Understanding I/O Direct Cache Access Performance for End Host Networking

MINHU WANG, Tsinghua University, China
MINGWEI XU, Tsinghua University, China
JIANPING WU, Tsinghua University, China

Direct Cache Access (DCA) enables a network interface card (NIC) to load and store data directly on the processor cache, as conventional Direct Memory Access (DMA) is no longer suitable as the bridge between NIC and CPU in the era of 100 Gigabit Ethernet. As numerous I/O devices and cores compete for scarce cache resources, making the most of DCA for networking applications with varied objectives and constraints is a challenge, especially given the increasing complexity of modern cache hardware and I/O stacks. In this paper, we reverse engineer details of one commercial implementation of DCA, Intel's Data Direct I/O (DDIO), to explicate the importance of hardware-level investigation into DCA. Based on the learned knowledge of DCA and network I/O stacks, we (1) develop an analytical framework to predict the effectiveness of DCA (i.e., its hit rate) under certain hardware specifications, system configurations, and application properties; (2) measure penalties of the ineffective use of DCA (i.e., its miss penalty) to characterize its benefits; and (3) show that our reverse engineering, measurement, and model contribute to a deeper understanding of DCA, which in turn helps diagnose, optimize, and design end-host networking.

CCS Concepts: • **Hardware → Networking hardware**; • **Networks → Network adapters**; • **Mathematics of computing → Markov processes**.

Additional Key Words and Phrases: Direct Cache Access; End Host Networking;

ACM Reference Format:
Minhu Wang, Mingwei Xu, and Jianping Wu. 2022. Understanding I/O Direct Cache Access Performance for End Host Networking. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 22 (March 2022), 37 pages. https://doi.org/10.1145/3508042

1 INTRODUCTION

Data-intensive workloads, such as software network functions [75], training deep learning models [83], and microservices [62, 66] require end-host networking with hundreds-of-gigabits-scale bandwidth and microseconds-level latency. This has in turn encouraged the quick progress of network hardware in recent years. Since the 2000s, the speed of mainstream network interface card (NIC) in data centers has increased from 1 Gbps to 100 Gbps [67]. Moreover, the rise of domain-specific hardware accelerators [13], such as FPGA [12, 22], TPU [42], and GPU [26], requires more frequent interactions between the processor and peripheral devices. The quickly increasing I/O speed and traffic add to the pressure on other components in the server, especially main memory,

Authors' addresses: Minhu Wang, minhuw@acm.org, Tsinghua University, Institute for Network Sciences and Cyberspace, BNRist, Beijing, China, 100084; Mingwei Xu, xumw@tsinghua.edu.cn, Tsinghua University, Institute for Network Sciences and Cyberspace, Department of Computer Science and Technology, BNRist, Beijing, China, 100084; Jianping Wu, wjp@tsinghua.edu.cn, Tsinghua University, Institute for Network Sciences and Cyberspace, Department of Computer Science and Technology, BNRist, Beijing, China, 100084.

2476-1249/2022/3-ART22 $15.00
https://doi.org/10.1145/3508042

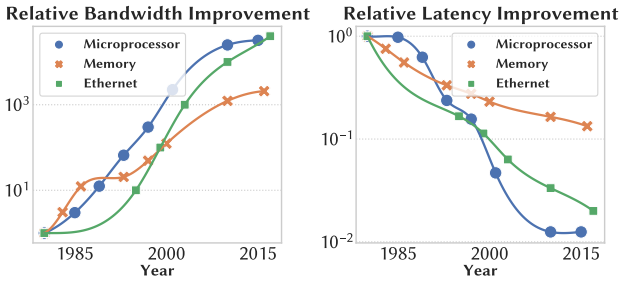


Fig. 1. Improvements (log scale) in the relative bandwidth and latency from the 1980s to the present [28].

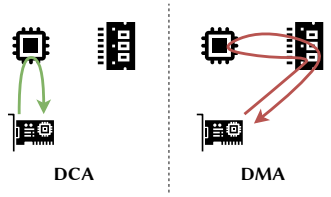


Fig. 2. A comparison of the network data paths in systems supporting DMA and DCA.

as shown in Fig. 1. Limited by its physical properties and DRAM market's preference for low cost and large capacity [10], the speed of DRAM falls behind that of both processors and NICs. Memory turns into a bottleneck in I/O operations in which it serves as the primary bridge between the processor and I/O devices. I/O devices hit the same *memory wall* [80] as processors if the performance divergence between NIC and DRAM grows continually.

The fast on-chip processor cache is the key to push beyond the *memory wall*. Direct Cache Access (DCA) extends Direct Memory Access (DMA) to enable I/O devices to also manipulate data directly in the fast on-chip processor cache, as shown in Fig. 2. DCA has been discussed in academic research [29, 49, 71] and implemented by vendors in widely used commercial hardware [31].

DCA is critical to scaling-up end-host networking, but it is challenging for applications to utilize it properly first owing to limited knowledge of DCA. Because the cache is transparent to software, cache implementations in commercial hardware, including DCA, are usually not fully documented. Each processor employs a unique combination of cache structures, an update policy, a placement/replacement policy, an inclusion policy, and a cache coherence protocol. Insufficient understanding of DCA may lead to inexact optimization guidelines (§4.5) or inaccurate performance diagnosis (§6.1). We disclose many details of DDIO, which is Intel's implementation of DCA, and use it as an example to emphasize the importance of a hardware-level understanding of DCA (Task 1, abbreviated as **T1**).

Writing software that makes the most of DCA, i.e., by completing all I/O operations in the cache such that it completely bypasses memory, is challenging. DCAed data that is not processed in time may be evicted back to memory due to cache collisions with subsequent DCAed data (*leaky DMA* problem [75]) or with data from neighbor applications (*latent contender* problem [82]). Evicted data will have to be brought back from memory, unfortunately making DCA useless. Since network I/O requires cooperation among the NIC, PCIe, driver, and application, the effectiveness of DCA operations (i.e., the hit rate) is affected by a wide range of factors in the I/O stack, including hardware specifications, system configurations, and workload characterization.

Moreover, networking applications have different performance-related objectives, and are under different resource constraints. DCA reduces the latency of I/O operations and memory traffic at the cost of a larger cache footprint, which may not be the goal of all applications. The available cache size and memory bandwidth also vary according to the execution environment. Current research gives many practical suggestions on how to tune DCA to meet the workload-specific requirements: *ResQ* [75] reduces the number of available network buffers to restrict the cache usage of a single Network Function (NF), thus ensuring performance isolation between multiple NFs sharing a physical machine. *Shenango* [59] chooses to copy packet once instead of applying

a zero-copy design to encourage buffer recycling. *PacketMill* [20] reserves most of the cache for DCA for extremely high performance packet forwarding. These empirical suggestions are specific to their contexts (e.g., multi-tenancy, dedicated machine) and targets (e.g., minimum interference, maximum performance) and thus challenging to generalize to arbitrary DCA optimization problems (e.g., generate less than 1 GB/s memory traffic while use no more than 20% of cache, §6.2).

To help network-intensive applications in a broad spectrum to take full advantage of DCA, we build an analytical framework to model the impacts of hardware specifications, system configurations, and workload characterization on the effectiveness of DCA, i.e., to answer the question: *how does the cache hit rate of DCA operations change if we tune X?* (**T2**) We also systematically measure the benefit of optimizing DCA, i.e., to answer the question: *what is the penalty of a miss by a DCA operation?* (**T3**) An application can determine its tuning target of DCA hit rate using the penalty measurement (**T3**), and can then learn tuning advice from the DCA model (**T2**) to utilize DCA according to its need.

We summarize the contributions of our paper as follows:

- We reverse engineer details of one commercial implementation of DCA, Intel's DDIO, especially how it is integrated with the cache coherence protocol in a non-inclusive cache hierarchy via a series of crafted microbenchmarks. We clarify several widespread misunderstandings of DDIO and identify unpublished details of its implementation (§3, **T1**).
- We develop an analytical framework to numerically predict the expected hit rate of DCA operations by considering cache specifications, system configurations, and workload characterization. We validate our model using synthesized microbenchmarks and realistic benchmarks (§4, **T2**).
- We systematically measure the latency of cache access, memory traffic, and the system's energy consumption to characterize the penalties of a DCA miss (§5, **T3**).
- We demonstrate how our knowledge and model may help diagnose performance of end-host networking, tune software configurations, and inspire hardware design (§6).

2 BACKGROUND

2.1 Ring-based CPU-NIC Interaction

NIC exchanges control and data messages with the processor in multiple ways, including interrupts, memory-mapped IO (MMIO), and Direct Memory Access (DMA). DMA is the primary way to transfer data. At the initialization of the networking subsystem, the processor allocates buffers in memory for incoming and outgoing packets. It records buffers' properties (e.g., their base addresses and lengths) in *descriptors* and stores these descriptors in two rings, one for transmitting (TX) and the other for receiving (RX). The processor and NIC maintain a head and a tail pointer, respectively, in each ring to implicitly synchronize their usages of descriptors.

As shown in figure Fig. 3 (**Sx** in this paragraph corresponds to the step **x** drawn in the figure), the NIC and processor cooperate closely in packet forwarding. **S1** At NIC initialization, the processor notifies it the configurations of the TX and RX rings using MMIO. **S2** When a packet arrives, NIC reads the descriptor pointed to by the ring head for an empty buffer, writes the packet content to it, and updates the descriptor to record properties of the packet. It moves the head pointer on and **S3** may also send an interrupt to the processor to notify of the arrival of a packet. **S4** The processor is notified by the interrupt or by busy polling on the descriptor pointed to by the tail pointer. It provides a new descriptor pointing to an empty buffer for the RX ring, moves the tail pointer on, and hands over the received buffer to the application. **S5** Once the handler routine has been completed, the driver registers the buffer on the empty TX descriptor pointed to by the head pointer and moves the pointer on to notify the NIC. **S6** NIC reads the TX descriptor, fetches the

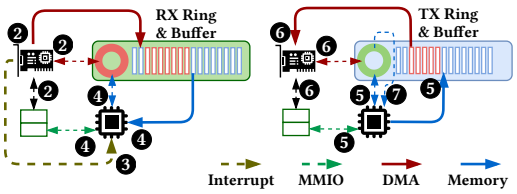


Fig. 3. How NIC and processor interact during packets forwarding.

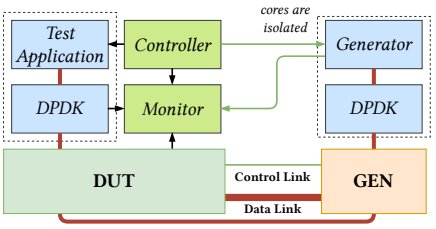


Fig. 4. Hardware connections and experimental framework of our testbed.

buffer, and sends its content to the wire. It moves the TX tail on to indicate the completion of packet sending. **S7** The driver periodically checks the TX ring to recycle buffers that have been sent.

The ring model is widely applied by NIC vendors because it allows the NIC and processor to communicate efficiently and safely without explicit synchronization. As reported in [61], most (40 of 44) drivers apply this model in the well-known DPDK [24] kernel-bypass I/O framework.

2.2 Cache in Modern Processor

Cache hardware in modern processors is complicated because it employs many techniques to recognize and utilize locality in the memory access stream, and targets a balance between performance (e.g., access latency, equivalent bandwidth, cache size) and cost (e.g., die area, energy consumption).

An emerging trend in processor design is the use of the exclusive cache hierarchy [7, 86]. Conventionally, each cache level contains a copy of all data in the higher levels. For example, L3 has a potential stale copy of any cache line in L2. The exclusive cache hierarchy instead ensures that each cache line is present in only one level at a time, thus increasing the effective cache size at the cost of management complexity.

Most processor vendors, including AMD (Athlon and Zen), Intel (Xeon), ARM (Cortex-A), and IBM (POWER), have used the exclusive cache hierarchy in their products [7]. For example, before the *skylake* microarchitecture, Intel's LLC is inclusive. However, it applies a non-inclusive LLC inclusive directory (NCID) approach [84] and designs an almost exclusive LLC for *skylake* [70]. An extra component called the Snoop Filter (i.e., the inclusive directory) is added to track states of the cache lines present only in L2. The non-inclusive hierarchy complicates the cache coherence protocol and the DCA implementation because LLC no longer tracks the states of all cache lines in the hierarchy.

2.3 DMA, DCA, and DDIO

Direct Memory Access (DMA) allows I/O devices to access memory without the involvement of the processor. The processor may switch to other tasks after issuing I/O operations, and is notified when I/O is completed. DMA frees the processor from the heavy memory copy in I/O operations, thus increasing processor utilization when dealing with high-speed I/O devices.

Direct Cache Access (DCA) improves DMA by changing its target from the memory to the processor cache. This helps reduce I/O latency and save memory bandwidth by preventing data from being transferred twice over the memory bus (shown in Fig. 2). DCA is a general terminology with varied implementations, such as the prefetch hint [29], cache injection [49], and a dedicated DMA cache [71]. Both DMA and DCA need to handle cache coherence issues. For DMA, data in the processor cache need to be invalidated on reception and be written back to memory before

being transmitted. For DCA, the cache coherence protocol must be tuned and needs to consider requests from I/O devices.

Data Direct I/O (DDIO) is Intel's implementation of DCA on its Xeon processor products. We focus on Intel in this paper because it was, to our best knowledge, the only vendor that provided commercially available processors with DCA support when we started this work. However, other vendors are also actively implementing DCA. For example, ARM has released its version of DCA, Cache Stashing, as part of its DynamIQ technology [5]. Cache Stashing is likely to be supported in upcoming AWS Graviton2 server processors [6].

DDIO is enabled by default, and is transparent to hardware, firmware, and drivers of IO devices. Intel reveals only limited details on how DDIO works, but reverse engineering of it is ongoing [21, 48, 72, 81, 82]. From official documents [31] and reverse engineering work, we have learned rich details of DDIO, which are, however, discussed mainly in the context of inclusive LLC. If target cache lines are present in LLC, both DDIO read and write operations are completed in the cache. Otherwise, PCIe write misses allocate cache lines in **two fixed ways** in LLC while PCIe read misses reads data from memory without cache line allocation. The allocation of DDIO is restricted to prevent it from polluting the entire cache. In the optimal case, all I/O operations are accomplished without generating any memory traffic.

In this paper, we use the term DCA to refer to the general idea of allowing I/O devices to read and write data directly from/to the processor cache, and DDIO to refer to the specific DCA implementation provided in Intel's processors. We use {PCIe, DDIO, DCA} read (write, request) interchangeably to refer to memory read (write, request) issued by I/O devices, and use processor read (write, request) to refer to memory read (write, request) issued by processor cores.

3 REVERSE ENGINEERING CACHE COHERENCE FOR DDIO

3.1 Motivation

While the placement and replacement policy of DDIO is clear, how DDIO is integrated into the cache coherence protocol, especially in a non-inclusive hierarchy, is not known. Conventionally, cache coherence is not discussed in detail in the context of cache modeling for two reasons. First, cache coherence influences only cache lines accessed concurrently by multiple cores with at least one writer. Second, LLC is the primary concern in cache models for SMP processors with an inclusive hierarchy [9] because the available cache size for an application is equivalent to its occupancy of LLC, while cache coherence has no effect on cache line insertion and eviction in LLC.

However, every DCA access involves multiple cache agents (i.e., the I/O device and the core) and is directly affected by the cache coherence protocol. As a result, the coherence protocol decides both the available cache size (§3.4) and the access latency (§5) of DDIO requests in both inclusive and non-inclusive hierarchy. Though the cache coherence protocol is essential for DDIO characterization, we do not know much about it. Specifically, we need to learn the coherence transactions that are issued for different types of DCA requests (§3.3) and how the cache line state changes on transaction handling (§3.4). [1] Although this paper focuses on high-speed NIC, the knowledge gained here also benefits the development on other peripheral devices, such as FPGA and GPU.

[1]We restrict the scope of this paper to the local device case and run all experiments on cores in the socket to which the NIC is attached. The use of remote I/O devices is generally not recommended [15, 17, 32] because the bandwidth of the interconnecting bus (QPI [87] or UPI [70]) is limited, and cross-NUMA cache coherence is more complex such that it incurs a more significant overhead [27, 68]; both issues lead to non-optimal I/O performance. Furthermore, *IOctopus* [65] has shown that vendors in the future may provide a dedicated PCIe physical function for each NUMA node to make one I/O device local to every NUMA node.

3.2 Methodology

We study how NIC uses DDIO in both Intel's *broadwell* [47] (with a inclusive cache hierarchy) and *skylake* [70] (with a non-inclusive cache hierarchy) microarchitectures by using the framework shown in Fig. 4. For each microarchitecture, two machines with identical hardware are connected back to back, in which one serves as the device under test (DUT) and the other serves as the packet generator (GEN). DUT and GEN are connected with two physical links: a 1 Gbps control link for control messages and a 40 Gbps (*broadwell*) / 100 Gbps (*skylake*) data link for test traffic. The detailed hardware specifications are given in §A.1. In each experiment, the controller configures properties of the generated traffic, executes the test application, waits for a configurable duration to stabilize the entire system, and then queries and saves metrics from the monitor. We use the same testbed and framework across this paper but apply varying workloads for different purposes. When multiple cores are utilized, traffic is distributed on them using receive side scaling (RSS). We also utilize Cache Allocation Technique (CAT, refer to §A.4 for more details) to isolate LLC between different applications.

To investigate the cache coherence protocol for DDIO requests, we develop a small benchmark called *packet-acrobat* (pseudocode given in §A.2) which mimics common packet access patterns in end-host networking. *packet-acrobat* works in three modes: `receive`, `send`, and `forward`. As the names indicate, it receives and then frees packets in the `receive` mode, generates and sends out packets in the `send` mode, and receives and then sends out packets in the `forward` mode. *packet-acrobat* may read, write, or flush some cache lines in the incoming or outgoing packets as the corresponding parameter `{write, read, flush}_ratio` specifies. For example, when *packet-acrobat* runs in the `forward` mode and is given parameter `write_ratio=0.4`, it keeps receiving packets, writing random bytes to the first 40% of cache lines in each received packet, and sending them out.

We infer cache behavior by tuning the parameters of *packet-acrobat* and measuring two types of hardware metrics: (1) the hit rate of PCIe requests and (2) the distribution of cache line states when touched. Specific counters and events for all metrics used in this paper are listed in §A.3. We will discuss how the parameters are set and the metrics are used in detail for each experiment.

3.3 Cache Transactions for DDIO

To link DDIO requests with coherence transactions, we run *packet-acrobat* many times using different TX/RX ring sizes, packet sizes, traffic rates, and acrobat parameters. We collect 336 samples in total for each microarchitecture. We calculate the numbers of expected DDIO requests and measure the numbers of all cache coherence transactions in each sample. To explore their relationships, we run a linear regression for each type of memory requests by setting its number as the dependent variable and setting the numbers of coherence transactions as independent variables. For example, we find that the number of partial cache line write is roundly equal to the number of Request for Ownership (`RFO`) transactions, and conclude that the I/O controller issues a `RFO` for each partial cache line write. We omit the results of regression for brevity and summarize only the conclusion here.

In both *broadwell* and *skylake*, for a full cache line write, I/O devices first issue an Invalid to Modified (`ItoM`) transaction to retrieve the ownership of the target cache line, and then issue a Writeback Modified to Invalid (`WbMtoI`) transaction to write the updated cache line back to LLC. They apply `RFO`, instead of `ItoM`, for partial cache line write and issue a Read Current (`PCIRdCur`) transaction for both full and partial cache line read. The conclusion matches what has been revealed in an open-source processor monitor project [58] and discussions on Intel's online forum [76].

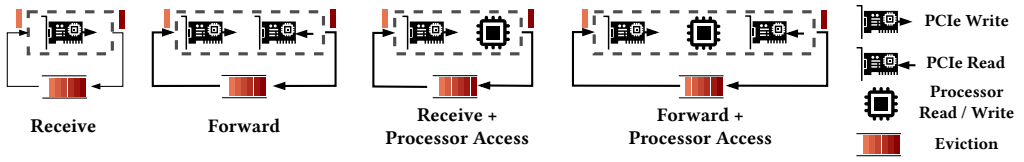


Fig. 5. Memory access sequences on cache lines for different workloads.

It's noticeable that processor cores issue different coherence transactions for the same type of memory requests: cores issue `RFO` for both full and partial cache line write, and seldom give up cache line ownership voluntarily by using `WbMtoI`. Instead of `PCIRdCur`, cores issue `DRd` for data read and `CRd` for code read. Both allocate a new cache line on a cache miss. It implies that the cache subsystem may differentiate between requests from I/O devices and those from cores, which is validated in §5.

3.4 Cache Line State Transition for DDIO Transactions

To speculate on the transitions of cache line states due to coherence requests, we develop a tool called the **buffer annealing** experiment. It utilizes the observation that the driver always keeps the RX ring full by replenishing it with a recently freed buffer. Because DDIO allocates cache lines for write misses, a buffer just written by the NIC must reside in the cache and is also very likely to stay there when it is inserted back into the RX ring if it is processed sufficiently quickly. As the NIC continually fetches empty buffers from the RX ring, the buffer will finally be reused but it may have been evicted to memory during the wait. As each buffer is alternatively moved to the cache by a sequence of memory accesses during processing (the buffer is heated up), and is evicted back to memory during the wait (the buffer cools down slowly), we call this buffer annealing.

As shown in Fig. 5, we may infer the cache coherence protocol by applying different memory access sequences during packet processing (different ways to heap up) and observing changes in the PCIe write hit rate and the captured coherence requests (different properties after cooling down). For example, if we find that the PCIe write hit rate is higher under sequences [PCIe write → processor write] than sequences [PCIe write], we know that processor write moves a cache line to L2 cache because L2 cache (1 MiB × 10 cores) is much larger than the DDIO portion in LLC (3.5 MiB) in our testbed. We may also learn cache behavior by tuning the available cache size and the RX ring size, which implicitly controls the scale of living network buffers. For example, we can reduce the available LLC by using CAT; if the PCIe write hit rate decreases, we conclude that cache lines are moved to LLC somehow.

For all experiments in this section, we utilize 10 cores to forward traffic at 10 Gbps in total so that all RX and TX queues are close to empty under such a light load. We send packets with a constant size of 1472 bytes (23 cache lines) and at a uniform interval. Unless specified otherwise, we disable the hardware prefetcher following Intel's guide [33] to eliminate unexpected memory requests.

3.4.1 PCIe Write. For PCIe write, we run `packet_acrobat` in `receive` mode, and tune the processor write ratio to apply a mixture of sequence I [PCIe write] and sequence II [PCIe write → processor write].

In *skylake*, as shown in Fig. 6 (b) and (c), a cache line may stay in two states: *M* and *SF E*[2] when a PCIe write hits it. When the ring size is 64 and fits in the DDIO portion, the proportion of *SF E* is roughly equal to `write_ratio`, showing that processor writes move cache lines to L2. We also

[2] *M* means *Modified* and *SF E* means *Snoop Filter Exclusive*, which is specific to the non-inclusive hierarchy. *SF E* indicates that the cache line resides in a higher-level cache, and is tracked by the Snoop Filter. More details are given in §A.1.

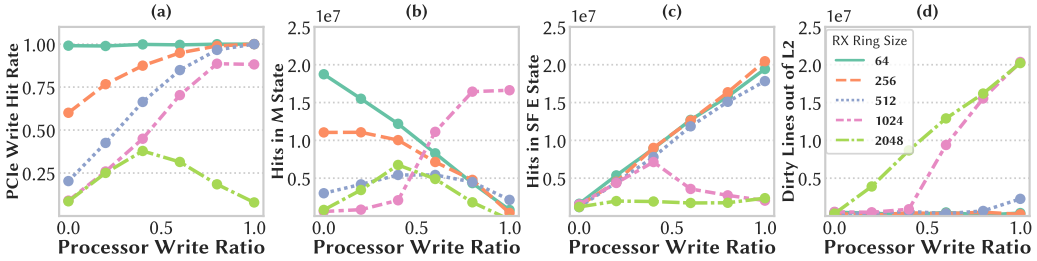


Fig. 6. How (a) the hit rate of PCIe write and (b-c) states of cache lines accessed by PCIe write hit change as the processor writes to more cache lines per packet in *skylake*. We show (d) the rate at which cache lines are evicted to LLC to help understand how their states change when the ring size is large.

find that processor writes subsequent to PCIe writes always reach cache lines in *M* state in LLC (not shown in the figure), which means that PCIe writes move cache lines in L2 back to LLC. It is consistent with the definition of the exclusive cache: cache lines are moved between different cache levels.

As the RX ring size increases, the size of the living network buffers exceeds the available DDIO space. Fig. 6 (a) shows that the PCIe write hit rate increases as `write_ratio` increases and more cache lines are moved to L2 by processor writes. If the ring size is increased further, an increasing number of cache lines are evicted from L2 to LLC as shown in Fig. 6 (d). The states of evicted cache lines are turned to *M* as shown in Fig. 6 (b). It is interesting that these cache lines are not restricted by the DDIO allocation rule. When the ring size is 2048, the PCIe write hit rate reaches its maximum when around half of the cache lines are moved to L2, in which case the buffers are dispersed between the two DDIO ways and some ways available for evicted cache lines from L2. We guess that the evicted cache lines are placed in the ways available to the application (the entire LLC when CAT is not applied), and validate it by repeating the experiment with a fixed ring size of 1024 and varied LLC ways. In this paper, we use a hex bitmask to record the CAT configuration. For example, given a total of 11 ways, `700` or `0x700` represents the 3 leftmost ways. As shown in Fig. 6 (e), reducing the number of LLC ways available to the application hurts the PCIe write hit rate only when `write_ratio` is high enough to cause evictions from L2, thus proving our hypothesis.

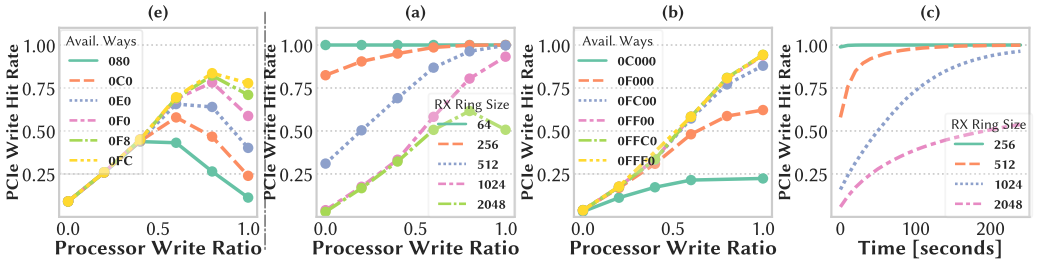


Fig. 6 (con'd). (e). Relationship between the PCIe write hit rate and the processor write ratio when CAT restricts the number of LLC ways available for the application in *skylake*; Fig. 7. (a-b). How the PCIe write hit rate changes with the processor write ratio under different buffer sizes and available cache sizes in *broadwell*. (c). How the PCIe write hit rate changes over time since the start of the application in *broadwell*.

In *broadwell*, all PCIe write hits meet M state cache lines in LLC because LLC is inclusive. But subsequent processor writes also reach LLC, showing that PCIe write invalidates copies at higher cache levels. However, it is surprising that increasing the processor write ratio also benefits the PCIe write hit rate as shown in Fig. 7. Because processor access helps store buffers much larger than the DDIO portion in LLC, we think that cache lines touched by the processor are moved to non-DDIO ways somehow. We restrict the LLC ways available to *packet-acrobat*, and observe a drop in the PCIe write hit rate as the number of available ways decreases, proving that cache lines are moved there.

Further investigation shows that in *broadwell*, the PCIe write hit rate increases gradually from when the application starts to when it becomes stable, as shown in Fig. 7. Such a phenomenon has never been observed in *skylake*. We suspect that cache lines written by NIC may have been evicted to memory before it is written by the processor, and are later put in non-DDIO ways (i.e. ways not available for DDIO) by processor writes and remain there. The tendency to migration holds until all cache lines touched by the processor have been moved or the pressures on non-DDIO and DDIO ways become equal.

We then repeat the buffer annealing experiment by using processor read instead of write, and obtain similar results. We also repeat the experiment with hardware prefetch enabled. As shown in Fig. 8 (a) and (b), the effect of hardware prefetch is similar with that of a higher write ratio because the prefetcher implicitly issues more requests from the processor.

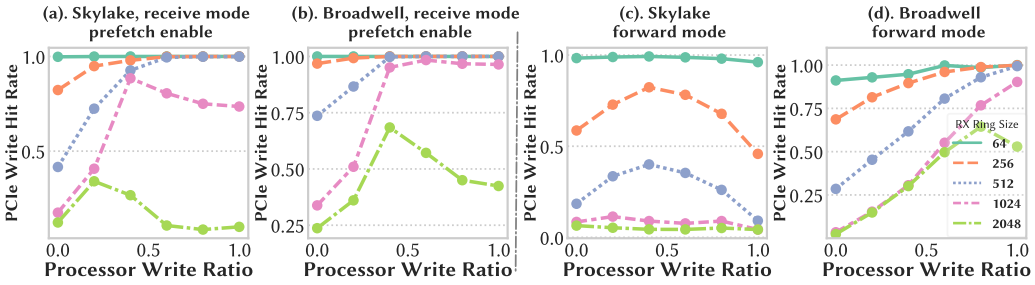


Fig. 8. (a-b). How the PCIe write hit rate changes as the processor touches more cache lines per packet when hardware prefetch is enabled (c-d). How the PCIe write hit rate changes as the processor touches more cache lines per packet when packets are forwarded rather than discarded.

3.4.2 PCIe Read. We repeat the buffer annealing experiment running *packet-acrobat* in the `send` mode and tune `write_ratio` to apply a mixture of sequence I [PCIe read] and sequence II [processor write → PCIe read]. In our testbed, TX rings are used eagerly: descriptors are updated on demand and freed soon after sending is completed. It is thus not feasible to control the scale of the living network buffers by tuning the TX ring size. But we still learn much about the behavior of PCIe read. Processor writes subsequent to PCIe reads always reach LLC in both *skylake* and *broadwell*, showing that PCIe read also invalidates *Modified* cache lines in L2. However, processor read never reaches LLC, indicating that PCIe read keeps *Shared* state copies in L2.

However, it is surprising that in *skylake*, both memory traffic and coherence transactions are observed when reading cache lines with `Shared` copies in L2. Such requests are also counted as cache hits by HPC. We think it's an optimization whereby the cache agent concurrently requests L2 and memory, and then returns the first response. A similar optimization has been reported in Intel's

inter-NUMA Source Snoop coherence implementation [30] but has not been reported in the intra-NUMA protocol. Recent discussions show that it is possibly related with a not well-documented feature called XPT Prefetcher [78].

To investigate how the positions of cache lines change on PCIe reads, we run *packet-acrobat* in the `forward` mode and tune `write_ratio` to apply a mixture of sequence I [PCIe write → PCIe read] and sequence II [PCIe write → processor write → PCIe read]. As shown in Fig. 8(b), there is no difference between the `receive` and `forward` modes on *broadwell*, which shows that in *broadwell* PCIe read does not change states of cache lines in LLC. However, for *skylake*, an extra PCIe read in the access sequence hurts the PCIe write hit rate as shown in Fig. 8(a). We suspect that PCIe reads bring cache lines that reside in L2 back to LLC in *skylake*. The similar pattern whereby the hit rate reaches its maximum when the processor write ratio is around 50% shows that PCIe reads also move cache lines to some non-DDIO ways. We limit ways available to *packet-acrobat* but surprisingly observe no change in the PCIe write hit rate, showing that there is another set of special ways in addition to the two DDIO ways.

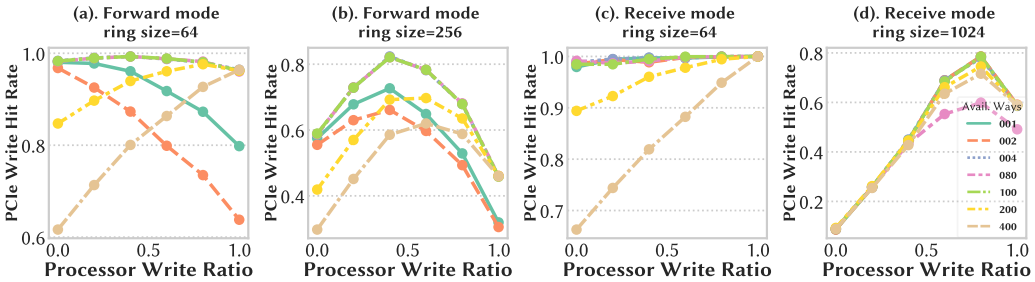


Fig. 9. How the PCIe write hit rate drop when a certain LLC way is shared with a distractor program exploiting LLC heavily under different ring sizes and packet processing modes.

To locate these latent ways, we artificially generate interference on each way[3] by using a distractor program (pseudocode given in §A.2) which heavily exploits LLC, and measure changes in the PCIe write hit rate. We always allocate the middle four ways (`0x0F0`) to *packet-acrobat*. As shown in Fig. 9, we locate two sets of special ways: interference on the two leftmost ways (`0x400` and `0x200`) reduces the PCIe write hit rate when `write_ratio` is low in both `forward` and `receive` modes. Interference on the two rightmost ways (`0x002` and `0x001`) takes effect only when `write_ratio` is high and in the `forward` mode. It implies that the two leftmost ways (`0x600`) are the two DDIO ways and the two rightmost ways (`0x003`) are prepared for cache lines present in both L2 and LLC. PCIe reads turn cache lines in L2 into *Shared* states and bring *Shared* copies to the two rightmost ways.

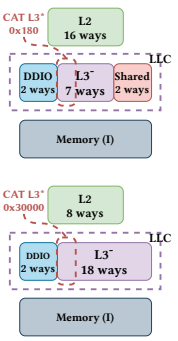
Past work [81] has also found that two ways are shared between L2 and LLC in the inclusive directory structure (i.e., the Snoop Filter), but did not reveal their functionality. We think that tags of cache lines present in both L2 and LLC are restricted to be stored in these two ways to maintain an invariant whereby there is only one entry in the directory for any given cache line. However, our conclusion is opposite to that reported in another study [21]. The authors there found that interference on the two rightmost ways increases the cache miss rate of an L2 forwarding application, and concluded that these are DDIO ways. Its observation is consistent with

[3]It has been reported that a hardware bug in CAT configures the way mask 0x400 as 0x600 by mistake, although it has not been officially acknowledged by Intel yet. However, it does not change the main conclusion here.

our conclusion: as shown in Fig. 9, all cache lines touched by the processor are brought from L2 to the two rightmost ways by PCIe reads. Contention on these two ways thus increases the miss rate of the application. The above-mentioned study also reported that cache misses of the distractor program increased when it was assigned the two leftmost ways (`0x600`), but could not explain why. It also matches our conclusion, as the two leftmost ways are just DDIO ways. Moreover, the model-specific register (MSR) `IIO_LLC_WAYS` [4] that controls the DDIO ways has the default value `0x600` on *skylake*, providing more evidence that DDIO uses the two leftmost ways.

Table 1. Transactions issued by DDIO operations and transition of cache line states in reaction to them. It records the next state for each pair of cache line state and coherence transaction. "-" denotes no change, "X" stands for dropping the cache line, and "+" means generating an additional copy. `DDIO` denotes LLC ways available for allocation on PCIe write (`0x600` on *skylake* and `0xC0000` on *broadwell*); `Shared` denotes ways for cache lines present in both L2 and LLC (`0x003` on *skylake*, and not applicable to *broadwell*). `L3-` stands for all remaining ways without special usage. Besides, `L3*` denotes ways available for applications if CAT is applied, and may overlap with `DDIO` or `Shared`.

Memory Operation	Cache Transaction	Cache State					
		DDIO (M)	L3-(M)	Shared (M)	L2 (EM)	L2 (S)	Memory (I)
Skylake (Non-Inclusive LLC)							
PCIe Write	ItoM + WbMtoI	-	-	-	DDIO(M)	DDIO(M)	DDIO(M)
	RFO + WbMtoI	-	-	-	DDIO(M)	DDIO(M)	DDIO(M)
PCIe Read	PCIRdCur	-	-	-	Shared(M)	L2(S)	-
Processor Write	RFO	L2(EM)	L2 (EM)	L2(EM)			L2(E)
Processor Read	DRd	L2(EM)	L2(EM)	L2(EM)			L2(S)
Eviction		Memory(I)	Memory(I)	Memory(I)	L3*(M)	-	
Broadwell (Inclusive LLC)							
PCIe Write	ItoM + WbMtoI	-	-		X	X	DDIO(M)
	RFO + WbMtoI	-	-		X	X	DDIO(M)
PCIe Read	PCIRdCur	-	-		X	X	Memory(I)
Processor Write	RFO	- + L2(EM)	- + L2(EM)				L3(E) + L2(M)
Processor Read	DRd	- + L2(EM)	- + L2(EM)				L3(E) + L2(E)
Eviction		Memory(I)	Memory(I)		L3(M)	X	



3.4.3 Summary. We summarize coherence transactions and transitions in the cache line states as DDIO request is handled in Tab. 1. The result of measurements is specific to only *broadwell* and *skylake* machines, and may be out of date as vendors improve their products. However, the methodology is general and buffer annealing experiments can be applied to other architectures as well. We postpone investigations on more architectures when their DCA implementations are available, such as Cache Stashing on ARM, to our future work.

3.5 Discussion

Lesson Learned #1: I/O May Use Much More Cache Than Was Previously Thought. Because DDIO writes allocate cache lines in only two ways, 10% (or 18.2% in *skylake*) of LLC is a widely used estimation of the available cache size for network buffers in the literature [11, 21]. However, applications or hardware prefetchers may bring cache lines elsewhere, explicitly or implicitly, and extend the available cache size for network buffers. For example, an IPSec gateway (that rewrites the entire packet) may use more cache for network buffers than an L3 Router (which updates only the IP header) by moving cache lines to L2 or non-DDIO ways as shown in Fig. 6 and Fig. 7. One should consider both application characterizations and processor microarchitectures to estimate the cache footprint precisely.

[4] Although the register is not recorded in the official MSR document [38], it was recorded by one motherboard manufacturer [69], has been discussed in Intel's online forum [77], and applied in the literature [20, 21].

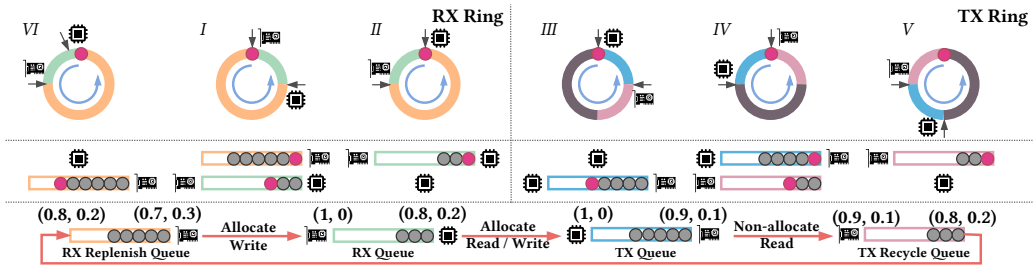


Fig. 10. The *ring model* and *queue model* for buffers' life cycle, where the descriptor and the buffer it points to are regarded as a whole. The arrow shows the descriptor being processed by the NIC and the processor in the *ring model*. Producers (left) and consumers (right) of all logical queues and sources (above) and destinations (below) of all queue switches are given for the *queue model*.

Lesson Learned #2: Microarchitecture Matters. Interestingly, progress in microarchitecture may unexpectedly influence seemingly irrelevant hardware features. As an example, a non-inclusive LLC renders the behavior of DDIO completely different. In *broadwell*, cache lines never move since insertion. However, in *skylake*, they move between L2 and various portions of LLC. We will discuss how this influences the DCA effectiveness (§4) and performance (§5) later in this paper, but it is already clear that understanding processors' microarchitectures is essential to using them well.

4 MODELING DCA HIT RATE

Although the cache hit rate of both DCA writes and reads can be measured precisely at runtime using HPC, an analytical model is still valuable in many ways. (1) It provides insight into how different factors in the I/O stack influence the cache hit rate, thus guiding optimization. (2) It also helps when the cost of measurement is unacceptable, such as when tuning parameters in a large search space, or when measurement is not available at all, such as when designing new processors.

We develop an analytical model to predict the hit rate of three types of DCA requests (abbreviated as the DCA hit rate): write and read from the NIC, and the first request from the processor to the DCAed data. We consider rich inputs from four categories: (1) hardware specifications, such as cache size and associativity; (2) system configurations, such as RX / TX ring size; (3) workload characterization, such as packet size; and (4) metrics revealing the runtime behaviors of the workload. All variables used in our model are summarized in App. B for quick reference. Our model assumes only that NIC adopts the RX/TX ring methodology described in §2.1, and is not limited to any specific DCA implementation. We select DDIO in *skylake* as a concrete example to illustrate our model, but a slightly simplified model for *broadwell* can be derived by using analogous steps.

4.1 Key Observations and the Basic Idea

We summarize observations regarding end-host networking that have inspired our analytical model.
Observation #1: Finite Packet Buffers. To eliminate dynamic memory allocation overhead, I/O frameworks typically allocate buffers in advance and create an object pool to manage them. All DCA requests reach these finite buffers. Whether a DCA request hits or not depends on whether the target buffer resides in the cache. We should thus study the properties of these finite network buffers to model the hit rate of possibly infinite DCA requests.
Observation #2: Cyclic & Homogeneous Buffer Life Cycle. As discussed in §2.1, NIC and CPU communicate via the TX and RX rings. We show the life cycle of a buffer from the perspective of these rings in Fig. 10. In the *ring model*, buffers are continually inserted into and removed from

these two rings. We may divide each ring into multiple consecutive slices segmented by head and tail pointers.

The *ring model* is conceptually equivalent to the *queue model* when each slice is regarded as a FIFO queue, and the movements of the head and tail pointers are considered to be the pops and pushes of buffers. For example, when NIC receives an incoming packet, it consumes a pending buffer and moves the head on to transfer the buffer's ownership from the NIC to the processor. It is equivalent to popping one buffer from a virtual *RX replenish* queue and pushing it into a virtual *RX queue*. In the *queue model*, the NIC and processor serve as producers and consumers of multiple logical FIFO queues. Buffers are moved in a circle of these queues, as illustrated in Fig. 10. The *queue model* provides an important view in which all buffers traverse these logical queues in the same order and wait in each queue for an equal time on average once the system reaches a steady state. As a result, buffers have a homogeneous life cycle and are identical in our model.

Observation #3: Logical and Physical States. As a buffer [5] traverses the processing pipeline, it may be brought into the cache by NIC and processor, and may be evicted back to memory due to cache collisions. We define the logical state of a buffer as its position in the logical queues and its physical state as its location in the cache hierarchy. An example of physical states and logical states in *skylake* is shown in Tab. 2. The transitions of physical and logical states are interrelated. For example, when NIC writes to a buffer, its logical state changes from *RX Replenish* to *RX*, and its physical state transits to `DDIO`, `Shared`, or `L3-`, depending on the previous state.

Table 2. The **physical** state of a buffer, describing its location on the cache hierarchy (see §3.4.3 for more details), and its **logical** state, describing its position in the circle of logical queues (see §2.1 and Fig. 10).

Physical State	Description
`L2`	only present in L2
`Shared`	shared ways in L3
`DDIO`	DDIO ways in L3
`L3-`	remaining ways in L3
`Memory`	only in memory

Logical State [Abbreviation]	Description
RX Replenish [RX/R]	empty; prepared for inbound packets
RX [RX]	store inbound packets waiting for processor processing
TX [RX]	store outbound packets waiting for NIC processing
TX Recycle [TX/R]	store packets having been sent and waiting for recycling
Idle	backup buffer for *RX Replenish Queue*

Observation #4: Queue Length-proportional Memory Access. The eviction probability of a buffer is determined by the memory access between two consecutive accesses to the buffer. As a typical networking application is built around a packet processing pipeline with a "receive → classify → process → send" loop, the amount of memory access of such a workload over a period is generally proportional to the number of packets received, processed, and sent in this period. Moreover, the properties of memory access are also stable so long as the pattern of network traffic remains unchanged. As a result, we may estimate the eviction probability by using the number of packets traversing the host during the wait given the traffic pattern.

Once a networking system has stabilized, its speed of receiving, transmitting, and processing packets must match; otherwise, the queues grow and the system becomes unstable. Suppose a buffer enters a *RX* queue with length L; the processor then has to first deal with the preceding L packets before handling it. Because all speeds match, the number of packets received and sent during the packet's wait are also equal to L. We may thus estimate the eviction probability of a buffer using the length of the queue in which it waits.

By combining the observations above, we build a Markov chain to model the transitions of the logical and physical states of network buffers during network processing, and estimate the expected DCA hit rate using the stable distributions of physical states at certain logical states.

[5]Typically, the terminology *buffer* refers to the entire memory space prepared for a packet that consist of multiple cache lines, but we abuse the terminology a little here to refer to only a single cache line in a buffer.

4.2 The Markov Model

To emphasize the core concept, we make several assumptions and discuss how they may be relaxed in App. D. Expressly, we assume that the load is uniformly distributed among multiple cores (§D.2), and that memory addresses of network buffers are uniformly and randomly mapped to the entire cache space (§D.5). The processor forwards all packets (§D.4) while the NIC treats all cache lines in the network buffers identically (§D.3). Further, `L3*` does not overlap with `Shared` or `DDIO`, because of which the destination of a cache line is always definite when its state changes (§D.1).

For a system with M distinct physical states and Q distinct logical states, we are concerned only with the distributions of physical states before and after changes in buffers' logical states, i.e., before and after buffers are accessed by the processor or NIC. We define the probability of a buffer staying in physical state m and logical state q as $p\left(m,q\right), m \in \{m_0, \cdots, m_{|M|}\}, q \in \{q_0, q_{0'}, \cdots, q_{|Q|}, q_{|Q|'}\}$, in which q represents the event buffer leaving queue q, and q' as the event buffer entering queue q. We further define the physical state distribution of buffers when their logical states are q as $\vec{p}_x = \left(p\left(m_0,x\right), \cdots, p\left(p_{|M|},x\right)\right), x \in \{q_0, q_{0'}, \cdots, q_{|Q|}, q_{|Q|'}\}$. The expected hit rate $\mathrm{E}\{h\}$ of one type of DCA requests is decided by the stable physical state distribution when buffers exit the source queue q of that type, i.e., $\mathrm{E}\{h\} = \sum_{m\in C} p(m,q) / \sum p(m,q)$. For example, the source queue of PCIe write is the *RX Replenish* queue, as shown in Fig. 10. Given $P_{x\to y}$ as the transition matrix of buffers' physical states when their logical states transition from x to y, the equilibrium equation is

$$\begin{bmatrix} \vec{p}_{q'_0} \\ \vec{p}_{q_0} \\ \cdots \\ \vec{p}_{q_{|L|'}} \\ \vec{p}_{q_{|L|}} \end{bmatrix} = \begin{bmatrix} 0 & \cdots & 0 & 0 & P_{|L|\to 0'} \\ P_{0'\to 0} & \cdots & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & P_{|L|-1\to |L|'} & 0 & 0 \\ 0 & \cdots & 0 & P_{|L|'\to |L|} & 0 \end{bmatrix} \begin{bmatrix} \vec{p}_{q'_0} \\ \vec{p}_{q_0} \\ \cdots \\ \vec{p}_{q_{|L|'}} \\ \vec{p}_{q_{|L|}} \end{bmatrix} \tag{1}$$

We divide all $P_{x\to y}$ into two types: an ***access matrix*** describing transitions on processor or NIC access, and an ***eviction matrix*** describing transitions when buffers wait in logical queues. The ***access matrix*** is determined by placement policies and cache coherence protocols, as summarized in Tab. 1. The ***eviction matrix*** is determined by probabilities cache lines are evicted during waits, which can be estimated using queue lengths, as stated in observation #4. We denote the probability that a cache line is evicted from cache region x when its logical state changes from q' to q as $p_q^{q'}\left(C_x(T)\right)$, in which T represents the sequence of memory accesses to region x during the wait, and $C(T)$ represents a set of statistics that determine the cache hit rate, and depends only on the length of q (i.e., L_q) given definite hardware specifications and workload characterizations. We discuss details of the ***eviction matrix*** in §4.3 but note that the ***eviction matrix*** depends on the physical state distribution because it determines how memory accesses are distributed in different cache regions. Two examples of transition matrices are shown in Fig. 11.

$$P_{0\to 1'} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_{\mathtt{L2}} \\ p_{\mathtt{DDIO}} \\ p_{\mathtt{L3\text{-}}} \\ p_{\mathtt{Shared}} \\ p_{\mathtt{Memory}} \end{bmatrix} \qquad P_{2\to 2'} = \begin{bmatrix} p^{\mathtt{L2}}_{\mathtt{L2}}(C_{\mathtt{L2}}(T)) & 0 & 0 & 0 & 0 \\ 0 & p^{\mathtt{DDIO}}_{\mathtt{DDIO}}(C_{\mathtt{DDIO}}(T)) & 0 & 0 & 0 \\ p^{\mathtt{L2}}_{\mathtt{L3\text{-}}}(C_{\mathtt{L2}}(T), C_{\mathtt{L3\text{-}}}(T)) & 0 & p^{\mathtt{L3\text{-}}}_{\mathtt{L3\text{-}}}(C_{\mathtt{L3\text{-}}}(T)) & 0 & 0 \\ 0 & 0 & 0 & p^{\mathtt{Shared}}_{\mathtt{Shared}}(C_{\mathtt{Shared}}(T)) & 0 \\ p^{\mathtt{L2}}_{\mathtt{Memory}}(C_{\mathtt{L2}}(T), C_{\mathtt{L3\text{-}}}(T),) & p^{\mathtt{DDIO}}_{\mathtt{Memory}}(C_{\mathtt{DDIO}}(T)) & p^{\mathtt{L3\text{-}}}_{\mathtt{Memory}}(C_{\mathtt{L3\text{-}}}(T)) & p^{\mathtt{Shared}}_{\mathtt{Memory}}(C_{\mathtt{Shared}}(T)) & 1 \end{bmatrix} \begin{bmatrix} p_{\mathtt{L2}} \\ p_{\mathtt{DDIO}} \\ p_{\mathtt{L3\text{-}}} \\ p_{\mathtt{Shared}} \\ p_{\mathtt{Memory}} \end{bmatrix}$$

(a) Transition on NIC write (b) Transition during the wait in the TX queue.

Fig. 11. Two examples of transition matrices of physical states. Note that each column adds up to be 1.

Once all transition matrices are ready, we can obtain the stable probability by any a mature implementation of numeric eigenvectors algorithms in common matrix tools [50, 56]. Because the stable distribution determines the equilibrium equation, we apply an iterative method to solve

it. Specifically, we set the initial distribution to $\vec{p}_x^0$ as $\{0, 0, \cdots, 1\}$, which means that all memory access ends with a miss, obtain the initial equilibrium equation using $\vec{p}_x^0$, and solve it to obtain the updated stable distribution $\vec{p}_x^1$. We construct an updated equilibrium equation using $\vec{p}_x^1$, and repeat the procedure until all $\vec{p}_x$ converge.

4.3 Eviction Matrices

We first discuss how to select a proper $C_x(T)$, and then show how to get $p_y^x\,(C_z(T))$ for different cache replacement policies.

4.3.1 Stack Reuse Distance. The stack reuse distance $L(x)$ for address x is defined as the number of unique memory objects accessed between two consecutive accesses to x. For example, for a memory access sequence `A B C A C D D B`, the stack reuse distances of A, B, C, and D are 2, 3, 1, and 0, respectively. We refer to it as reuse distance for brevity. It is developed to measure locality in memory access stream, and has been widely used in different cache models [3].

We choose the stack reuse distance as $C(T)$. Suppose the average packet size is M, the number of cores is S, the ratio of cache lines touched by the processor is α, and the queue length is L. The reuse distances for buffers in different cache regions then are

$$
\begin{aligned}
C_{\texttt{DDIO}}(T) &= (p(\texttt{DDIO}, RX') - \alpha p(\texttt{DDIO}, RX))\, SML = \mathcal{K}_{\texttt{DDIO}} SML \\
C_{\texttt{Shared}}(T) &= (p(\texttt{Shared}, RX') - \alpha p(\texttt{Shared}, RX) + \alpha p(\texttt{L2}, TX))\, SML = \mathcal{K}_{\texttt{Shared}} SML \\
C_{\texttt{L3*}}(T) &= (p(\texttt{L3*}, RX') - \alpha p(\texttt{L3*}, RX) + \alpha p(\texttt{L3*}, TX))\, SML = \mathcal{K}_{\texttt{L3*}} SML \\
C_{\texttt{L2}}(T) &= \alpha ML + \mathcal{A}(L)
\end{aligned}
$$

$\mathcal{A}(L)$ is the number of expected unique memory requests issued by the processor to code and data other than to the network buffers while processing L packets. For a simple workload such as the L2 forwarding, $\mathcal{A}(L)$ can be calculated using only program semantics. Otherwise, it may be approximated using the absolute reuse distance at the cost of precision [19]. Absolute reuse distance counts just the number of memory access, i.e. without the unique constraint, so can be measured efficiently at runtime using HPC.

4.3.2 Replacement Policy. **Least recently used (LRU)** The eviction probability of LRU cache is conventionally studied using the stack reuse distance [51]. In an N-associative LRU cache with C sets, if a cache line x that has been just accessed still resides in the cache after another T unique cache lines have been accessed, the number of cache line(s) assigned to the same set as x must be smaller than the associativity N. If cache lines are randomly assigned to sets, the number of cache lines assigned to a given set satisfies the well-known binomial distribution(with CDF $F_{\mathcal{B}}$ and PDF $f_{\mathcal{B}}$ in Eq. 5, 6 and 7). Therefore, the expected cache hit rate for a cache line with an average reuse distance T is

$$
p_x^x(C(T_x); N, C) = F_{\mathcal{B}}\left(C(T_x), N-1, \frac{1}{C}\right) \tag{2}
$$

Bi-modal Insertion Policy (BIP) enhanced LRU Intel has developed a Bi-modal Insertion Policy (BIP [63]) to relieve the cache thrashing problem in a LRU cache, which shows that cyclical access on a buffer slightly larger than the cache never hits. Instead of regarding a newly inserted cache line as the most recently used (MRU, or the youngest) one, BIP sets it as the MRU with a small probability ε and as the least recently used (LRU, or the oldest) one with a probability $1 - \varepsilon$. As most cache lines inserted are the oldest and are evicted first if not accessed, a cache line may not be evicted even if more than C cache lines are allocated to the same set in a BIP cache. The expected

overall hit rate of sequential memory access on a array with size T is then given as

$$p(C(T);N,C) = F_{\mathcal{B}}\left(C(T), N-1, \frac{1}{C}\right) + \sum_{k=N}^{\infty}\left(\frac{N-1}{k+1}\right) f_{\mathcal{B}}\left(C(T), k, \frac{1}{C}\right) \tag{3}$$

The Hard Truth: BIP + RRIP + Set Dueling Beyond BIP, Intel applies Set Dueling [63], which dynamically switches between two cache replacement policies (an LRU variant, and another BIP enhanced LRU variant) according to the hit rates of two small subsets of the cache with fixed replacement policies. The uncertain replacement policy prevents us from modeling Intel's cache precisely. Moreover, Intel applies a variant of LRU called the QLRU (Q for quad) or Re-reference Interval Prediction (RRIP [40]) which approximates the LRU by using only two age bits per cache line. QLRU provides limited resistance to the cache thrashing problem because a cache line can be at only four states, and thus ages to the oldest state after four accesses. We approximate the eviction matrix using Eq. 2 as the lower bound and a truncated version of Eq. 3 as the upper bound. Specifically, we cut the upper bound of the sum in Eq. 3 from ∞ to 3.

4.3.3 Eviction in an Exclusive Hierarchy. An evicted cache lines from L3 must be present in memory until the next access. However, cache lines evicted from L2 may remain in L3, and we are concerned with the probability of such a case. Observation #4 shows that the number of memory access is generally proportional to the number of packets, because of which the distributions of L2 and L3 access over time are generally uniform so long as the traffic is stable. As a result, we may estimate the number of remaining L3 access when a cache line is evicted from L2 using the ratio of L2 access that have arrived. The probability that the cache receives t unique L2 requests before eviction satisfies the negative binomial distribution $\mathcal{NB}$. We therefore get

$$p^{\mathrm{L2}}_{\mathrm{L3\text{-}}}\left(C(T_{\mathrm{L3\text{-}}}), C(T_{\mathrm{L2}})\right) = \sum_{t}^{C(T_{\mathrm{L2}})} f_{\mathcal{NB}}\left(t; N_{\mathrm{L2}}, 1-\frac{1}{C}\right) p^{\mathrm{L3\text{-}}}_{\mathrm{L3\text{-}}}\left(\left(1-\frac{t}{C(T_{\mathrm{L2}})}\right) C\left(T_{\mathrm{L3\text{-}}}\right)\right) \tag{4}$$

4.4 Model Validation

We validate our model by answering two questions: (1). Does it provides an insight into predicting impacts of various factors on the effectiveness of DCA? (2). Does our model gives a numerically accurate estimation of the DCA hit rate? For the first question, we apply a hypothesis-validation methodology: we first discuss how system configurations and application characteristics may affect the DCA hit rate using our model, and then validate the hypothesis on a *packet-acrobat*-based microbenchmark. For the second question, we use our model to predict the DCA hit rate on several widely used network functions when processing realistic packet traces. We measure only the hit rates of PCIe write and read because the hit rate of requests on specific memory locations cannot be measured without expensive software instrumentation. We report both the upper (if BIP is applied) and lower (if LRU is applied) bounds of our estimations in the micro-benchmark and an average of them in the benchmark.

4.4.1 Microbenchmark. We run *packet-acrobat* to imitate a pure forwarding task with little computational overhead as it has been shown to be the most sensitive to DCA workload [21]. We utilize four cores, each equipped with one TX and one RX ring with 512 descriptors, to forward traffic consisting of packets with a fixed length (by default 1472 bytes), and a uniform sending interval. We set `write_ratio` to 0.2 to add some computation overhead to mimic network protocol processing in a packet forwarding task.

Network Traffic & Queuing Effect Due to the unique access pattern, i.e., cyclically accessing finite buffers, larger network traffic means only repeating the same memory access sequence more times, and has no effect on the DCA hit rate. However, traffic determines system loads, and

influences the *TX* and *RX* lengths, which in turn changes the reuse distance. As Kingman's formula [44] suggests, the higher the load is, and the more dispersed the arrival interval and processing duration are, the longer is the queue. As shown in Fig. 12, queues never build up until the traffic reaches the line rate, and both PCIe read and write hit rates also remain stable until queues arise.

A smaller packet size implies less memory access given the same number of packets, and therefore a shorter reuse distance given the same queue length. As shown in Fig. 12, a smaller packet size always gives a higher DCA write hit rate when queues are not built up. Under a heavy load, smaller packets impose a more significant pressure on the PCIe bus [54] and generates a longer TX queue. We forward small packets at the maximum available speed on our testbed, and show the result in Fig. 12. Although the length of the *TX* queue (L_{TX}) is long when the packet size is small, the reuse distance, which is roughly the product of L_{TX} and the packet size, is still short, resulting in a high DCA hit rate.

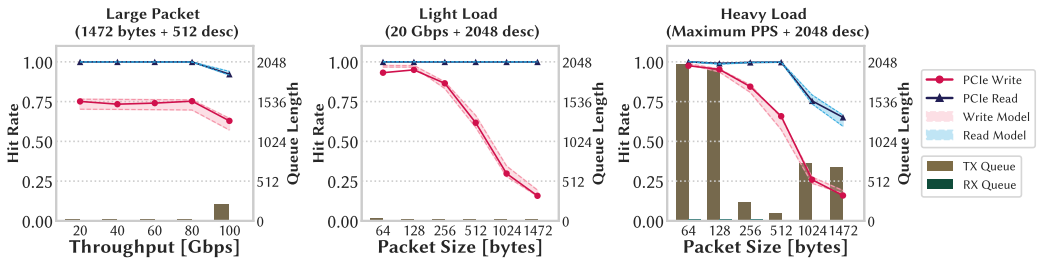


Fig. 12. Impact of the network traffic properties (throughput and packet size) on the effectiveness of DCA.

Ring Size Though not explicitly appearing in our model, the TX and RX ring sizes (Q_{TX} and Q_{RX}) influence queue lengths in many ways. First, the ring size may directly determine queue lengths depending on how RX and TX rings are used. For instance, the `mlx5` driver used in our testbed always fills the RX ring using empty RX descriptors, making the length of *RX Replenish* equal to the RX ring size minus the *RX* length ($L_{\text{RX/R}} = Q_{\text{RX}} - L_{\text{RX}}$). On the contrary, the TX descriptor is recycled soon after sending completes, and thus the length of *TX Recycle* is irrelevant to the TX ring size under a light load. As Fig. 13 shows, increasing Q_{RX} equivalently increases $L_{\text{RX/R}}$ under a light load, and reduces the PCIe write hit rate. But increasing Q_{TX} alone does not change the DCA hit rate (not shown in the figure).

Second, as queuing theory shows, a larger capacity reduces the drop rate at the cost of a longer queue under a high load. As shown in Fig. 13, a larger ring size leads to a longer *TX* queue under a heavy load, and reduces the PCIe read and write hit rates.

Third, the ring size serves as a hard upper bound for the lengths of *RX* and *TX* queues. Once the system is overloaded, the RX ring is always full of pending received packets, making L_{RX} equal to Q_{RX} and $L_{\text{RX/R}}$ equal to 0. We utilize only one core to overload the RX pipeline and show the result in Fig. 13. Interestingly, the zero *RX Replenish* queue results in close PCIe read and write hit rates, as the NIC writes to buffers almost immediately after reading them. It guides us to construct a counter-intuitive example in which the PCIe-side DCA metrics are desirable but the system performance is poor. We use two cores but change `write_ratio` from 0.2 to 1.0, so the system is still overloaded. As shown in Fig. 13, almost all PCIe reads and writes hit, but the system is overloaded, which shows that PCIe-side metrics cannot fully reveal the effectiveness of DCA.

Cores and Rings Sharing hardware TX or RX rings between cores requires expensive synchronization operations, and is usually not a suitable choice. We thus discuss the effects of cores and rings

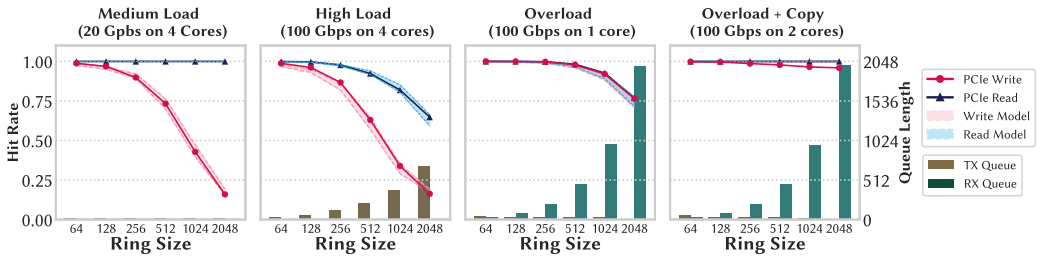


Fig. 13. Impact of the TX/RX ring size on the effectiveness of DCA.

concurrently. More cores and rings mean a longer reuse distance given the same queue length, but it also means more processing power on the processor side and, equivalently, less traffic on each core/ring pair such that the *RX* length may be smaller. On the NIC side, its scalability to the number of rings determines the *TX* length. We compare two scenarios: increasing the number of cores with a constant number of descriptors for each ring, or with a constant number of total descriptors, and show the results in Fig. 14. When the load is light and queue lengths are short, the effect of more cores is equivalent to that of a larger ring size. Under a heavy load, the effect of the number of cores is uncertain on the TX side as the product of the *TX* queue length and the number of cores is uncertain. Nevertheless, on the *RX* side, more cores and a shorter *RX* queue yield a longer *RX Replenish*, thus a worsened PCIe write hit rate.

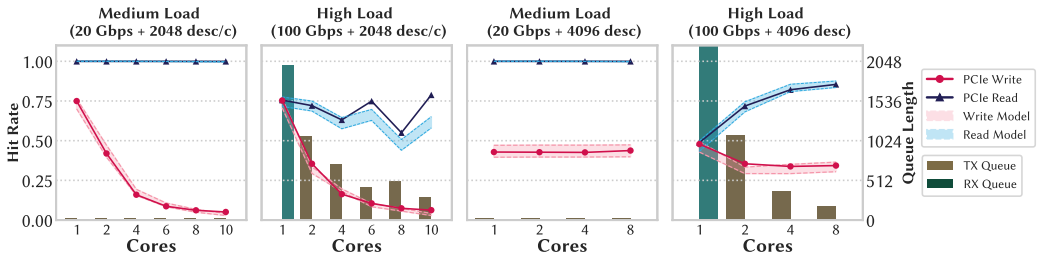


Fig. 14. Impact of the number of core/queue pairs on the effectiveness of DCA.

4.4.2 Benchmark. We validate the predictive accuracy of our model on MAWI [23] packet traces and four widely used network functions: L3 Router, ACL-based Firewall, Maglev-like [18] Load Balancer, and IPSec Gateway. The details of our traces and implementations of network functions are given in §A.2.3. Note that two assumptions, of identically accessed cache lines and uniformly distributed load, are no longer true due to the mixture of packets of varying sizes. We develop two extra models without these two assumptions in addition to the *Basic* model described in §4.2. For the *Per-Cache Line* model described in §D.3, we build separate *Basic* models for cache lines with different offsets in buffers and calculate the extended reuse distance which considers the possibility that a cache line is not touched in a loop. For the *Per-Core* model described in §D.2, we build a dedicated *Per-Cache Line* model for each core. We report the predictions and absolute errors of the three models on the Load Balancer, Firewall, and IPsec Gateway in Fig. 15. We also report the overall prediction errors and average convergence times of the three models. The results on L3 Router are not shown because they are almost identical to those of Load Balancer because they have the same memory access pattern.

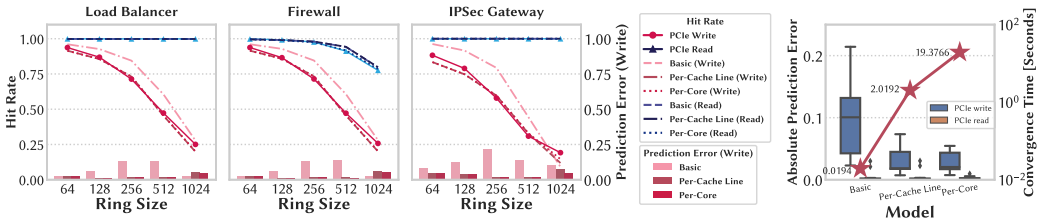


Fig. 15. Accuracy and convergence time of our models on realistic traces and varied network functions

As shown in Fig. 15, the *Per-Core* model generally gives more precise predictions but also takes more time to converge compared with two other models. The *Per-Core* model is only slightly better than the *Per-Cache Line* model as the imbalance is not very serious (the busiest core handles around 20% more packets than the idlest one). The best *Per-Core* model gives an average of 2.7% and a maximum of 5.4% prediction errors on the PCIe write hit rate, and less than 1% errors on the PCIe read hit rate, proving that our model can provide accurate predictions on the realistic workload. We find that the *Per-Cache Line* (and the *Per-Core*) model consistently gives less accurate results when the ring size is too small (64) or too large (1024). DPDK aggressively employs the batch policy to improve its throughput, which is currently not considered in our model. It makes our model give a less accurate queue length estimation when the ring size is close to the batch size (by default 32). We also suspect that the assumption that memory addresses are uniformly mapped to the whole cache space (§D.5) is not true, so a small part of cache lines always resides in the cache regardless of the reuse distance, making our model underestimating the hit rate when the ring size, and thus the reuse distance, is large.

4.5 Discussion

Lesson Learned #3: Take Care of Your Queue. Queue length is at the center of our model, and dominates the DCA hit rate. In the working set model, longer queues mean a larger working set and thus a lower hit rate. In our model, longer queues mean a higher eviction probability. Our model shows that queue lengths also determine the relative hit rates of the three types of requests in buffers' life cycle. For example, a long *RX* and a short *RX Replenish* lead to a high PCIe write hit rate and a low processor access hit rate, while a short *RX* and a long *RX Replenish* give the opposite.

Short queues are critical for reducing the cache footprint and taking full advantage of DCA. However, it is not always feasible due to hardware- or software-related issues. For example, PCIe contention may intermittently prevent NIC from fetching packets and results in a longer L_{TX} queue [54]. Workloads with high dispersion theoretically build up longer *RX* queues [62]. Even so, monitoring the changes in L_{TX} and L_{RX} is still critical for DCA performance diagnosis.

Lesson Learned #4: Be Cautious with Performance Metrics. PCIe-side metrics, i.e., the hit rate of PCIe read and PCIe write, are widely used to monitor the effectiveness of DCA [21]. However, as shown in Fig. 13, in cases involving large L_{RX} and small $L_{RX/R}$, the system exhibits high PCIe read and write hit rates but suffers from the *DMA leaky* problem [75]: DCAed buffers are evicted from the cache before being processed, making DCA writes useless. We need to consider the entire life cycle of buffers, rather than only PCIe-side metrics, to evaluate the effectiveness of DCA.

5 MEASURING DDIO MISS PENALTY

The cache miss penalty is commonly referred to as the extra latency introduced by a cache miss, and is used in the expression for the well-known average memory access time (AMAT [28]) formula:

$t_{\text{access}} = p_{\text{hit}} \times t_{\text{hit}} + p_{\text{miss}} \times t_{\text{miss}} = t_{\text{hit}} + p_{\text{miss}} \times t_{\text{penalty}}$. It is essential for the performance evaluation of cache systems. Besides, DDIO misses generate more memory traffic, consume valuable memory bandwidth, and increase the power of DRAM. We study all these penalties in this section.

We measure the latency of DDIO operations using the *uncore* HPC in the cache agent (CA). It measures the time from a coherence transaction (e.g., `RFO` or `ItoM`) entering the CA to leaving it. We cross-validate it with the latency of PCIe requests measured with HPCs in the integrated I/O controller (IIO). It gives the latency from a DMA request entering the IIO to leaving it, but is available only in *skylake*. [6] For example, a DDIO write miss generates three coherence transactions: an `ItoM`, a `WbMtoI`, and an `Eviction`, in which `Eviction` may not complete when the write succeeds as the store buffer in the processor helps move the actual memory write out of the critical path.

We tune the parameters of *packet-acrobat* to modify the states of the cache lines when they are accessed following Tab. 1. The results show that the latency of cache transactions and PCIe operations are highly correlated in Tab. 3. As expected, the latency of cache transactions is determined by cache line states in both microarchitectures. For example, a *Shared* copy could be invalidated directly, but a *Modified* copy has to be written back to LLC, making an `ItoM` access to the former faster than to the latter, but both are slower than access to a cache line without any copy in L2.

Table 3. The latency of cache transactions for cache lines in different states. We add the latency of PCIe write and read, measured in IIO, for cross-validation, but it is available only in *skylake*. [M] means that the latency is determined by the latency of memory access.

	skylake				*broadwell*			
	LLC	L2 S	L2 M	Memory	LLC	+L2 S	+L2 M	Memory
PCIRdCur	15.1 *ns*	189.0 *ns* (20 Gbps) 135.0 *ns* (100 Gbps)	110 *ns* (20 Gbps) 56.9 *ns* (100 Gbps)	94.1 *ns* [M]	14.2 *ns*	33.0 *ns*	162 *ns*	80.6 *ns* [M]
PCIe Read	171.6 *ns*	776.03 *ns*	223.34 *ns*	425.5 *ns*				
ItoM	14.8 *ns*	34.5 *ns*	44.9 *ns*	15.4 *ns*	13.1 *ns*	35.2 *ns*	41.7 *ns*	91.1 *ns*
RFO	16.7 *ns*	36 *ns*	41.2 *ns*	89.9 *ns* [M]	14.3 *ns*	33.2 *ns*	43.3 *ns*	93.4 *ns* [M]
WbMtoI	50.7 *ns*	49.5 *ns*	49.3 *ns*	50.2 *ns*	36.8 *ns*	36.8 *ns*	36.8 *ns*	37.1 *ns*
Eviction				51.4 *ns*				36.1 *ns*
PCIe Write	138.2 *ns*	145.47 *ns*	155.93 *ns*	146.0 *ns*				

The result also shows that full cache line write (`ItoM`) is different from partial cache line write (`RFO`) in that the latency of `RFO` depends on the latency of memory access while ItoM does not. The difference is due to the semantics of `RFO` and `ItoM`: `RFO` modifies only part of a cache line, and thus has to retrieve the unmodified part from memory. `ItoM` overwrites the entire cache line, leaving behind the stale value. It is also interesting that a PCIe write miss is not necessarily slower than a hit in *skylake*, and even faster if a *Modified* state copy is present in L2. This is reasonable as a write miss does not require an inter-core coherence transaction. Although it has to evict a victim cache line, the eviction is moved out of the critical path.

As for PCIe read (`PCIRdCur`), the results exceed our expectations, and are honestly odd. First, the read latency of cache lines present only in L2 decreases as the load increases. Second, PCIe read to cache lines in L2 is even slower than to those in memory when the load is light. We speculate that `PCIRdCur` has very low priority in the cache subsystem unless pending requests pile up and stall the pipeline, because of which its latency decreases as the load increases. But it cannot explain why reading a `Shared` copy in L2 is slower than reading one from memory, even when considering that requests are concurrently sent to L2 and the memory controller, as discussed in §3.

[6] The latency of PCIe requests may also be measured in a programmable NIC [54] using custom DMAs. But it captures the effects of all components in the I/O path, including the DMA engine, PCIe bus, and IOMMU.

Our measurement is consistent with the PCIe performance benchmark [54]. Both show that a partial cache line write miss is much slower than a hit, and so is a PCIe read. But a full cache line write miss is only slightly slower than a hit. However, the prior benchmark considered only whether the cache line resides in the cache, and ignored the impact of its coherence states.

We measure changes in memory traffic and DRAM power usage as the DCA miss rate increases, and show the results in Fig. 16. PCIe read misses generate memory read traffic as I/O devices have to retrieve data from memory, while PCIe write misses generate memory write traffic as the evicted cache lines have to be written back. As the ring size increases, power consumption increases from around 33 W to 50 W due to increasing memory traffic. Meanwhile, the power consumption of the entire package (including both *core* and *uncore*) remains stable. It validates that the ineffectiveness of DCA leads to the side effects of increased memory traffic and energy consumption. Optimizing DCA is thus still important even if the latency penalty is not always significant.

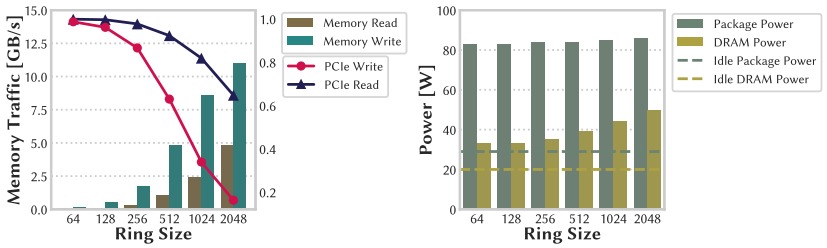


Fig. 16. Memory traffic and power increase due to more DCA misses.

Lesson Learned #5: Miss May not be a Disaster While Hit May not be a Pleasure. The increasing complexity of cache coherence protocol and the overhead of on-chip networks blurs the distinction between a cache hit and miss. First, the access latency of a cache miss may not be longer than that of a hit, as shown in Tab. 3. Second, requests that hit may also generate memory traffic, as discussed in §3.4.2. Blindly pursuing a high cache hit rate may not always benefit the end-to-end performance, and we have to *always measure one level deeper* [60].

6 CASE STUDY

6.1 Performance Diagnosis

Although DDIO is frequently referred to in performance diagnosis, it sometimes serves as a convenient scapegoat to explain abnormal results due to our limited knowledge of it. We analyze one of such cases to show that how a deeper understanding of DDIO helps in performance diagnosis.

Golestani et al. [25] studied the scalability of software data planes. They saturated a software data plane, changed the number of cores used in forwarding, and, surprisingly, found that small packet forwarding performance dropped suddenly when the number of cores(queues) increased from 7 to 8. They meanwhile observed increasing LLC access and decreasing memory traffic, and concluded that DDIO is enabled only when more than 8 queues are employed.

However, it is inconsistent with our knowledge as DDIO always takes effect, and is transparent to the driver or the application. As they always saturated the link and overloaded the software data plane, the RX ring was always full. Increasing the number of cores led to a longer reuse distance, and lower PCIe write and read hit rates, and thus led to more memory traffic. We suspect that the abnormal performance drop is not due to DDIO. We successfully reproduce their experiment, and find that the `mlx5` driver automatically enables an optional TX inline feature when the number of

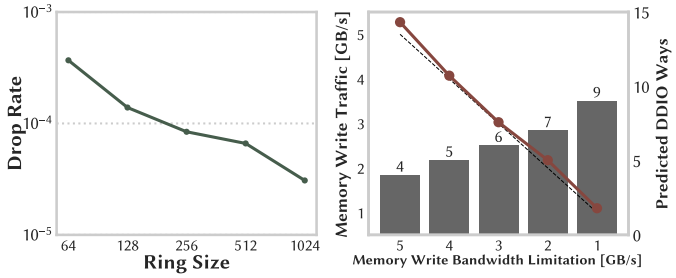


Fig. 17. The Impossible Trinity among the drop rate, memory bandwidth, and cache usage.

queues is at least 8 [16]. It copies the content instead of the address of a packet into the descriptor to help NIC fetch both in a single PCIe operation, which improves PCIe performance at the cost of more CPU cycles on data copy. Due to the inline copy, the processor brings evicted packets back before PCIe read. It reduces the amount of memory traffic due to the empty *RX replenish* queue, similar to that in Fig. 13. However, TX inline makes the forwarding task more computation intensive, and forwarding performance drops until enough cores are provided. However, it is impossible to reach the line rate for small packets without TX inline due to the limitations of PCIe. [54].

6.2 Software Tuning

Impossible Trinity in DCA Tuning Current implementations of DDIO make three objectives impossible to achieve together: (1). low memory traffic, (2). low cache usage, (3). resistance to fluctuations in workload. If one employs a short TX and RX ring, like *ResQ* [75], to limit its cache usage, it achieves (1) and (2) but fails to meet (3) because network burst or back pressure from the processor easily overflows the short queues, leading to packet loss [74]. Instead, one may employ a large TX and RX ring to minimize the loss rate at the cost of massive memory traffic induced by evicted buffers, unless allocating a larger portion of cache for DDIO as *PacketMill* [20] does. We show how our model may help make tuning choices in Fig. 17. We run the Load Balance benchmark on §4.4.2 using varied ring sizes. The results prove that the packet drop rate decreases as the ring size increases. When the ring size is set to 1024 to minimize the loss rate, we successfully use our model to predict the required DDIO ways to meet the given constraints of memory write throughput.

Our model currently assumes that queue lengths are irrelevant to the tuned parameter, which is not always true, as shown in §4.4.1. Parameters such as the ring size have a significant impact on queue lengths, and we thus need a queuing model which predicts the lengths of RX and TX queues to help our model cover more parameter tuning problems. We also need to integrate our DCA model into a general cache model [9] to support networking applications with intensive memory access. We leave these two problems as our future work.

Cache Management & LLC Allocation A popular cache management policy is to allocate LLC ways between different applications. It ensures performance isolation and usually yields better performance thanks to the mitigation of resource contention, especially in a multi-tenant environment. The existence of `Shared` ways has not been revealed yet and therefore is not considered in current allocation strategy [82]. As all cache lines touched by the processor are moved there, contention on `Shared` ways also violates performance isolation, and should be avoided.

6.3 Hardware Design

Unnecessary Eviction Under a light load, the *RX* and *TX* queues are short, and thus most eviction happens in the *RX Replenish* and *TX Recycle* queues. These evictions generate massive but unnecessary memory write traffic as the evicted data is never read again. Although in the software layer, the driver releases buffers after sending completes, and is indifferent to whether their value is saved. This message is not passed to the hardware layer. I/O data is different from program data and code as they are disposable. Now that they can be dropped once they have discharged their duties, further DCA design may mark cache lines used for I/O, and discard these released cache lines instead of writing them back to memory to reduce memory traffic.

7 RELATED WORK

Cache Reverse Engineering and Modeling Unpublished details of caches in commercial hardware are actively revealed for security- or performance-related issues [1, 2, 39, 52, 53, 81]. Researchers have developed two methodologies in reverse engineering: (1). measuring high-level hardware metrics on workloads with a strong pattern, and (2). measuring the access latency of precisely crafted memory access sequences. We use the former methodology in this study. Variants of the reuse distance are widely applied in analytical models for LRU-like caches [3, 14, 51, 55, 64, 79, 85]. Typically, distributions of reuse distances are utilized to describe memory access patterns in workloads, but the cyclic access pattern of network buffers simplifies our model.

DCA and DDIO DCA is proposed in the age of 10Gbps Ethernet to help the memory subsystem catch up with quickly growing network traffic [29, 45, 49, 71]. The benefits and limitations of early prefetch hint-based DCA have been discussed in detail [45, 46]. Since its introduction, DDIO has been frequently considered for the performance diagnosis of network-intensive applications [25, 59, 75]. Security researchers are also interested in side-channels exposed by DDIO [48, 72]. Alian et al. studied DDIO using an architecture simulator [4], and Farshin et al. characterized it empirically [21]. Neither of them conducts systematical microarchitecture-level measurements or builds a numerical model to understand DDIO analytically. Knowledge of DDIO has been shown to be helpful in the effective allocation of LLC in a multi-tenant environment [82], and in the extreme optimization of networking applications [20].

Endhost Networking Characterisation In order to solve the *killer microsecond* [8] problem, i.e., optimize the end-host networking to match the raw speed of network hardware, each component in the I/O path, such as the PCIe bus [54], network stack in the operating system [11], and threading model of networked applications [66], has to be measured and modeled to identity the bottleneck and improve performance. Our work contributes to a missing piece in the big picture.

8 CONCLUSION

I/O direct cache access provides optimization opportunities and design challenges for end-host networking. Making the most of DCA is tricky given the increasing complexity of the memory subsystem and I/O stacks in modern processors. This paper shows that architecture-level measurements and modeling of DCA is essential for first understanding and then utilizing it. We hope that tools and models developed in our work and the lessons we have learnt in our exploration can help end-host networking diagnosis, tuning, and design.

ACKNOWLEDGMENTS

We thank our shepherd Daniel S. Berger and anonymous SIGMETRICS reviewers for their valuable comments. The research is supported by the National Natural Science Foundation of China under Grant 61832013 and 61872426. Mingwei Xu is the corresponding author.

REFERENCES

[1] Andreas Abel and Jan Reineke. 2014. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE, Monterey, CA, USA, 141–142.

[2] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, Providence, RI, USA, 673–686.

[3] Anant Agarwal, Mark Horowitz, and John L. Hennessy. 1989. An Analytical Cache Model. *ACM Trans. Comput. Syst.* 7, 2 (1989), 184–215.

[4] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. 2020. Data Direct I/O Characterization for Future I/O System Exploration. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020*. IEEE, Boston, MA, USA, 160–169.

[5] ARM. 2021. Arm DynamIQ Shared Unit Technical Reference Manual. https://developer.arm.com/documentation/100453/0401 [Online; accessed 15-January-2022].

[6] AWS. 2021. Announcing AWS Graviton2-based instances for Amazon Neptune. https://web.archive.org/web/20211201211023/https://aws.amazon.com/about-aws/whats-new/2021/11/aws-graviton2-based-instances-amazon-neptune/ [Online; archived 1-December-2021; accessed 15-January-2022].

[7] Luna Backes and Daniel A. Jiménez. 2019. The impact of cache inclusion policies on cache management techniques. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*. ACM, Washington, DC, USA, 428–438.

[8] Luiz André Barroso, Mike Marty, David A. Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Communication of ACM* 60, 4 (2017), 48–54.

[9] Nathan Beckmann and Daniel Sánchez. 2016. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE, Barcelona, Spain, 225–236.

[10] Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5 (2011), 67–77.

[11] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*. ACM, Virtual Event, USA, 65–77.

[12] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE, Taipei, Taiwan, 7:1–7:13.

[13] William J. Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[14] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. ACM, San Diego, CA, USA, 245–257.

[15] DPDK. 2020. How to get best performance with NICs on Intel platforms. https://web.archive.org/web/20201031183151/https://doc.dpdk.org/guides/linux_gsg/nic_perf_intel_platform.html [Online; archived 3-Octeber-2020; accessed 5-September-2021].

[16] DPDK. 2021. 34. MLX5 poll mode driver — Data Plane Development Kit. https://web.archive.org/web/20210507064300/https://doc.dpdk.org/guides/nics/mlx5.html [Online; archived 3-May-2021; accessed 5-January-2022].

[17] DPDK. 2021. MLX5 poll mode driver — Data Plane Development Kit documentation. https://doc.dpdk.org/guides/nics/mlx5.html [Online; accessed 1-October-2021].

[18] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. USENIX, Santa Clara, CA, USA, 523–535.

[19] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*. IEEE, White Plains, NY, USA, 55–65.

[20] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostic. 2021. PacketMill: toward per-Core 100-Gbps networking. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, Virtual Event, USA, 1–17.

[21] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostic. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX, Virtual Event, 673–689.

[22] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. USENIX, Renton, WA, USA, 51–66.

[23] Romain Fontugne, Patrice Abry, Kensuke Fukuda, Darryl Veitch, Kenjiro Cho, Pierre Borgnat, and Herwig Wendt. 2017. Scaling in Internet Traffic: A 14 Year and 3 Day Longitudinal Study, With Multiscale Analyses and Random Projections. *IEEE/ACM Trans. Netw.* 25, 4 (2017), 2152–2165.

[24] Linux Foundation. 2015. Data Plane Development Kit (DPDK). http://www.dpdk.org

[25] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F. Wenisch. 2019. Software Data Planes: You Can't Always Spin to Win. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, Santa Cruz, CA, USA, 337–350.

[26] Samuel Greengard. 2016. GPUs reshape computing. *Commun. ACM* 59, 9 (2016), 14–16.

[27] David B. Gustavson. 1992. The Scalable Coherent Interface and related standards projects. *IEEE Micro* 12, 1 (1992), 10–22.

[28] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture - A Quantitative Approach, 6th Edition*. Morgan Kaufmann, Waltham, MA, USA.

[29] Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. 2005. Direct Cache Access for High Bandwidth Network I/O. In *32st International Symposium on Computer Architecture (ISCA 2005), 4-8 June 2005, Madison, Wisconsin, USA*. IEEE, Madison, Wisconsin, USA, 50–59.

[30] Intel. 2009. An Introduction to the Intel® QuickPath Interconnect. https://web.archive.org/web/20210409183234/https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html [Online; archived 18-April-2021; accessed-6-January-2022].

[31] Intel. 2012. Intel® Data Direct I/O Technology (Intel® DDIO): A Primer. https://web.archive.org/web/20210225132434/https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf [Online; archived 25-February-2021; accessed 6-September-2021].

[32] Intel. 2012. Process and Interrupt Affinity on Intel® Xeon® Processor E5 Servers with Intel® DDIO Technology. https://web.archive.org/web/20150419082907/http://www.intel.com/content/dam/www/public/us/en/documents/application-notes/xeon-e5-ddio-appl-notes.pdf [Online; archived 19-April-2015; accessed 6-September-2021].

[33] Intel. 2014. Disclosure of Hardware Prefetcher Control on Some Intel® Processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html [Online; accessed 19-September-2021].

[34] Intel. 2016. Intel® Xeon® Processor E5 and E7 v4 Product Families Uncore Performance Monitoring Reference Manual. https://www.intel.com/content/www/us/en/products/docs/processors/xeon/xeon-e5-e7-v4-uncore-performance-monitoring.html [Online; accessed 8-September-2021].

[35] Intel. 2016. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://web.archive.org/web/20210527071318/https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html [Online; archived 27-May-2021; accessed 17-September-2021].

[36] Intel. 2017. Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring. https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-uncore-performance-monitoring-manual.html [Online; accessed 8-September-2021].

[37] Intel. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-volume-3b-system-programming-guide-part-2.html [Online; accessed 8-September-2021].

[38] Intel. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-software-developers-manual-volume-4-model-specific-registers.html [Online; accessed 19-September-2021].

[39] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*. IEEE, Madeira, Portugal, 629–636.

[40] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA 2010),*

June 19-23, 2010, Saint-Malo, France. ACM, Saint-Malo, France, 60–71.

[41] Natalie D. Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. *On-Chip Networks*. Morgan & Claypool Publishers, Waltham, MA, USA.

[42] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, Toronto, ON, Canada, 1–12.

[43] S. Kent. 2005. *IP Encapsulating Security Payload (ESP)*. RFC 4303. RFC Editor.

[44] J. F. C. Kingman. 1961. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* 57, 4 (1961), 902–904.

[45] Amit Kumar and Ram Huggahalli. 2007. Impact of Cache Coherence Protocols on the Processing of Network Traffic. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*. IEEE, Chicago, IL, USA, 161–171.

[46] Amit Kumar, Ram Huggahalli, and Srihari Makineni. 2009. Characterization of Direct Cache Access on multi-core systems and 10GbE. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*. IEEE, Raleigh, NC, USA, 341–352.

[47] Mahesh K. Kumashikar, Shridhar G. Bendi, Srikanth Nimmagadda, Anup Jyoti Deka, and Anil Agarwal. 2017. 14nm Broadwell Xeon® processor family: Design methodologies and optimizations. In *IEEE Asian Solid-State Circuits Conference, A-SSCC 2017, Seoul, Korea (South), November 6-8, 2017*. IEEE, Seoul, Korea (South), 17–20.

[48] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. NetCAT: Practical Cache Attacks from the Network. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, San Francisco, CA, USA, 20–38.

[49] Edgar A. León, Kurt B. Ferreira, and Arthur B. Maccabe. 2007. Reducing the Impact of the MemoryWall for I/O Using Cache Injection. In *15th Annual IEEE Symposium on High-Performance Interconnects, HOTI 2007, Stanford, CA, USA, August 22-24, 2007*. IEEE, Stanford, CA, USA, 143–150.

[50] Mathworks. 2021. Subset of eigenvalues and eigenvectors - MATLAB eigs - MathWorks. https://ww2.mathworks.cn/help/matlab/ref/eigs.html [Online; accessed 29-September-2021].

[51] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.

[52] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, Vol. 9404. Springer, Kyoto, Japan, 48–65.

[53] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*. IEEE, Beijing, China, 739–748.

[54] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*. ACM, Budapest, Hungary, 327–341.

[55] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri E. Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE, Orlando, FL, USA, 37–48.

[56] numpy. 2021. numpy.linalg.eig — NumPy v1.21 Manual. https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html [Online; accessed 29-September-2021].

[57] University of Oregon. 2021. Route Views Archive Project. http://archive.routeviews.org/ [Online; accessed 24-December-2021].

[58] opcm. 2021. opcm/pcm: Processor Counter Monitor. https://github.com/opcm/pcm

[59] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, USA, February 26-28, 2019*. USENIX, Boston, MA, USA, 361–378.

[60] John K. Ousterhout. 2018. Always measure one level deeper. *Commun. ACM* 61, 7 (2018), 74–83.

[61] Solal Pirelli and George Candea. 2020. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX, Virtual Event, 225–241.

[62] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, Shanghai, China, 325–341.

[63] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. 2007. Adaptive insertion policies for high performance caching. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*. ACM, San Diego, California, USA, 381–391.

[64] Rathijit Sen and David A. Wood. 2013. Reuse-based online models for caches. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '13, Pittsburgh, PA, USA, June 17-21, 2013*. ACM, Pittsburgh, PA, USA, 279–292.

[65] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. 2020. IOctopus: Outsmarting Nonuniform DMA. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, Lausanne, Switzerland, 101–115.

[66] Akshitha Sriraman and Thomas F. Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX, Carlsbad, CA, USA, 177–194.

[67] Ron Stein. 2021. Key drivers of 100Gbps network adoption. https://web.archive.org/web/20210416000644/https://www.datacenterdynamics.com/en/opinions/key-drivers-100gbps-network-adoption/ Online; archived 16-April-2021; accessed 29-September-2021.

[68] Per Stenström, Truman Joe, and Anoop Gupta. 1992. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture. Gold Coast, Australia, May 1992*. ACM, Gold Coast, Australia, 80–91.

[69] supermicro. 2017. SUPERSERVER® 1028UX-CR-LL1 1028UX-CR-LL2 USER'S MANUAL. https://web.archive.org/web/20220105080830/https://www.supermicro.com/manuals/superserver/1U/MNL-1702.pdf [Online; archived 5-January-2022; accessed 5-January-2022].

[70] Simon M. Tam, Harry Muljono, Min Huang, Sitaraman Iyer, Kalapi Royneogi, Nagmohan Satti, Rizwan Qureshi, Wei Chen, Tom Wang, Hubert Hsieh, Sujal Vora, and Eddie Wang. 2018. SkyLake-SP: A 14nm 28-Core xeon® processor. In *2018 IEEE International Solid-State Circuits Conference, ISSCC 2018, San Francisco, CA, USA, February 11-15, 2018*. IEEE, San Francisco, CA, USA, 34–36.

[71] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. 2010. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*. IEEE, Bangalore, India, 1–12.

[72] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. 2020. Packet Chasing: Spying on Network Packets over a Cache Side-Channel. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, Valencia, Spain, 721–734.

[73] David E. Taylor and Jonathan S. Turner. 2005. ClassBench: a packet classification benchmark. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*. IEEE, Miami, FL, USA, 2068–2079.

[74] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. 2018. Dark packets and the end of network scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems, ANCS 2018, Ithaca, NY, USA, July 23-24, 2018*. ACM, Ithaca, NY, USA, 1–14.

[75] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina J. Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. USENIX, Renton, WA, USA, 283–297.

[76] Intel & Intel's Users. 2016. DDIO hit/miss metric? (or PCIeItoM not ticking). https://web.archive.org/web/20201020093626/https://community.intel.com/t5/Software-Tuning-Performance/DDIO-hit-miss-metric-or-PCIeItoM-not-ticking/m-p/1124331 [Online; archived-20-October-2020; accessed-6-January-2022].

[77] Intel & Intel's Users. 2017. pcie bandwidth drops on Skylake-SP. https://web.archive.org/web/20220112023100/https://community.intel.com/t5/Software-Tuning-Performance/pcie-bandwidth-drops-on-Skylake-SP/m-p/1167451 [Online; archived-6-January-2022; accessed-6-January-2022].

[78] Intel & Intel's Users. 2021. DDIO does not reduce Memory Read Bandwidth Despite 100% PCIRdCur Hit Rate. https://web.archive.org/web/20220112021118/https://community.intel.com/t5/Processors/DDIO-does-not-reduce-Memory-Read-Bandwidth-Despite-100-PCIRdCur/m-p/1325633 [Online; archived-6-January-2022; accessed-6-January-2022].

[79] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. 2013. Studying multicore processor scaling via reuse distance analysis. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*. ACM, Tel-Aviv, Israel, 499–510.

[80] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (1995), 20–24.

[81] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, San Francisco, CA, USA, 888–904.

[82] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't Forget the I/O When Allocating Your LLC. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, Valencia, Spain, 112–125.

[83] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is Network the Bottleneck of Distributed Training?. In *Proceedings of the 2020 Workshop on Network Meets AI & ML, NetAI@SIGCOMM, Virtual Event, USA, August 14, 2020*. ACM, Virtual Event, USA, 8–13.

[84] Li Zhao, Ravi R. Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. 2010. NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Proceedings of the 7th Conference on Computing Frontiers, 2010, Bertinoro, Italy, May 17-19, 2010*. ACM, Bertinoro, Italy, 121–130.

[85] Minshu Zhao and Donald Yeung. 2015. Studying the impact of multicore processor scaling on directory techniques via reuse distance analysis. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*. IEEE, Burlingame, CA, USA, 590–602.

[86] Ying Zheng, Brian T. Davis, and Matthew Jordan. 2004. Performance evaluation of exclusive cache hierarchies. In *2004 IEEE International Symposium on Performance Analysis of Systems and Software, March 10-12, 2004, Austin, Texas, USA, Proceedings*. IEEE, Austin, Texas, USA, 89–96.

[87] Dimitrios Ziakas, Allen Baum, Robert A. Maddox, and Robert J. Safranek. 2010. Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *IEEE 18th Annual Symposium on High Performance Interconnects, HOTI 2010, Google Campus, Mountain View, California, USA, August 18-20, 2010*. IEEE, GMountain View, California, USA, 1–6.

A EXPERIMENT SETTINGS

A.1 Hardware Details

Table 4. Hardware specification of our testbeds.

	skylake	*broadwell*
Processor	Xeon Gold 6132@2.60GHz 1 socket × 14 cores × 2 hyper-threads 64 sets × 8 ways = 32 KiB L1d & L1i per core 1024 sets × 16 ways = 1 MiB L2 per core 14 × 2048 sets × **11 ways** = 19.25 MiB non-inclusive LLC	Xeon E5-2650 v4@2.20GHz 2 sockets × 12 cores × 2 hyper-threads 64 sets × 8 ways = 32 KiB L1d & L1i per core 512 sets × 8 ways = 256 KiB L2 per core 12 cores × 2048 sets × **20 ways** = 30 MiB inclusive LLC
Memory	32 GiB DDR4 @2666 MHz 2 channels × 2 banks = 128 GiB	16 GiB DDR4 @2400 MHz 4 channels × 2 banks = 128 GiB
NIC	Mellanox ConnectX-5 100Gbps × 2 ports	Intel XL710 40Gbps × 2 ports
Operating System	Ubuntu 18.04 Linux kernel 4.15.0-158-generic	Ubuntu 16.04 Linux kernel 4.4.0-186-generic

We provide the detailed hardware specifications of our machines in two testbeds in Tab. 4. As the *skylake* microarchitecture is the *tock* step in Intel's famous tick-tock production plan, it applies a brand new microarchitecture, where one of the most significant changes is the adoption of the non-inclusive cache hierarchy.

As shown in Fig. 20, Intel conventionally describes the coherence states of cache lines using the *Modified (M)*, *Exclusive (E)*, *Shared (S)*, *Invalid (I)*, and *Forward (F)* (MESIF in abbreviation) five states, among which cores can read cache lines in MES states and write to only those in ME states. For example, if a core wants to write to a cache line that is not in its L2 (*I* state, implicitly), it issues a Request for Ownership (RFO) transaction to LLC (precisely, to the Cache Agent). If a copy of this cache line resides in another L2, LLC retrieves the latest value and invalidates the copy by sending a coherence request to the current owner. Data and permission are then responded to the new owner. The core allocates a cache line in its L2, stores the latest value, and marks its state as *Exclusive* or *Modified*, depending on whether the cache line is clean (i.e., whether its value is identical to the copy in the memory). The stalled write operation then continues as normal.

In the inclusive world, LLC records states of all cache lines, including the locations and coherence states of all copies. LLC thus knows what coherence transactions should be issued, and where

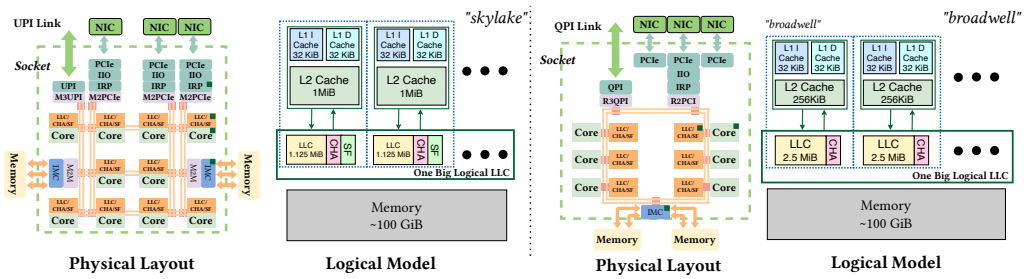


Fig. 18. Architecture sketch of Intel's *skylake* [70] and *broadwell* [47] microarchitecture. Intel has designed a sliced LLC connected using an on-chip network [41] with a ring topology in *broadwell* and a mesh topology in *skylake*. Cache sizes of different levels are also re-balanced along with the switch from inclusive LLC to exclusive LLC.

State	Op	Dirty	Exclusive
Modified	RW	Yes	Yes
Exclusive	RW	No	Yes
Shared	R	Yes	No
Invalid	No	No	No
Forward	R	No	Yes

Fig. 19. Properties of different cache line states.

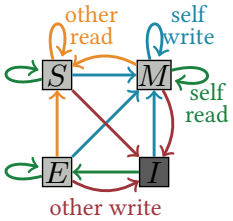


Fig. 20. Transitions on MESI protocol.

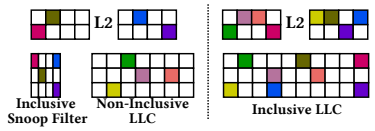


Fig. 21. Comparisons between inclusive and non-inclusive cache hierarchies.

coherence requests and responses should be sent. In the non-inclusive world, a new component called the Snoop Filter (SF) is added to record the states of cache lines not in LLC. The CA now concurrently looks up LLC and SF for requests; the lookup thus may also meet cache lines in *SF E*, *SF H*, and *SF S* state other than the conventional MESIF states, as revealed in Intel's *uncore* reference [36]. However, there is no official explanation of the meanings of these three states. In our experiments, we find that looking up the cache lines present only in L2 reaches *SF E* states. We thus believe that *SF E* means Snoop Filter Exclusive and is assigned to cache lines present only in L2.

It's suspected that the terminology "non-inclusive cache" as used by Intel indicates implicitly that it is neither exclusive nor inclusive, but something in between, such that cache lines may be present in both LLC and L2 under certain conditions. We guess that cache lines concurrently accessed by more than two cores are stored in both LLC and L2 to improve the performance of the cache coherence protocol. *SF H* and *SF S* may refer to *Snoop Filter Hybrid* and *Snoop Filter Shared* and assigned to dirty and clean cache lines with multiple copies in the cache, respectively. However, we need more requirements to validate it.

A.2 Workload Details

Algorithm 1: Packet Acrobat

```
Input: mode, read ratio, write ratio, flush ratio
for do
    if mode is "receive" or "forward" then
        pkt ← receive_pkt();
    else
        pkt ← gen_pkt();
    set_mac_address();
    start, end ← (0, ceil(pkt.len() * read_ratio)) ;
    temp ← pkt[start : end];
    start, end ← (end, end + ceil(pkt.len() * write_ratio));
    pkt[start : end] ← rand();
    start, end ← (end, end + ceil(pkt.len() * flush_ratio));
    flush(pkt[start:end]);
    if mode is "forward" or "send" then
        send(pkt);
```

A.2.1 packet-acrobat.

Algorithm 2: LLC Distractor

Input: array x with length L, step s
while do
  for $i = 0; i < L; i += s$ **do**
    x[i] ⟵ 1.0 + x[(i + L - s) % L];

A.2.2 LLC distractor. L controls the size of the working set, and s controls speed at which cache lines are retrieved. Large L and s exacerbate the contention on LLC.

A.2.3 network functions and realistic traces.
Packet Traces We use daily traces captured in an Internet backbone network provided by the MAWI project[23]. We use only unfragmented IPv4 packets containing TCP or UDP data, which occupies more than 90% of the original trace. We select five 30-seconds traces from daily traces captured between December 1 and December 5, which contains 1,174,639 unique TCP flows and 344,422 UDP flows in total. We load the traces to memory and replay them at the maximum available speed. Our generator reaches 71.8 Gbps (or 14.9 Mpps, with an average packet length of 603 bytes).
Network Functions We port all network functions from DPDK's examples and reuse core functionalities (e.g, LPM routing and IPSec encryption) provided by DPDK. It provides consistent code quality and optimization levels; for example, all NFs utilize SIMD instructions.

- **L3 Router** We implement an IPv4 core router that finds the next hop for each packet using Longest Prefix Match (LPM) algorithm. We use DPDK's `rte_lpm` library, which internally implements a variation of the DIR-24-8 algorithm. We extract address blocks from a Tier-1 router (using data provided by Routerview [57]) and add one entry for each address block to construct a large routing table consisting of 937,710 entries. It reaches 14.9 Mpps and, equivalently, 71.8 Gbps in our testbed.
- **Load Balancer** We implement a maglev[18]-like stateless L4 load balancer using DPDK's `rte_hash` library. Specifically, we use the five tuple of each packet as the hash key, then use the hash value as the index to get the destination node from the maglev mapping list. It reaches 14.9 Mpps and, equivalently, 71.8 Gbps in our testbed.
- **Firewall** We implement an ACL-based simple firewall using DPDK's `rte_acl` library. We generate a synthesized ruleset containing 10,000 rules using ClassBench [73]. It reaches 10.7 Mpps and, equivalently, 50.2 Gbps throughput in our testbed.
- **IPSec Gateway** We implement an IPSec gateway working in the ESP [43] IPv4 tunnel mode that encrypts all incoming packets with the cipher suite `aes-128-gcm` using DPDK's `rte_ipsec` library. It reaches 9.48 Mpps and, equivalently, 44.7 Gbps throughput on our testbed.

A.3 Hardware Performance Counter

Modern processors provide a set of registers called hardware performance counters (HPC) to count specific hardware events, such as issuing instructions or looking up LLC. HPC helps performance evaluation and diagnosis in depth. We summarize how we utilize HPCs following the reference manuals [34, 36, 37] on *broadwell* in Tab. 5 and *skylake* in Tab. 6.

A.4 Cache Allocation Technique

Intel supports cache allocation technology (CAT) [35], which provides hardware isolation on LLC between different cores. CAT works at the granularity of way, similar to that in DDIO. A process is

Table 5. Counters and events used in all experiments for *broadwell* microarchitecture.

Metrics	Counter	Event	Config	Note
PCIe write hit rate (full)	Uncore CBo	UNC_C_TOR_INSERTS. {LOCAL_OPCODE, MISS_LOCAL_OPCODE}	opcode = ItoM, nid=3F	
PCIe write hit rate (partial)		UNC_C_TOR_INSERTS. {LOCAL_OPCODE, MISS_LOCAL_OPCODE}	opcode = RFO, nid=3F	$1 - \frac{\text{MISS\_LOCAL\_OPCODE}}{\text{LOCAL\_OPCODE}}$
PCIe read hit rate		UNC_C_TOR_INSERTS. {LOCAL_OPCODE, MISS_LOCAL_OPCODE}	opcode = PCIRdCur, nid=3F	
latency for miss cache request x	Uncore CBo	UNC_C_TOR_INSERTS.LOCAL_MISS_OPCODE UNC_C_TOR_OCCUPANCY.LOCAL_MISS_OPCODE	opcode = x, nid=3F	
latency for hit cache request x	Uncore CBo	UNC_C_TOR_INSERTS. {LOCAL_OPCODE, LOCAL_MISS_OPCODE} UNC_C_TOR_OCCUPANCY. {LOCAL_OPCODE, LOCAL_MISS_OPCODE}	opcode = x, nid=3F	$\frac{\text{OCCUPANCY}}{\text{INSERTS}}$
number of requests meeting cache lines in given state x	Uncore CBo	UNC_C_LLC_LOOKUP. {DATA_READ, READ, WRITE}	state = x	
memory write traffic memory read traffic	Uncore IMC	UNC_M_CAS_COUNT.RD UNC_M_CAS_COUNT.WR		RD × 64 WR × 64
LLC {reference, Miss}	Core	LLC_{REFS, MISSES}		
L2 Data Read {Hit, Miss}	Core	L2_RQSTS_DEMAND_DATA_RD_{HIT, MISS}		
L2 Data Write {Hit, Miss}	Core	L2_RQSTS_RFO_{HIT, MISS}		

Table 6. Counters and events used in all experiments for *skylake* microarchitecture.

Metrics	Counter	Event	Config	Note
PCIe write hit rate (full)	Uncore CHA	UNC_CHA_TOR_INSERTS.{IO_HIT, IO_MISS}	opc0 = ItoM, tid_en=0	$\frac{\text{IO\_HIT}}{\text{IO\_HIT+IO\_MISS}}$
PCIe write hit rate (partial)		UNC_CHA_TOR_INSERTS.{IO_HIT, IO_MISS}	opc0 = RFO, tid_en=0	
PCIe read hit rate		UNC_CHA_TOR_INSERTS.{IO_HIT, IO_MISS}	opc0 = PCIRdCur, tid_en=0	
latency for miss cache request x	Uncore CHA	UNC_CHA_TOR_INSERTS.IO_MISS UNC_CHA_TOR_OCCUPANCY.IO_MISS	opc0 = x, tid_en=0	$\frac{\text{OCCUPANCY}}{\text{INSERTS}}$
latency for hit cache request x	Uncore CHA	UNC_CHA_TOR_INSERTS.IO_HIT UNC_CHA_TOR_OCCUPANCY.IO_HIT	opc0 = x, tid_en=0	
number of requests meeting cache lines in given state x	Uncore CHA	UNC_CHA_REQUESTS.{INVITOE_LOCAL, READS_LOCAL, WRITES_LOCAL}	state = x	
memory write traffic memory read traffic	Uncore IMC	UNC_M_CAS_COUNT.RD UNC_M_CAS_COUNT.WR		RD × 64 WR × 64
LLC {reference, Miss}	Core	LLC_{REFS, MISSES}		
L2 Data Read {Hit, Miss}	Core	L2_RQSTS_DEMAND_DATA_RD_{HIT, MISS}		
L2 Data Write {Hit, Miss}	Core	L2_RQSTS_RFO_{HIT, MISS}		
{Dirty, Clean} cache lines evicted from L2	Core	L2_LINES_OUT.{NON_SILENT, SILENT}		
latency for PCIe Read operation	Uncore IIO	UNC_IIO_COMP_BUF_INSERTS.CMPD UNC_IIO_COMP_BUF_OCCUPANCY.CMPD		$\frac{\text{OCCUPANCY}}{\textit{INSERTS}}$
latency for PCIe Write operation	Uncore IRP	UNC_I_CACHE_TOTAL_OCCUPANCY.MEM UNC_I_COHERENT_OPS.{PCITOM, RFO}		$\frac{\text{OCCUPANCY}}{\textit{INSERTS}}$
{Package, DRAM} Power		MSR_{PKG, DRAM}_ENERGY_STATUS		

allowed to allocate new cache lines only in ways configured by CAT. We apply CAT to control the LLC space available for an application to reduce variance in its performance or infer its behavior. By convention, a bitmask is applied to denote LLC ways. For example, `0x700` represents the three leftmost ways, given a total of 11 ways, and `0x0F000` means the fifth to the eighth ways of a total of 20 ways.

B SYMBOL REFERENCE TABLE

C WELL-KNOWN FORMULAE

We list some well-known formulae here for quick reference.

Equation 5 gives the probability distribution function of binomial distribution with parameters n and p. It means that an experiment with probabilities p of succeeding and $1 - p$ of failing succeeds

Table 7. Symbols and descriptions of parameters used in the model

Category	Symbol	Description
Cache Specification	C_x	number of sets in a set-associative cache x
	N_x	associativity of a set-associative cache x
	B	cache line size counted in bytes
System Configuration	Q_{RX}	RX descriptor ring size
	Q_{TX}	TX descriptor ring size
	S	the number of cores used in networking application.
Workload Properties	M	packet size counted by the number of cache lines
	α	the portion of packet accessed by the processor.
	S	the number of cores used in networking applications.
Measurement Result	L_{RX}	average length of RX waiting queue
	$L_{RX/R}$	average length of RX Replenish queue
	L_{TX}	average length of TX waiting queue.
	$L_{TX/R}$	average length of TX recycling queue.

for k times in n individual and identical repeats.

$$f_{\mathcal{B}}(k;n,p) = \binom{n}{k} p^k (1-p)^{n-k}, \forall k \in \mathbb{N}, k \leq n \tag{5}$$

Equation 6 gives the cumulative distribution function of binomial distribution with parameters n and p. It means that an experiment with probabilities p of succeeding and $1-p$ of failing succeeds for at most k times in n individual repeats.

$$F_{\mathcal{B}}(k;n,p) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}, \forall k \in \mathbb{N}, k \leq n \tag{6}$$

We extend the domain of the definitions of f_B and F_B to $k > n$ as follows to simplify the notations in the derived equations.

$$f_B(k;n,p) = 0, F_B(k;n,p) = 1, \forall k \in \mathbb{N}, k > n \tag{7}$$

Equation 8 gives the probability distribution function of the negative binomial distribution. It represents the probability of k successes occuring before r failures if the experiment is repeated, with probability p of succeeding and $1-p$ of failing.

$$f_{\mathcal{NB}}(k;r,p) = \binom{k+r-1}{k} (1-p)^r p^k, \forall k \in \mathbb{N} \tag{8}$$

D MODEL EXTENSION

D.1 Shared Ways

If `L3*` overlaps with `DDIO` or `Shared` ways, cache lines brought into LLC such as those evicted from L2 may also be brought to *DDIO* or *Shared* if the victim happens to be there. A cache line in `DDIO` or `Shared` ways may be evicted due to collisions with memory access to `L3*`. We discuss how these two effects are integrated into our model.

If a new cache line x in DDIO ways resides in the cache after $T_{\texttt{DDIO}}$ unique `DDIO` accesses and T_{L3^*} unique `L3*` accesses arrives, two conditions must be met: (1) The number of cache lines allocated in the DDIO portion must be smaller the number of DDIO ways. (2) The number of cache lines assigned to the set must be smaller than the number of ways. The probability then is

$$p_x^x(T, T_D; N, \alpha, C) = \sum_{i=0}^{\alpha-1} \left\{ f_B\left(T_D, i, \frac{1}{C}\right) F_B\left(T, N-1-i, \frac{1}{C}\right) \right\} \tag{9}$$

For a cache line x in non-DDIO ways, only condition (2) is required. But regardless of how many unique *explicit DDIO requests* arrive, they occupy no more than α DDIO ways. We thus get

$$p_{\text{L3-}\rightarrow\text{L3-}}(T, T_{IO}; N, \alpha, C) = \sum_{i=0}^{T_{IO}} \left\{ f_B\left(T_{IO}, i, \frac{1}{C}\right) F_B\left(T, N-1-\min(i, \alpha), \frac{1}{C}\right) \right\} \tag{10}$$

The probabilities a new cache line is placed in different ways may not be equal. For example, if DDIO requests are the majority in the memory access stream, the two DDIO ways are used more frequently. Cache lines in these two ways are more likely to be younger than those in the rest non-DDIO ways, making new cache lines that are not restricted more likely to be placed at the non-DDIO ways.

Consider an N-associative LRU cache receiving a sequence of unique memory requests. Each request has a probability α of being restricted in allocating in only the m leftmost ways (called restricted ways), similar as how DDIO requests are restricted in DDIO ways. We are concerned with the probability p that the cache line for an unrestricted request is allocated in the restricted ways. Intuitively, if α is larger, the restricted ways are used more frequently and are thus more likely to be younger, resulting in a higher p. In the extreme case $\alpha \rightarrow 1$, the remaining ways are almost never touched, so they are free for use, making the probability close to 1 ($p \rightarrow 1$). If $\alpha = 0$, then there is no difference between all ways, and the probability is just $\frac{m}{N}$ ($p = \frac{m}{N}$).

If we sort all cache lines in a set from the youngest to the oldest and record the order of a cache line as its age, we can define an age vector $(A_0, A_1, \cdots, A_m)$ for the restricted ways to model the probability they are selected for unrestricted requests. Notice that the cache line with $A_i = N - 1$ is selected as the victim on cache allocation. Use $A_{(n)}$ to denote the nth-order statistic of $A_0, A_1, \cdots, A_m$. We define a diff age vector for them as

$$(x_0, x_1, \cdots, x_m), x_i = A_{(i)} - A_{(i-1)} \tag{11}$$

If a new cache line is allocated in the remaining $N - m$ ways, all m cache lines age by 1, and the diff age vector becomes $(x_0 + 1, x_1, \cdots, x_m)$. Otherwise, the oldest cache line in the m ways is evicted, and the diff age vector becomes $(1, x_1, \cdots, x_{m-1})$.

For example, given $N = 5$ and $m = 3$, if the ages of the 3 leftmost cache lines are $5, 2, 3$, the diff age vector is $(2, 1, 2)$. A new cache line will be allocated in the three restricted ways, which converts the age vector and diff age vector to $(1, 3, 4)$ and $(1, 2, 1)$, respectively. The next new cache line will be allocated at the remaining two ways, yielding an age vector $(2, 4, 5)$ and a diff age vector $(2, 2, 1)$.

We build a Markov chain to describe the transition on the diff age vector of a representative set as follows:

$$x_i \in \mathbb{Z}^*, \sum x_i < n : \begin{pmatrix} 1 \\ x_1 \\ \cdots \\ x_{m-1} \end{pmatrix} \xleftarrow[\sum x_i \neq n]{\alpha} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_m \end{pmatrix} \xrightarrow[\sum x_i \neq n]{1-\alpha} \begin{pmatrix} x_1+1 \\ x_2 \\ \cdots \\ x_m \end{pmatrix}$$

$$x_i \in \mathbb{Z}^*, \sum x_i = n : \begin{pmatrix} 1 \\ x_1 \\ \cdots \\ x_{m-1} \end{pmatrix} \xleftarrow[\sum x_i = n]{1} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_m \end{pmatrix}$$

$\sum x_i = n$ means that the oldest cache line resides in the m restricted ways, and will be selected as the victim no matter whether the next access is restricted or not. However, $\sum x_i \neq n$ means that the oldest cache lines reside elsewhere, and cache lines in the m leftmost ways are selected only when the next access is restricted. Solving the Markov chain, we get

$$P\begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_m \end{pmatrix} = \frac{(1-\alpha)^{(\sum x_i - m)}}{\sum_{k=0}^{n-m}\left[\binom{m-1+k}{m-1}(1-\alpha)^k\right]}$$

Thus, the probability that an unrestricted memory access reaches the m restricted ways is

$$p = \sum_{\sum x_i = n} P\begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_m \end{pmatrix} = \frac{\binom{n-1}{m-1}(1-\alpha)^{(n-m)}}{\sum_{k=0}^{n-m}\left[\binom{m-1+k}{m-1}(1-\alpha)^k\right]} \tag{12}$$

We draw the placement probability given different values of n, m, and α in Fig. 22. When α is larger than 50%, i.e., DDIO requests are the majority in the system, the program data is very unlikely to be placed in DDIO ways as if there is a soft isolation between `DDIO` and `L3*`. We may omit the probability evicted cache lines are placed on `DDIO` or `Shared` if DCA (restricted) requests are the majority. Otherwise both the reuse distances of cache lines in different cache regions (§4.3.1) and L2-related elements in the eviction matrix (§4.3.3) should be modified using Eq. 12.

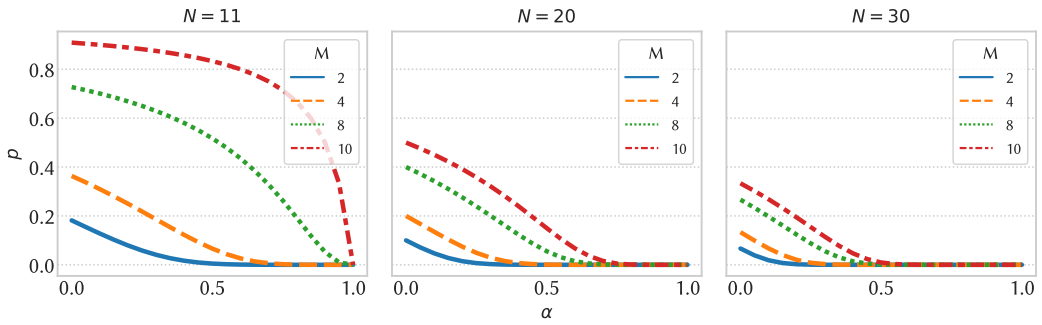


Fig. 22. Relationship between the probability that unrestricted memory requests are allocated in DDIO ways and the ratio of restricted memory requests α, the number of ways N, and the number of restricted ways M.

D.2 Imbalance Load Between Multiple Cores

If the load is not evenly distributed among multiple cores, queue lengths differ in each ring. As a result, we should model the hit rate of each core individually and estimate the system-wise hit rate using an average of core hit rates weighted by the load. For the reuse distance estimation, as the processing speed between different cores is not matched, when a packet crosses a queue of length L, other cores may process more or less. Processed packets in other cores can be estimated using the relative processing speed.

We denote the traffic reaching cores 0 to $N-1$ as $t_0, t_1, \cdots, t_{N-1}$, and define the traffic factor β_i as

$$\beta_i = \frac{\sum_k t_K}{t_i}$$

We build N dedicated models, one for each core, to replace the equivalent number of cores S with β_i as and $L_{\text{RX},i}$, $L_{\text{RX/R},i}$, L_{TX}, and $L_{\text{TX/R}}$, i.e., queue length measured for each queue in the ith model. We calculate the equivalent hit rate as

$$p(x,y) = \frac{1}{\beta_i} p_i(x,y), \forall x \in M, y \in Q$$

And the reuse distance for model i is

$$C_{x,i}(T) = \mathcal{K}_x \beta_i ML, x \in \{\texttt{DDIO}, \texttt{Shared}, \texttt{L3*}\}$$

D.3 Cache Lines with Mixed Access Patterns

Cache lines may not be used identically for two reasons: First, packet sizes are varied so that cache lines at a larger offset in the buffer are less likely to be used. Second, the application may not touch all cache lines; for example, an L3 router reads and updates only the first cache line in a packet.

We denote the maximum buffer length by N (counted by the number of cache lines), and build N models for cache lines at the offset from 0 to $N-1$. Given the packet size distribution $\mathcal{D}$, the probability that the cache line at offset i is used is

$$d_i = \begin{cases} 1 - F_{\mathcal{D}}(i-1), & i > 0 \\ 1, & i = 0 \end{cases}$$

Note that the average packet length is $\sum d_i$ and the equivalent hit rate is

$$p(x,y) = \frac{1}{\sum d_i} \sum d_i p_i(x,y), \forall x \in M, y \in Q$$

If the probability that cache line i is touched by the processor is t_i, the transition matrix of processor access for the ith model is

$$\begin{bmatrix} 1 & t_i & t_i & t_i & t_i \\ 0 & 1-t_i & 0 & 0 & 0 \\ 0 & 0 & 1-t_i & 0 & 0 \\ 0 & 0 & 0 & 1-t_i & 0 \\ 0 & 0 & 0 & 0 & 1-t_i \end{bmatrix}$$

Although it is possible to write a networking application that randomly manipulates packets, we discuss only a practical access pattern here: A cache line at offset i is either touched or not, and so $t_i = 0$ or $t_i = 1$. It is true for all workloads considered in this paper. We also assume that the application respects the packet length, which means that cache lines written by the NIC when receiving a packet are also read by the NIC on transmissions.

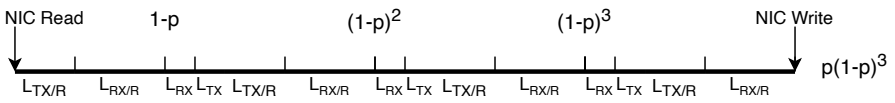


Fig. 23. An example of a cache line with probability p of being written by NIC that is not written over three iterations.

If a cache line is not touched by the NIC or the processor, its average reuse distance to the next access is extended. Fig. 23 gives such an example. Consider a cache line at offset i, the probability that it resides in the cache till the next PCIe write is

$$p_x^x = \sum_{k=0}^{\infty} d_i (1-d_i)^k p_x^x(C_x(T_k))$$

The extended reuse distance $C_x(T_k)$ is

$$C_x(T_k) = \mathcal{K}_x S\mathcal{R}(L_{\text{TX}} + L_{\text{TX/R}} + k(L_{\text{TX}} + L_{\text{TX/R}} + L_{\text{RX}} + L_{\text{RX/R}}))$$

It is important that the stack reuse distance is not proportional to the queue length because buffers are used circularly. We may thus use the same cache line more than once during the long wait. The effective queue length is

$$\begin{aligned}\mathcal{R}(L_{\text{TX}} + L_{\text{TX/R}} + k(L_{\text{TX}} + L_{\text{TX/R}} + L_{\text{RX}} + L_{\text{RX/R}})) &= \sum_{i=0}^{k-1}\sum_{j=0}^{N-1}(1-d_j)^i d_j (L_{\text{TX}} + L_{\text{TX/R}} + L_{\text{RX}} + L_{\text{RX/R}}) \\ &+ \sum_{j=0}^{N-1}(1-d_j)^k d_j (L_{\text{TX}} + L_{\text{TX/R}}) \\ &\le N(L_{\text{TX}} + L_{\text{TX/R}} + L_{\text{RX}} + L_{\text{RX/R}})\end{aligned}$$

D.4 Only TX / RX Path, or More Software Queues

For workloads containing only the TX path, such as the traffic generator, we can cut the original four logical states to only *TX* and *TX Recycle* while keeping all the remaining methods. Similarly, an application may employ extra software queues. For example, in the RPC framework with network threads and worker threads, the network thread moves packets from the hardware ring to a software queue while worker threads consume packets from it. Waits in software queues should also be counted by adding an extra logical state for each software queue.

D.5 Addresses Unevenly Mapped to the Entire Cache

Memory addresses of network buffers may not be evenly mapped to the cache if addresses expose specific patterns. For example, in the Linux kernel network stack, buffers should be aligned with the 4K page boundary such that the first cache line in each buffer must be 4k-aligned. Due to Intel's placement policy, only 32 sets are available for the 4k-aligned cache line in each slice [72]. As a result, we have to divide the cache and memory access into multiple groups according to their mapping relationships, and model each separately.

Received October 2021; revised December 2021; accepted January 2022