and restrictions apply for POWER's `CLI` [24] and TILE-Gx's `Inv` [56] instruction. `CBO.INVAL` from RISC-V's recent extension for cache management operations (Zicbom) appears to match the desired semantics, but its required privilege level is unclear [48]. All these instruction instances indicate that the required mechanisms to realize `clsweep` are largely in place and are not disruptive to coherence or other critical microarchitectural components. Sweeper presents a novel use case for such instructions, motivating their use in a new context.

Correctness and security concerns. Careless use of the `clsweep` instruction can result in memory corruption of the user's process, but that effect is similar to a memory bug (e.g., accessing a dynamically allocated object after it has been freed). Special care is required when the operating system reclaims a page and allocates it to a different process, as it could result in a privacy breach. Although the operating system zeroes out ("resets") the page before transferring ownership between processes, the new owner process could read the previous one's values in that page by invoking `clsweep` to prevent the page's newly zeroed out values from propagating to memory.

There are several possible solutions to address such privacy breaches. First, if the OS zeroes out pages by scheduling a conventional DMA that does not make use of DDIO, the aforementioned concern does not arise. If zeroing out a page before transferring ownership to a new process involves caching the page's zeroed out cache blocks, a kernel extension to `CLWB` all of a page's cache blocks right after resetting their value to zero again mitigates the privacy concern. Instead of doing this for *every* newly allocated page, the writeback could be enforced only for pages that are allocated to processes that make use of `clsweep`. Distinguishing processes as such would require marking a process' control block accordingly, after the process initially uses a new dedicated system call that requests permission for use of `clsweep` in userspace.

C. Sweeper in Action

Figure 3 demonstrates Sweeper's operation, which saves excess memory bandwidth usage by avoiding writing consumed network buffers back to memory, after taking explicit hints from the software. The top left core executes a networked

server application, shown as a pseudocode snippet. At step ①, the following sequence of events has already happened:

(i) The NIC write-allocated an incoming network packet (here assumed to occupy two cache blocks) into the LLC. The corresponding cache blocks are therefore dirty.

(ii) The core identified the packet arrival and read the RX buffer from the LLC, bringing clean copies of the corresponding cache blocks into its private L1 and L2 caches.

After the application is done reading the contents of the RX buffer, it uses our introduced `relinquish` function (§V-A) (step ②). Note that this does not imply the application can only read the RX buffer once; rather, the application relinquishes the buffer after the *last time* it uses that buffer instance. By executing this function call, the application declares that no entity, CPU or NIC, will read this buffer again before the NIC overwrites it with a new packet's data. The function call is compiled into a set of `clsweep` instructions, one per cache block comprising the target buffer, which in turn inject `sweep` messages into the cache hierarchy (step ③). As the `sweep` messages propagate down the cache hierarchy, they invalidate the corresponding cache blocks found in each of the cache levels—L1, L2, and LLC (steps ③ to ⑤). Step ⑥ demonstrates that a dirty cache block invalidated by a `sweep` does not trigger a writeback to memory, conserving memory bandwidth.

D. Sweeper on the Transmit Path

We have so far only focused on network data movement implications on the RX path, as this is where data buffer bloat commonly results in network data leaks and performance problems. Similar complications may arise on the transmit path, either due to overprovisioned transmit buffers, or because of applications implementing a *zero-copy receive-to-transmit* optimization scheme, using the RX buffer where a packet was received to transmit the same packet back to the network. The latter is a pattern often implemented by Network Functions (NFs): the NF performs in-place operations on the packet received in an RX buffer, and then passes a pointer to this modified RX buffer to the NIC, thus avoiding an explicit copy from the RX buffer to a transmit buffer prior to the packet's transmission. This optimization is very specific to NFs that transmit the same packets they receive, only minimally modified. NFs employing such zero-copy packet transmission cannot directly leverage Sweeper in the form described earlier in this section, because, in this mode of operation, the CPU is not the last entity to access an RX buffer during its lifetime. The RX buffer only becomes dead after the NIC reads it on the transmit path. Thus, on the transmit path, it is the NIC—and not the application—that must declare the buffer's lifetime expiration.

Our proposed approach is equally applicable when network data leaks emerge and become problematic on the transmit path. Sweeper's design can be slightly adapted for use in such scenarios, whereby buffer sweeping is initiated by the NIC instead of the CPU, with the core of the mechanism remaining the same. Sweeping relies on the same `sweep` coherence message (§V-B), but an additional software API is required for instructing the NIC to initiate a buffer sweep after transmitting its contents and releasing it. Such API is less intrusive than an ISA extension. All that is necessary is an additional boolean `SweepBuffer` field in the memory-mapped Work Queue that the CPU uses to schedule packet transmissions, as exemplified in Figure 4 for a sample Work Queue entry of the Scale-Out NUMA protocol (similar to RDMA). The CPU may opt to set this field when it creates the Work Queue entry (step ①). When the NIC reads the Work Queue entry (step ②), it checks that field. Once the transmission of the Work Queue entry's corresponding transmit buffer completes (step ③), and before releasing the buffer to be reused by the CPU at a later time, the NIC injects `sweep` coherence messages into the cache hierarchy, for the cache blocks comprising the buffer (step ④).

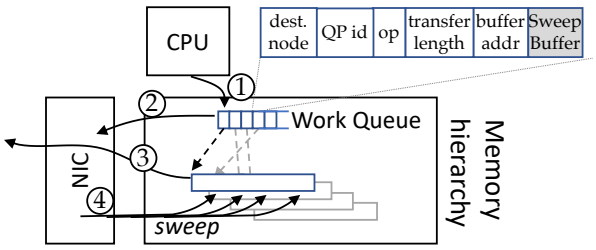


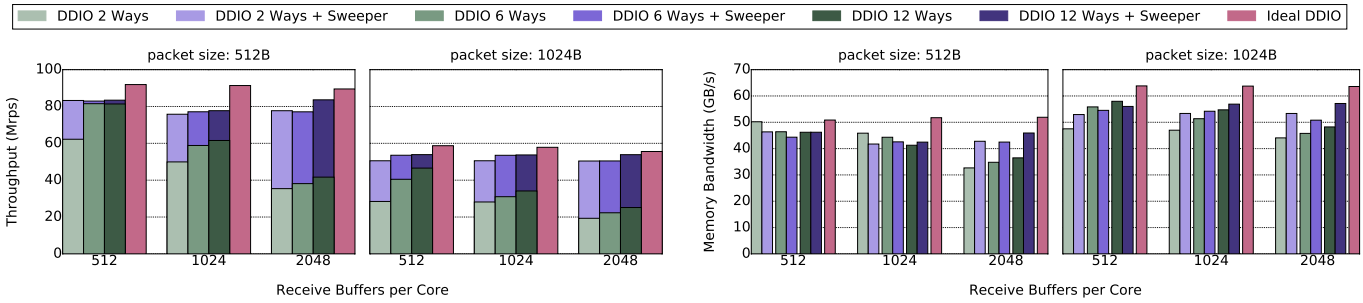
Fig. 4: Sweeper's NIC-driven buffer cleaning for zero-copy networking.

Instead of offloading TX buffer sweeping to the NIC, and alternative approach would be to delegate this responsibility back to the CPU, after the packet's transmission. The required protocol modifications would be similar, but on the inverse direction (i.e., NIC notifies CPU to sweep after transmission, via a Completion Queue entry). The drawback of this alternative approach is that such hand-off introduces timeliness implications, as the core's reaction time to the NIC's sweeping hint depends on software events and the CPU's occupancy.

For the sake of clarity and brevity, our evaluations are focused on demonstrating Sweeper's ability to alleviate problems arising from network data leaks on the *ingress path*. Therefore, our evaluation in §VI does not include results for instances with transmit buffer bloat, or with applications that use the aforementioned receive-to-transmit zero-copy mechanism occurring in certain instances of NF processing. The key takeaways and results are equally applicable to such cases, by leveraging the additional support for Sweeper described in this subsection.

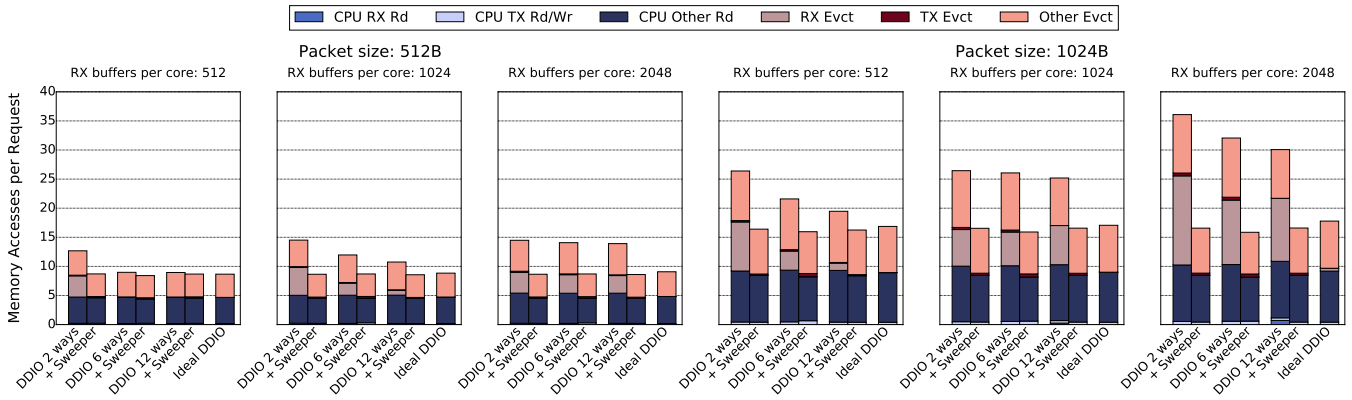
VI. Evaluation

In this section we extend the evaluations presented in §IV to provide a sensitivity analysis of key system parameters that affect the onset and magnitude of network data leaks, along with their impact on performance. In addition, we demonstrate the benefits of our proposed Sweeper technique. We implement Sweeper in our simulation infrastructure and



(a) Application throughput.

(b) Memory bandwidth utilization at each configuration's peak throughput.



(c) Breakdown of memory accesses per KVS request.

Fig. 5: Effect of DDIO ways allocation on network data leaks and KVS performance.

base our evaluation on the methodology previously covered in §III, with further details available in Appendix A.

Sweeper improves performance by eliminating network data leaks due to consumed buffer evictions. We quantify its impact in §VI-A. We then demonstrate that Sweeper's bandwidth savings also translate to improved average memory access time in §VI-B. Next, §VI-C repeats §IV-B's scenario and illustrates that Sweeper is beneficial even in the presence of premature buffer evictions. We proceed to showcase that Sweeper remains an effective performance-boosting mechanism even when available memory bandwidth is overprovisioned in §VI-D. We then employ a scenario of collocated applications to verify that Sweeper's benefit also extends to non-networked applications in §VI-E. Finally, §VI-F demonstrates that the alternative approach of shallow buffering to mitigate network data leaks results in increased packet drop rates.

A. Sensitivity to DDIO Configuration

The root cause of network data leaks is insufficient LLC capacity to hold the entirety or majority of all used network buffers. Thus, a straightforward first attempt to alleviate the problem would be to allocate additional DDIO ways. We employ the MICA KVS from §IV-A to study its sensitivity to the number of DDIO ways, and evaluate the effect of Sweeper in each case. In this study, we also vary the KVS item size, resulting in commensurate sizes for packets carrying data. We evaluate scenarios with 512B and 1KB items.

Figure 5 shows our KVS' peak performance when DDIO is allocated 2, 4, 6 or 12 ways. Note that the 12-way configuration allows DDIO to write-allocate in any of the LLC ways without restrictions. As the allocated space for DDIO grows, the application's performance—measured in Millions of requests per second (Mrps)—improves (Figure 5a), because the rate of RX buffer evictions to memory diminishes, as evidenced in Figure 5c. Fewer consumed buffer evictions directly correlate to higher application throughput, with 12-way DDIO outperforming 2-way DDIO by $1.2-1.6\times$. Still, the best-performing 12-way DDIO configuration only approaches ideal-DDIO for the smallest 512-buffers/512B-packet scenario (Figure 5a leftmost bar cluster), which corresponds to a 6MB buffer footprint that can comfortably fit in the 36MB LLC. In every other case, even 12-way DDIO achieves $11-55\%$ lower throughput than ideal-DDIO. The performance obstacle is excess data movement, as each processed request triggers up to $2\times$ more memory accesses (Figure 5c), resulting in higher memory bandwidth consumption per unit of work.

Figure 5c shows that Sweeper completely eliminates write-backs of consumed RX buffers (*RX Evct* in the figure), virtually matching ideal-DDIO's memory access count per KVS request. In turn, that reduction in memory bandwidth pressure yields drastic performance improvements over DDIO, as illustrated in Figure 5a. When applied on top of each DDIO configuration, Sweeper boosts performance by $1.02-2.6\times$, delivering performance within $2-18\%$ of ideal-DDIO. The

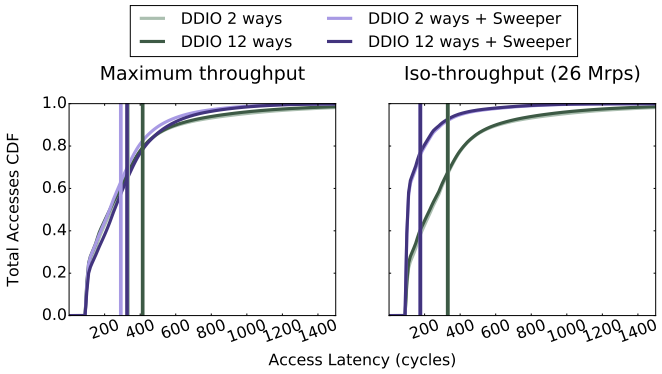
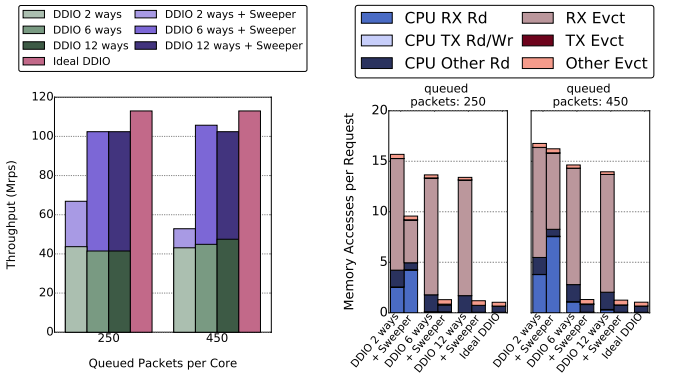


Fig. 6: Memory access latency CDFs for the KVS application. Left: at each configuration's peak load. Right: iso-throughput comparison, at 2-way DDIO configuration's achieved peak. Vertical lines mark the respective curve's effective average memory access latency.



(a) Application throughput.

(b) Breakdown of memory accesses per packet processed.

Fig. 7: Sweeper's effect on premature buffer evictions.

remaining performance gap is attributed to the fact that, with ideal-DDIO, RX buffers do not consume *any* capacity in the entire multi-level cache hierarchy, yielding exclusive use of all the cache resources to application data.

Ultimately, Sweeper enables maximum performance and is largely insensitive to RX buffer provisioning, in stark contrast to baseline DDIO, whose performance noticeably degrades as RX buffer provisioning grows. Sweeper thus *breaks the tradeoff of choosing between shallow buffering for higher steady-state performance and deep buffering for packet drop resilience.* Sweeper allows deployment of deep buffers to absorb heavy packet arrival bursts without paying a performance cost for that added resilience.

B. Effect on Memory Access Latency

As mentioned in §IV-A, a key benefit of reduced memory bandwidth utilization is lowered average memory access time, which in turn reduces service time and increases throughput. To illustrate this effect, Figure 6 shows the CDFs of memory access latency (i.e., DRAM access time) for the 1024 buffers of 1KB packets scenario, corresponding to Figure 5a's fifth bar cluster.

Figure 6(left) compares the resulting memory access latency distribution for 2- and 12-way DDIO, with and without Sweeper. For both DDIO configurations, Sweeper noticeably reduces both average and tail memory access latency, despite operating at higher application throughput. For 2-way/12-way DDIO, Sweeper reduces average latency by 12%/21% and p99 latency by 3%/6%, while achieving $1.57-1.79\times$ higher application throughput (c.f. Figure 5a).

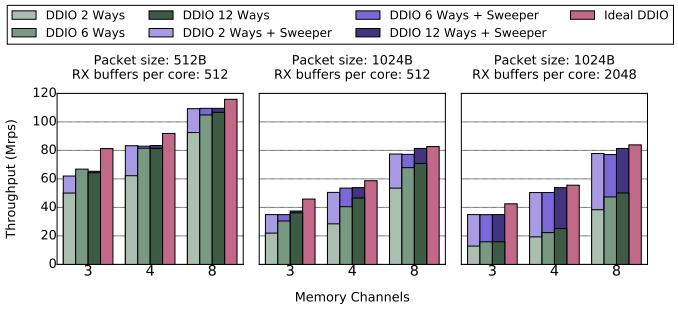
Figure 6(right) compares the resulting memory access latency distribution for the same four configurations, but at the 2-way DDIO configuration's peak achieved throughput of 26Mrps. Sweeper's elimination of unnecessary memory traffic reduces average and p99 memory access latency by 47% and 20% respectively.

C. Sweeper with Premature Buffer Evictions

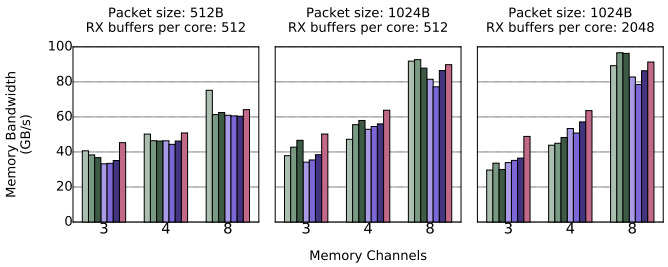
Sweeper mitigates performance implications of memory bandwidth waste due to consumed buffer evictions. As our study in §IV showed, although premature buffer evictions are less common, in some cases they are also a considerable memory bandwidth consumption contributor. We now revisit the two specific L3fwd scenarios with deep queues of unconsumed network packets from §IV-B that exhibited noticeable premature buffer evictions.

Figure 7 shows the performance and breakdown of memory accesses for the two scenarios of interest, including results with Sweeper. Overall, Sweeper reduces the number of memory accesses per processed packet (Figure 7b), resulting in performance improvements of $1.2-2.4\times$ (Figure 7a). Figure 7b reveals that Sweeper completely eliminates consumed buffer evictions in every DDIO configuration. That is evidenced by the fact that, with Sweeper enabled, the remaining RX evictions (*RX Evct*) exactly match the CPU's misses to RX buffers (*CPU RX Rd*). Hence, all remaining network-related data movements between the LLC and memory are attributed to premature buffer evictions, as every RX buffer that is evicted is later accessed by the CPU.

The memory breakdown behavior (Figure 7b) for the 2-way DDIO configuration is an interesting outlier, with Sweeper increasing *CPU RX Rd* misses compared to plain DDIO, implying more premature buffer evictions. This behavior is attributed to network buffers spillover from the DDIO ways: as network buffers are prematurely evicted from the LLC, they are brought back to non-DDIO LLC ways when later accessed by the CPU and stay there, effectively increasing the LLC fraction occupied by network buffers. In contrast, Sweeper cleans such runaway buffers after their use, thus truly containing network buffers to two ways. The increased contention then results in more premature evictions. However, even in this case, Sweeper reduces the aggregate added network-data-related memory traffic, leading to an overall performance gain.



(a) Application throughput.



(b) Memory bandwidth utilization.

Fig. 8: Effect of network data leaks and Sweeper on performance as a function of memory bandwidth availability.

D. Sensitivity to Memory Bandwidth

As demonstrated in §VI-A, memory bandwidth utilization is a critical performance determinant for network-intensive applications resulting in network data leaks. Our evaluation so far was based on a 24-core server CPU provisioned with four DDR4 channels, which is a medium-range setup: memory bandwidth provisioning in modern server-grade CPUs typically ranges from one DDR4 channel per eight cores (e.g., AMD EPYC [22]) to one per three cores (e.g., Xeon Gold 6342 [29]). These bandwidth-to-core ratios correspond to 3–8 memory channels in our simulated 24-core CPU.

Figure 8 shows a performance sensitivity analysis as a function of available memory bandwidth, controlled by provisioning 3, 4, and 8 memory channels. Results are based on the MICA KVS application. The general trends observed in §VI-A's thorough analysis hold across memory channel configurations. Increasing DDIO ways improves the KVS performance, but still lags behind ideal-DDIO due to excess memory bandwidth usage. Sweeper noticeably boosts performance even in the system provisioned with 8 memory channels and the smallest RX buffer footprint configuration (512 buffers per core and 512B packets—Figure 8a left). Unsurprisingly, the larger the buffer configuration, the more pronounced Sweeper's impact is. For the largest configuration (2048 buffers per core and 1KB packets), Sweeper on a system with 4 memory channels boosts application throughput by $2.1-2.6\times$, as previously seen in Figure 5a. For this configuration, Sweeper's relative performance boost grows to $2.2-2.7\times$ when memory controllers are reduced to 3, and drops to $1.6-2\times$ when increased to 8. In conclusion, although higher memory bandwidth provisioning partially mitigates the performance impact of network data leaks, Sweeper's mitigation of wasteful data movement delivers significant performance improvements even in systems with highly provisioned memory bandwidth.

E. Application Collocation Scenarios

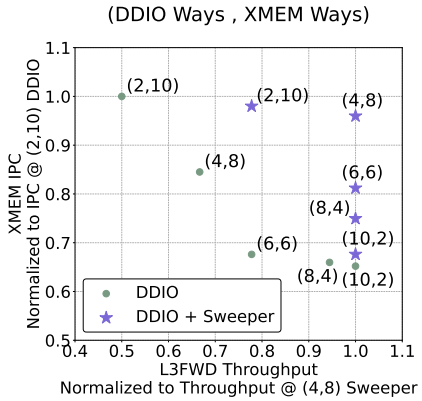
§VI-A's results indicate that the fewer the LLC ways allocated to DDIO, the more pronounced Sweeper's effect. Equivalently, Sweeper's relative performance improvement diminishes, as DDIO is configured to use more LLC ways. However, maxing out DDIO ways is not always the optimal choice. Although that is usually the case for network-intensive applications running in isolation, like our evaluated KVS, doing so in a multi-tenant scenario can hurt the performance of collocated applications. Therefore, limiting DDIO ways is desirable, as hinted by DDIO's default two-way setting. To demonstrate this case, we evaluate a collocated scenario.

We use §IV-B's L3fwd application as a network-intensive tenant, and X-Mem [18] as a memory-intensive tenant. We run 12 instances of each, matching the total number of cores. L3fwd's allocates 2048 RX buffers per core and handles 1KB packets, and its dataset is L1-resident, so any pressure it generates in the LLC or memory is due to packet RX/TX. Each X-Mem process performs sequential random accesses to a private 2MB dataset, which exceeds the aggregate capacity of private L1 and L2 caches.

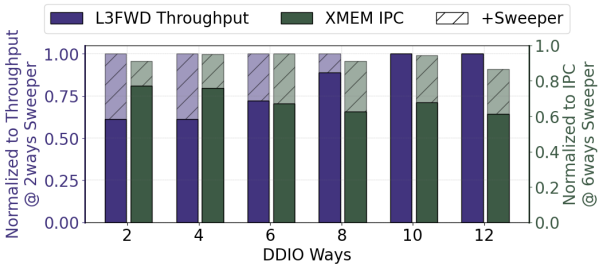
Figure 9 shows results for two collocated application scenarios. The first scenario (Figure 9a) isolates X-Mem threads and DDIO ways hosting the L3fwd's network traffic in non-overlapping LLC partitions. We partition the LLC into two disjoint parts A and B and assign DDIO ways to partition A, X-Mem ways to partition B. We evaluate six (A, B) configurations, with the invariant $A+B=12$. For instance, (2, 10) means that DDIO is assigned 2 ways, and X-Mem is restricted to the remaining 10 ways.

Figure 9a shows the normalized performance of the two applications on a 2-D plane. As we reallocate more cache ways from X-Mem to DDIO, L3fwd performance increases and X-Mem's performance drops. As X-Mem's dataset is squeezed out of the LLC, its performance deteriorates by up to 35%, due to a $12\times$ increase in LLC miss ratio. The performance points on the 2-D plane form a Pareto frontier—the closer to the top right corner, the better. Sweeper noticeably improves both L3fwd's and X-Mem's performance, as its entire performance frontier is closer to the top right corner than plain DDIO's frontier. The most balanced configuration for the two applications is (4, 8). In this configuration, Sweeper boosts L3fwd's and X-Mem's performance by $1.5\times$ and $1.14\times$, respectively.

In the second collocation scenario (Figure 9b), we always allow X-Mem to use the entire LLC, and gradually increase the DDIO ways from 2 to 12. As expected, X-Mem's performance degrades as DDIO's expansion evicts more of its data from the LLC. Sweeper boosts X-Mem's performance by $1.18-1.42\times$. Without Sweeper, 2-way DDIO results in 39% lower L3fwd performance than with 12-way DDIO. With Sweeper, L3fwd



(a) Collocation with non-overlapping LLC way partitions.



(b) Collocation with overlapping LLC way partitions.

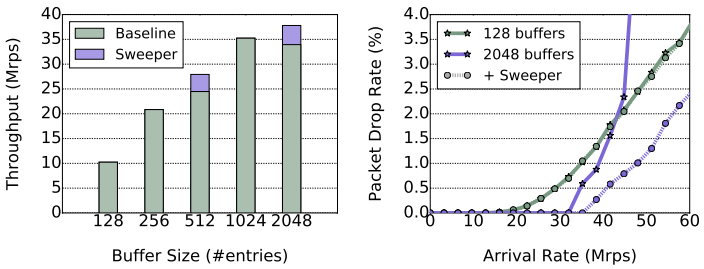
Fig. 9: Performance of collocated network- and memory-intensive applications.

remains completely insensitive to the allocated DDIO ways, achieving its maximum performance with as few as two ways.

F. Shallow Buffering

An alternative approach to mitigate network data leaks is to employ shallow buffering, like ResQ [57]. Limiting the number of available buffers to contain them in the LLC introduces tradeoffs, as it can limit a server's peak throughput and result in increased packet drops. For example, a core with 1024 descriptors forwarding packets at 100Gbps must pause senders every 500 packets to avoid drops [12, §3.1]. Optimal a priori RX buffer sizing to find the sweet spot of footprint and performance is challenging, especially in large-scale systems, and, even when possible, it requires application profiling and high level of expertise. Furthermore, optimal sizing for an application in isolation won't hold when collocation conditions on the same server change. Sweeper alleviates this optimization burden, by allowing application deployment with the benefits of deep buffers and without their performance implications.

We construct a simple scenario to demonstrate the drawbacks of shallow buffering. With the KVS workload as a base, we develop a microbenchmark where, with a small probability, each request suffers a processing delay randomly sampled from the [1,100]µs range, causing temporal queue buildup spikes—an effect also functionally equivalent to packet arrival bursts. Request packets are 1KB and we evaluate a range of



(a) Peak throughput. (b) Packet drop rate.

Fig. 10: Effect of buffer provisioning on performance of workload with spiky behavior.

buffer depths per core using the default 2-way DDIO configuration. Note that among the buffer provisioning configurations evaluated, both 128 and 256 entirely fit in the allocated DDIO space.

Figure 10a shows the peak throughput achievable without packet drops, demonstrating the benefits of deeper buffers. Shallow buffering handicaps peak throughput. While more buffers generally improve robustness, we observe a drop when growing from 1024 to 2048 buffers, as bandwidth interference from network data leaks overshadows the gains of further resilience. However, Sweeper's addition alleviates that problem, pushing throughput even higher.

Figure 10b shows the fraction of packet drops as a function of request arrival rate, for 128 and 2048 RX buffers per core. Deep buffering delivers 3.3× higher throughput without packet drops compared to shallow buffering, which grows to 3.7× with Sweeper. Without Sweeper, shallow buffering appears to sustain higher arrival rates once packet drop rate exceeds 2%. However, such drop rate is already unacceptably high for most realistic applications. Packet drops in datacenters critically disrupt performance [59]. Typical drop rates are in the 10^{-5} range [21] and anything near 1% is considered extraordinarily high and potentially prohibitive [20], [60].

In conclusion, sizing buffers to fit within DDIO does not deliver best performance. Sweeper delivers performance robustness for applications with spiky behavior and—unlike ResQ—achieves that without any expert buffer sizing or empirical profiling. Therefore, Sweeper is a versatile technique that can facilitate deployment of applications with intensive network activity, while improving performance robustness with only minimal additional developer effort.

VII. Related Work

The problem of network data leaks we address has been identified in several prior works and is often referred to as "leaky DMA" [12], [16], [40], [44], [53], [57], [58]. ResQ [57] mitigates leaky DMA by shrinking the network stack's RX ring buffer. NeBuLa [53] targets applications with extremely tight tail-latency SLO and implements a specialized architecture featuring a hardware mechanism that proactively drops packets when the queue depth of waiting packets is such that will lead to SLO violation. This policy implicitly bounds

network buffer occupancy in the LLC, preventing network data leaks to memory. Constraining data buffer depth is an effective solution to the leak problem in some cases, but is not generally applicable. Shallow buffer provisioning runs the risk of undesirable packet drop on packet arrival bursts. Sweeper breaks that tradeoff, by removing the performance implications of network data leaks in presence of large network buffers.

IAT monitors and dynamically reallocates LLC ways between DDIO and all collocated applications to minimize the performance impact of leaky DMA and maximize overall system performance [58]. IDIO [1] selectively steers incoming packets to the large private L2 cache of modern server CPUs, effectively expanding the cache capacity network buffers can leverage, beyond a few LLC ways. Sweeper is orthogonal to such techniques, which dynamically expand the aggregate cache capacity network buffers can occupy.

Several prior works characterize architectural limitations of direct cache injections of network traffic in the cache hierarchy [34], [38], [52]. Farshin et al. [12] study DDIO behavior in depth and demystify the mechanism's low-level operational details. To the best of our knowledge, no prior work identifies that the primary contributor of performance-hampering network data leaks are due to consumed network buffers, which can be avoided to recoup the associated performance loss.

Other data movement optimizations. Self-invalidation cache block and page mechanisms have been used in the context of shared-memory multiprocessors and DSMs to reduce coherence overheads [35], [37]. A similar approach has been proposed to mitigate TLB shootdown costs [5]. Dead-block prediction is a general technique used to make better caching and prefetching decisions, based on predicted data reuse patterns [32], [36]. Sweeper does not simply invalidate buffers, but avoids writing them back to memory, and only does so after explicit software hints rather than predictions, as any wrong prediction would result in memory corruption.

In the context of network data movement and network function processing optimization, CacheDirector [11] places each packet's header in the LLC slice that is closest to the processing core, improving tail latency. PayloadPark [17], nicmem [45], and NFSlicer [49] share a conceptually similar data movement reduction optimization to improve the efficiency of shallow network function handling. These solutions minimize the movement of packet payloads, based on the insight that CPUs executing shallow NFs only manipulate packet headers, and temporarily store packet payloads in switch and NIC memory, respectively. Sweeper also targets overheads associated with excess data movement, particularly focusing on the effect of memory bandwidth contention rather than network and PCIe bandwidth, and has a broader scope, as it is not limited to shallow NFs.

VIII. Conclusion

Growing network rates are promoting the efficiency of data movement within a server's memory hierarchy to a major performance determinant. Modern DDIO technology alleviates data movement overheads by attempting to avoid memory involvement and contain network data traffic within the server's LLC. However, prior work has demonstrated several instances of significant network data leaks into memory, causing considerable memory bandwidth interference and, in turn, performance degradation.

In this work, we study the problem of network data leaks and identify that a major contributor to bandwidth interference is the eviction of dirty buffers that have already been consumed by the application. Writebacks of these buffers are both costly and wasteful, as they will typically not be read again before being fully overwritten by the NIC with a new incoming packet. We propose Sweeper, a hardware extension and API that allows applications to mark network buffers that have been conclusively consumed, indicating that their writeback to memory can be safely omitted. By omitting consumed buffer writebacks, Sweeper significantly reduces memory bandwidth consumption by up to 1.3×, resulting in up to 2.6× higher application throughput. Sweeper breaks a tradeoff in network buffer provisioning, as it allows allocating large buffers to minimize the probability of packet drops without resulting in increased network data leak rates typically caused by larger buffers.

Sweeper's key strength is its simplicity. It has practical value, as the hardware extensions required are minimal, without involving any new disruptive low-level operation within the memory subsystem and coherence mechanism. On the software front, applications require very limited modifications to benefit from Sweeper, akin to memory management most programmers are familiar with. Sweeper is therefore well-positioned for commercialization by mainstream CPU vendors.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their suggestions that improved this paper, including pointing out the potential privacy perils from allowing the use of `clsweep` in userspace without additional precautions. We also thank Moin Qureshi, Mark Sutherland, Anirudh Sarma, Hamed Seyedroudbari, and Divya Kadiyala for their constructive feedback on early drafts of the paper. This work was partially supported by a Georgia Tech School of Computer Science Incubator Graduate Fellowship and by the National Science Foundation under the award NSF-CCF-2006602.

Appendix

This appendix complements §III's methodology with additional details of our experimental setup. We extend the zSim simulator [54] with a NIC component implementing the Scale-Out NUMA userspace, hardware-terminated protocol [41] and with a traffic generator that injects packets at configurable Poisson arrival rate. We simulate a 24-core server-grade CPU, modeled after Intel Xeon Gold 6342 [29]. Table I summarizes key parameters of the simulated system.

Workload configurations. We use the MICA KVS throughout the paper's evaluations (§ IV-A, VI-A, VI-B, VI-D and VI-F)

with a write-heavy 5/95 GET/SET workload and medium to large items (512B or 1KB), as specified in each experiment. The MICA KVS is populated with 2.4M key-value pairs, and uses 1M buckets and a 256MB circular log. Requests for data items follow a zipf 0.99 key popularity distribution.

We use an L3 forwarder network function (L3fwd) as a representative of a network-intensive application with minimal service time and small dataset in §IV-B and §VI-E. In §VI-E's experiments, L3fwd's dataset fits in the L1 cache, introducing negligible LLC and memory accesses. In §IV-B's and §VI-C's experiment that aim to increase LLC pressure to demonstrate premature buffer evictions, L3fwd uses 16k forwarding rules, which barely fit in each core's private L2 cache. Both the MICA KVS and L3fwd were ported to the Scale-Out NUMA [53] protocol for our evaluation. Finally, we use X-Mem [18] as a memory-intensive application in our evaluated collocation scenarios (§VI-E).

With the exception of §IV-B, §VI-C, and §VI-F, which purposely push into deep buffering operational regions with heavy queuing to study premature buffer evictions, all results refer to peak application throughput achieved under a p99 latency SLO of 100× the deployed workload's average service time; hence, reported throughput improvements achieved by Sweeper over the respective baseline system refer to the same SLO target.

Buffer sizing. Buffer sizing affects both packet drops and network data leaks from the LLC to memory. Buffers must be sized to accommodate at minimum the bandwidth-delay product between the communicating endpoints, but also offer resilience to arrival bursts and service time spikes. DPDK's default RX size used to be 128 at the 1G NIC era, with 1024 being a value typically encountered today. The reduced opportunity for buffer multiplexing in user-level stacks exacerbates the need for more buffers, resulting in overprovisioning as a function of communicating endpoints (§II-C). To exemplify, a server with 1024 total RX entries communicating with 256 clients means each client can only have four packets in flight. Alternatively, if each of these 256 endpoint pairs provisions an RX ring to allow 128 packets in flight, the server's aggregate RX size grows to 32k.

High-performance systems often resort to per-core buffer provisioning, to mitigate inter-core synchronization [6], [8], [46], [53]. Given the latency-sensitive applications we study, we provision $B \in [512, 2048]$ network buffers per core, but also study effects of shallower buffering in §VI-F. While we set up experiments with each core handling a single RX ring of size B, the setup is also indicative of alternative scenarios where each core manages N M-deep RX rings with $N \times M = B$ (e.g., a core handling N InfiniBand connections).

CPU	24 x86-64 cores modeled after Ice Lake, 3.2GHz, OoO, 5-wide dispatch/retirement, 352-entry ROB
L1 Caches	Split L1d/i, 48/32KB 12/8-way, 64B blocks, 4-cycle access
L2 Caches	1.25MB, 20-way, 14-cycle access
LLC	Shared non-inclusive operating as victim cache for L2 evictions [28], 36MB, 12-way, 35-cycle access
NoC	Crossbar, 8-cycle latency
Memory	DDR4-3200, 3 to 8 memory channels 4 ranks per channel, 8 banks per rank Configuration parameters from Ramulator [33], [47]

TABLE I: System parameters for simulation on zSim.

References

[1] 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, pp. 30–33, 2021.

[2] 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)*, 2020, pp. 160–169.

[3] Arm Limited, "Arm® Architecture Reference Manual for A-profile architecture," https://developer.arm.com/documentation/ddi0487/ha/?lang=en, accessed August 2022.

[4] Arm Limited, "Arm DynamIQ Shared Unit Technical Reference Manual r3p0," 2018, https://developer.arm.com/documentation/100453/0300/functional-description/l3-cache/cache-stashing, accessed August 2022.

[5] A. Awad, A. Basu, S. Blagodurov, Y. Solihin, and G. H. Loh, "Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries," in *Proceedings of the 26th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2017, pp. 273–287.

[6] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane," *ACM Trans. Comput. Syst.*, vol. 34, no. 4, pp. 11:1–11:39, 2017.

[7] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 7:1–7:13.

[8] A. Daglis, M. Sutherland, and B. Falsafi, "RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 35–48.

[9] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414.

[10] D. Dunning, G. J. Regnier, G. L. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," *IEEE Micro*, vol. 18, no. 2, pp. 66–76, 1998.

[11] 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 2019 EuroSys Conference*, 2019, pp. 8:1–8:17.

[12] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostic, "Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks," in *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020, pp. 673–689.

[13] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 51–66.

[14] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 3–18.

[15] D. Gibson, H. Hariharan, E. Lance, M. McLaren, B. Montazeri, A. Singh, S. Wang, H. M. G. Wassel, Z. Wu, S. Yoo, R. Balasubramanian, P. Chandra, M. Cutforth, P. Cuy, D. Decotigny, R. Gautam, A. Iriza, M. M. K. Martin, R. Roy, Z. Shen, M. Tan, Y. Tang, M. Wong-Chan, J. Zbiciak, and A. Vahdat, "Aquila: A unified, low-latency fabric for datacenter networks," in *Proceedings of the 19th Symposium on Networked Systems Design and Implementation (NSDI)*, 2022, pp. 1249–1266.

[16] H. Golestani, A. Mirhosseini, and T. F. Wenisch, "Software Data Planes: You Can't Always Spin to Win," in *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*, 2019, pp. 337–350.

[17] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. I. Seltzer, "Parking packet payload with P4," in *Proceedings of the 2020 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, 2020, pp. 274–281.

[18] M. Gottscho, S. Govindan, B. Sharma, M. Shoaib, and P. Gupta, "X-Mem: A cross-platform and extensible memory characterization tool for the cloud," in *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 263–273.

[19] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," *Commun. ACM*, vol. 54, no. 3, pp. 95–104, 2011.

[20] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over Commodity Ethernet at Scale," in *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016, pp. 202–215.

[21] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis," in *Proceedings of the ACM SIGCOMM 2015 Conference*, 2015, pp. 139–152.

[22] L. Gwennap, "AMD Milan Extends Server Lead," *Linley Group Microprocessor Report*, March 2021.

[23] R. Huggahalli, R. R. Iyer, and S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005, pp. 50–59.

[24] IBM, "cli (Cache Line Invalidate) instruction," https://www.ibm.com/docs/en/aix/7.2?topic=set-cli-cache-line-invalidate-instruction, accessed August 2022.

[25] Intel Corporation, "Data plane development kit (DPDK)," https://www.dpdk.org, accessed August 2022.

[26] Intel Corporation, "Intel Data Direct I/O Technology," 2012, https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html, accessed August 2022.

[27] Intel Corporation, "CLFLUSHOPT—Flush Cache Line Optimized," 2016, intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L – https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf, page 3-141, accessed August 2022.

[28] Intel Corporation, "Intel Xeon Processor Scalable Family Technical Overview," 2019, https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, accessed August 2022.

[29] Intel Corporation, "Intel Xeon Gold 6342 Processor," 2021, https://www.intel.com/content/www/us/en/products/sku/215276/intel-xeon-gold-6342-processor-36m-cache-2-80-ghz/specifications.html, accessed August 2022.

[30] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the ACM SIGCOMM 2014 Conference*, 2014, pp. 295–306.

[31] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs," in *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, 2016, pp. 185–201.

[32] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling Dead Block Prediction for Last-Level Caches," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 175–186.

[33] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, 2016.

[34] A. Kumar, R. Huggahalli, and S. Makineni, "Characterization of Direct Cache Access on multi-core systems and 10GbE," in *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2009, pp. 341–352.

[35] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000, pp. 139–148.

[36] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, 2001, pp. 144–154.

[37] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, 1995, pp. 48–59.

[38] G. Liao, X. Zhu, and L. N. Bhuyan, "A new server I/O architecture for high speed networks," in *Proceedings of the 17th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2011, pp. 255–265.

[39] T. Ma, T. Ma, Z. Song, J. Li, H. Chang, K. Chen, H. Jiang, and Y. Wu, "X-RDMA: Effective RDMA Middleware in Large-scale Production Environments," in *Proceedings of the 2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–12.

[40] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018, pp. 327–341.

[41] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014, pp. 3–18.

[42] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir, and M. K. Aguilera, "Storm: a fast transactional dataplane for remote data structures," in *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019, pp. 97–108.

[43] NVIDIA, "NVIDIA ConnectX-7 NDR 400G InfiniBand Adapter Card," 2021, https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf, accessed August 2022.

[44] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads," in *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 361–378.

[45] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir, "The benefits of general-purpose on-NIC memory," in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*, 2022, pp. 1130–1147.

[46] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 325–341.

[47] Ramulator, "DDR4 configuration parameters," 2018, https://github.com/CMU-SAFARI/ramulator/blob/master/src/DDR4.h, accessed August 2022.

[48] "RISC-V Cache-Block Management Instructions," https://github.com/riscv/riscv-CMOs/blob/master/cmobase/Zicbom.adoc#insns-cbo_clean, accessed August 2022.

[49] A. Sarma, H. Seyedroudbari, H. Gupta, U. Ramachandran, and A. Daglis, "NFSlicer: Data Movement Optimization for Shallow Network Functions," *CoRR*, vol. abs/2203.02585, 2022.

[50] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: a decade of clos topologies and centralized control in Google's datacenter network," *Commun. ACM*, vol. 59, no. 9, pp. 88–97, 2016.

[51] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat, "1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters," in *Proceedings of the ACM SIGCOMM 2020 Conference*, 2020, pp. 708–721.

[52] W. Su, L. Zhang, D. Tang, and X. Gao, "Using Direct Cache Access Combined with Integrated NIC Architecture to Accelerate Network Processing," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th*

International Conference on Embedded Software and Systems (HPCC-ICESSS), 2012, pp. 509–515.

[53] M. Sutherland, S. Gupta, B. Falsafi, V. J. Marathe, D. N. Pnevmatikatos, and A. Daglis, "The NeBuLa RPC-Optimized Architecture," in *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 199–212.

[54] D. Sánchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 475–486.

[55] The Verge, " Amazon is reportedly working on custom networking chips," 2021, https://www.theverge.com/circuitbreaker/2021/3/30/22358633/amazon-reportedly-custom-network-switch-silicon-aws, accessed August 2022.

[56] Tilera, "TILE-GX Instruction Set Architecture," 2013.

[57] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. J. Argyraki, S. Ratnasamy, and S. Shenker, "ResQ: Enabling SLOs in Network Function Virtualization," in *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 283–297.

[58] 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 *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 112–125.

[59] D. Zats, A. P. Iyer, G. Ananthanarayanan, R. Agarwal, R. H. Katz, I. Stoica, and A. Vahdat, "FastLane: making short flows shorter with agile drop notification," in *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, 2015, pp. 84–96.

[60] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient Datacenter Load Balancing in the Wild," in *Proceedings of the ACM SIGCOMM 2017 Conference*, 2017, pp. 253–266.