A4: Microarchitecture-Aware LLC Management for Datacenter Servers with Emerging I/O Devices

Haneul Park
University of Illinois Urbana-Champaign
Urbana, USA
hnpark2@illinois.edu

Jiaqi Lou
University of Illinois Urbana-Champaign
Urbana, USA
jiaqil6@illinois.edu

Sangjin Lee
Chung-Ang University
Seoul, Republic of Korea
tkdwls0727@cau.ac.kr

Yifan Yuan
Meta
Menlo Park, USA
yifanyuan@meta.com

KyoungSoo Park
Seoul National University
Seoul, Republic of Korea
kyoungsoo93@snu.ac.kr

Yongseok Son
Chung-Ang University
Seoul, Republic of Korea
sysganda@cau.ac.kr

Ipoom Jeong
Yonsei University
Seoul, Republic of Korea
ipoom@yonsei.ac.kr

Nam Sung Kim
University of Illinois Urbana-Champaign
Urbana, USA
nskim@illinois.edu

Abstract

In modern server CPUs, the Last-Level Cache (LLC) serves not only as a victim cache for higher-level private caches but also as a buffer for low-latency DMA transfers between CPU cores and I/O devices through Direct Cache Access (DCA). However, prior work has shown that high-bandwidth network-I/O devices can rapidly flood the LLC with packets, often causing significant contention with co-running workloads. One step further, this work explores hidden microarchitectural properties of the Intel Xeon CPUs, uncovering two previously unrecognized LLC contentions triggered by emerging high-bandwidth I/O devices. Specifically, (C1) DMA-written cache lines in LLC ways designated for DCA (referred to as DCA ways) are migrated to certain LLC ways (denoted as inclusive ways) when accessed by CPU cores, unexpectedly contending with non-I/O cache lines within the inclusive ways. In addition, (C2) high-bandwidth storage-I/O devices, which are increasingly common in datacenter servers, benefit little from DCA while contending with (latency-sensitive) network-I/O devices within DCA ways. To this end, we present *A4*, a runtime LLC management framework designed to alleviate both (C1) and (C2) among diverse co-running workloads, using a hidden knob and other hardware features implemented in those CPUs. Additionally, we demonstrate that *A4* can also alleviate other previously known network-I/O-driven LLC contentions. Overall, it improves the performance of latency-sensitive, high-priority workloads by 51% without notably compromising that of low-priority workloads.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731114

CCS Concepts

• **Computer systems organization** → *Client-server architectures.*

Keywords

Direct Cache Access, Non-inclusive caches, Last-Level Cache, Datacenter server

ACM Reference Format:
Haneul Park, Jiaqi Lou, Sangjin Lee, Yifan Yuan, KyoungSoo Park, Yongseok Son, Ipoom Jeong, and Nam Sung Kim. 2025. A4: Microarchitecture-Aware LLC Management for Datacenter Servers with Emerging I/O Devices. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731114

1 Introduction

The on-chip Last-Level Cache (LLC) in modern server CPUs plays an important role in providing high performance and energy efficiency for datacenter servers, with much lower latency and energy consumption per access than the off-chip memory. For decades, several innovations in LLC architecture, technology, and management have been introduced to further improve performance and energy efficiency [28]. The latest innovations include (1) the evolution of the LLC from inclusive to non-inclusive cache architectures [24], (2) configurable LLC allocation per CPU core or process [23], and (3) Direct Cache Access (DCA) technology [21]. They facilitate (1) more efficient utilization of limited LLC capacity, (2) less contention among workloads within the LLC, and (3) lower latency for DMA transfers between I/O devices and CPU cores directly through the LLC. Such design trends are not limited to certain CPU vendors. Major CPU vendors like Intel, AMD, and Arm have implemented DCA features, known as Data Direct I/O (DDIO) [27], Smart Data Cache Injection (SDCI) [4], and cache stashing [5], respectively. Moreover, recent Intel Xeon CPUs and AMD Zen 4 CPUs have adopted the non-inclusive LLC [58, 62, 65]. While these solutions have proven

effective, they have also introduced various unintended contentions within the LLC, especially for servers connected to high-bandwidth network-I/O devices that DMA-transfer packets to the LLC at hundreds of Gbps [2, 18, 44, 56, 59, 67]. Prior work has identified the sources of these contentions within the LLC and eased them with software-only and hardware-assisted solutions. Yet, we observe that some contentions within the LLC remain unexplained and unresolved by previous work. In this work, with the latest advances in understanding the architecture of recent server CPUs [65] and improving the performance of storage-I/O devices [51] in mind, we uncover two sources of hidden **C**ontentions within the LLC (§3).

(C1) Contention between I/O and non-I/O cache lines within hidden inclusive ways. In addition to the previously known network-I/O-driven contentions within LLC ways designated for DCA (referred to as 'DCA ways' henceforth) (§2.2), we uncover another contention between I/O and non-I/O cache lines within certain LLC ways (denoted as 'inclusive ways' hereafter) (§3.1). This contention arises from the unique directory architecture of the non-inclusive cache architecture (§2.1). Contemporary Intel CPUs have two groups of directory ways to track cache-coherence states of cache lines in the LLC and/or the Mid-Level Caches (MLCs), with 11 and 12 ways dedicated to each group, respectively. The two groups share two directory ways, each coupled one-to-one with inclusive ways. These inclusive ways are the only LLC ways capable of holding cache lines simultaneously in both the LLC and MLCs [65]. We reveal that DMA-written, LLC-exclusive cache lines in DCA ways are migrated to the inclusive ways when accessed by CPU cores and brought into MLCs. This causes a significant contention between I/O and non-I/O cache lines within inclusive ways, degrading the performance of both the I/O and non-I/O workloads when the non-I/O workloads are obliviously allocated to inclusive ways. We refer to such contention as 'directory contention' henceforward.

(C2) Contention between storage-I/O and network-I/O cache lines within DCA ways. Although DCA was originally designed for network-I/O devices, it indiscriminately allows any I/O devices, including storage-I/O devices, to DMA-write I/O data to DCA ways. In the past, when storage-I/O devices provided an order of magnitude lower bandwidth than network-I/O devices, contentions between storage-I/O and network-I/O cache lines within DCA ways were negligible. However, as the bandwidth of storage-I/O devices has increased to the same order of magnitude as that of network-I/O devices (*e.g.*, 116Gbps for an NVMe SSD [11]), we hypothesize that storage-I/O cache lines can significantly interfere with network-I/O cache lines at DCA ways. To validate the hypothesis, we set up a server configured similarly to a class of datacenter servers with multiple high-bandwidth I/O devices and demonstrate the following (§3.2). DCA does not improve storage-I/O throughput, especially with large I/O blocks and deep I/O queues—typical strategies for maximizing storage-I/O throughput [7, 38, 57]. Meanwhile, storage-I/O cache lines frequently evict network-I/O cache lines from DCA ways before those cache lines are consumed, which degrades the performance of network-I/O workloads.

Next, we propose *A4*, a runtime micro**a**rchitecture-**a**ware LLC m**a**nagement fr**a**mework, which helps users address both (C1) and (C2) when co-running workloads with varying priorities: High-Priority Workloads (HPWs) and Low-Priority Workloads (LPWs). Specifically, *A4* provides two key **F**unctions built with a little-known knob of DCA, Cache Allocation Technology (CAT) [23], and performance counters implemented in server CPUs (§5).

(F1) Preventing LPWs from being obliviously allocated to inclusive ways. To avoid (C1), *A4* starts by strategically eschewing the allocation of inclusive ways to LPWs. It then adaptively adjusts the number of standard ways (*i.e.*, LLC ways excluding DCA and inclusive ways) allocated to these LPWs to keep performance metrics—such as LLC hit rates of HPWs—within ranges set to maximize the overall performance. Meanwhile, *A4* does not explicitly allocate HPWs to specific LLC ways. Instead, it allows HPWs, whose performance is highly sensitive to accessible LLC capacity, to use as much LLC space as possible, including any unused capacity from the LLC ways allocated to LPWs. This allocation strategy enables priority-based LLC partitioning between HPWs and LPWs while promoting efficient LLC sharing among workloads with the same priority. It remains effective even when the number of co-running processes exceeds the available number of LLC ways—a common challenge in prior work, improving the performance of HPWs by 30% without notably degrading that of LPWs.

(F2) Selectively disabling DCA for storage-I/O devices. When detecting (C2), *A4* first exploits a little-known knob of Intel CPUs that can selectively disable DCA for storage-I/O devices. This will make *antagonistic* storage-I/O LPWs get storage-I/O blocks through the device-memory-MLC path instead of the device-DCA-MLC path, eschewing a contention between network-I/O and storage-I/O cache lines in DCA ways. After these storage-I/O cache lines are consumed, nonetheless, they will be eventually evicted to the LLC and contend with other cache lines in standard ways, a phenomenon known as DMA-bloat [2]. To ease such a contention, it also adaptively decreases the number of standard ways allocated to these storage-I/O LPWs, named trash ways, to as few as one. Overall, these strategies improve the performance of especially network-I/O HPWs by 63%, without compromising the performance of storage-I/O LPWs that are insensitive to DCA and LLC capacity.

Lastly, *A4* can be easily extended to mitigate other previously known network-I/O-driven contentions within the LLC (§2.2). Specifically, it can address the latent contention within DCA ways [67] by extending (F1) to stop non-I/O HPWs from using DCA ways. This is achieved by extending (F1) to explicitly allocate non-I/O HPWs to all LLC ways but DCA ways, while I/O HPWs remain not explicitly assigned to any LLC ways. *A4* can also ease the contention caused by DMA bloat from network-I/O workload by allocating network-I/O workloads only to trash ways in (F2) after detecting DMA bloat. This stops DMA-bloated network-I/O cache lines from contending with non-I/O cache lines within other LLC ways. Overall, *A4* improves the performance of HPWs by 51% without compromising that of LPWs.

2 Background

In this section, we briefly describe modern cache architecture, Direct Cache Access (DCA), and storage-I/O stack.

2.1 Modern Cache Architecture

Non-inclusive LLC. With more cores on a CPU, the limited capacity of the traditional inclusive LLC becomes a performance bottleneck, especially in multi-tenant environments [28]. Meanwhile, to

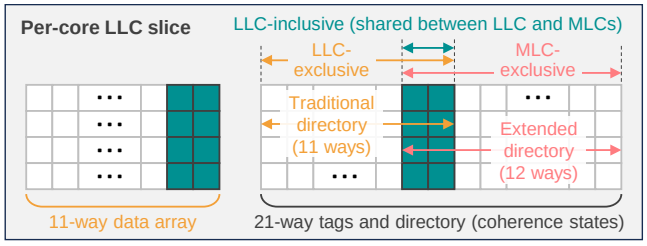


Figure 1: Reverse-engineered data array and directory structures of Intel Skylake CPUs [60, 65].

provide both high performance and strong isolation among CPU cores (*i.e.*, tenants), the capacity of private MLCs has increased, but that of the shared LLC cannot be proportionally increased under the chip-size constraint. As a solution, non-inclusive cache architectures have been adopted for LLCs, where CPU cores bring cache lines directly from memory to their private MLCs first upon LLC misses. Later, they might evict these cache lines to the LLC, which serves as a victim cache. When accessing these cache lines again, the CPU cores not only bring them into the MLCs but also leave them in the LLC until they are evicted from the LLC. As such, it can utilize a given LLC capacity more flexibly and efficiently for frequently accessed (shared) cache lines.

Inclusive directory. Recent work has comprehensively reverse-engineered the directory architecture for the non-inclusive LLC of the Intel Skylake CPU and uncovered the following [65]. The directory architecture consists of (1) 11 traditional directory ways that are one-to-one coupled with 11 LLC ways and (2) 12 extended directory ways that track the cache coherence states of cache lines in MLCs (Fig. 1). Two of the traditional directory ways are also used as two of the extended directory ways (the green region in Fig. 1). This gives rise to a coupling between the two hidden directory ways and two right-most LLC ways (also referred to as inclusive ways in this work). LLC-inclusive cache lines (*i.e.*, cache lines stored in both LLC and MLCs) are restricted to reside only in these two inclusive ways since other LLC ways cannot snoop MLCs, while LLC-exclusive cache lines may reside in any 11 ways.

2.2 Direct Cache Access

The key insight behind the introduction of DCA is that the LLC, instead of the memory, can serve as the source and destination of the DMA transfer of I/O data between CPU cores and I/O devices. As CPU cores are likely to consume DMA-written I/O data soon, putting them close to the CPU improves latency and reduces bandwidth consumption. If cache lines DMA-written by I/O devices are already present in the LLC, DCA performs in-place updates (*i.e.*, write update). Otherwise, new cache lines are allocated to DCA ways—typically the two leftmost ways—and I/O data are DMA-written to these cache lines (*i.e.*, write allocate). While DCA was originally introduced for low-latency network-I/O devices, it also allows any (PCIe-based) I/O devices, such as NVMe SSDs [3, 18, 31], to DMA-transfer I/O data to/from DCA ways.

Ingress path. The blue arrows in Fig. 2 depict the ingress path of conventional I/O with a non-inclusive cache architecture. After the allocation of kernel and user buffers, a given I/O device DMA-writes I/O data to the kernel buffer in the memory (①). Upon receiving

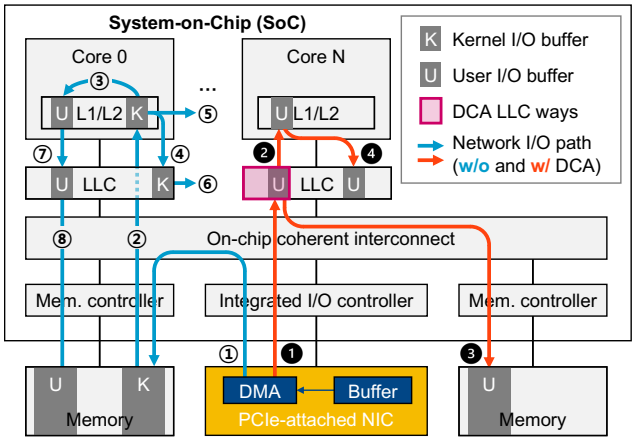


Figure 2: DMA paths of network I/O (blue: buffered I/O without DCA, red: kernel-bypass I/O with DCA)

an interrupt from the I/O device, CPU cores read the I/O data from the memory and then writes them to cache lines in their MLCs (②); we refer to cache lines storing I/O data as I/O cache lines henceforth. Then the CPU cores copy these I/O cache lines to cache lines storing a user buffer (user-buffer cache lines) for processing (③). Later, these I/O cache lines may be evicted to the LLC (④) or invalidated if reused/updated by the I/O device (⑥). Similarly, the user-buffer cache lines may reside in the MLCs or be evicted to the LLC (⑦) and/or to the memory (⑧). The red arrows show the ingress path of DCA-enabled, kernel-bypass network I/O such as DPDK [14], where a network-I/O device directly DMA-writes packets to cache lines in DCA ways (❶) instead of the memory. Then, the CPU cores read these I/O cache lines from the DCA ways (❷), obviating DMA-write to and then CPU-read from the memory (① and ②, respectively). Consequently, DCA effectively reduces the latency and bandwidth of accessing the memory for high-speed I/O. As these I/O cache lines are in a modified state and then written back to the memory (❸), the coherence is maintained.

Egress path. The I/O data flow of the egress path is the opposite of that of the ingress path. When I/O data written by CPU cores are only in their MLCs, I/O cache lines are copied to newly read-allocated cache lines in inclusive ways, and then DMA-read by I/O devices; if they are already in the LLC, they are DMA-read directly from the LLC [60]. If I/O data are not in the cache hierarchy, I/O data are DMA-read by I/O devices directly from the memory, which does not read-allocate any cache lines in the LLC [36]. Since this behavior is not described in the architecture reference document, it is easily misinterpreted and leads to incorrect conclusions [45].

Network-I/O-driven LLC contentions. A body of work has demonstrated network-I/O-driven contentions within DCA ways and other LLC ways. The contention within DCA ways can be further divided into two contentions. The first one, known as latent contention [67], occurs when a CPU core running a non-I/O workload is allocated to LLC ways that overlap with DCA ways, where cache lines storing network-I/O data (or simply network-I/O cache lines) contend with cache lines storing non-I/O data (or simply non-I/O cache lines) in DCA ways. The second one, referred to as DMA leak [18], happens when network-I/O cache lines in

DCA ways are evicted (❸) by other network-I/O cache lines before they are consumed by CPU cores (❷). The contention within other LLC ways can be caused by DMA bloat [2], occurring when consumed network-I/O cache lines in MLCs are evicted to LLC ways (❹) where (DMA-bloated) I/O cache lines and non-I/O cache lines contend. This breaks the isolation that DCA aims to enforce, *i.e.*, I/O cache lines remain in DCA ways when cached in the LLC.

2.3 Modern Storage-I/O Stack

The Linux kernel and user-space NVMe drivers (*e.g.*, SPDK [51]) have implemented software optimizations, such as Direct I/O [46] and the use of large I/O blocks [38] with deeper I/O queues [7, 57], for the storage-I/O stack to maximize throughput while minimizing performance cost. For example, Direct I/O is adopted by the standard Linux kernel and shares many features with Intel SPDK (*e.g.*, kernel bypassing). Similar to kernel-bypass network I/O, the Direct I/O (*e.g.*, the `read()` system call with the `O_DIRECT` flag) facilitates a workload to communicate with the SSD through a user buffer, bypassing the kernel page cache [43]. A body of work has investigated the interaction between network I/O and DCA [2, 18, 44, 56, 59, 60, 67]. Yet, less attention has been paid to the interaction between storage I/O and DCA, even though the NVMe SSD has begun to offer as high bandwidth as the NIC [34]. If the NIC in Fig. 2 is replaced with storage, the DMA path of Direct I/O with DCA is analogous to that of the network I/O (red arrows). This example assumes the user buffer has already been cached in LLC ways, and thus the corresponding I/O cache lines are write-updated in place (❶). Later, we show that storage I/O may cause significant contention between co-running workloads within both DCA ways and other LLC ways (§3.2).

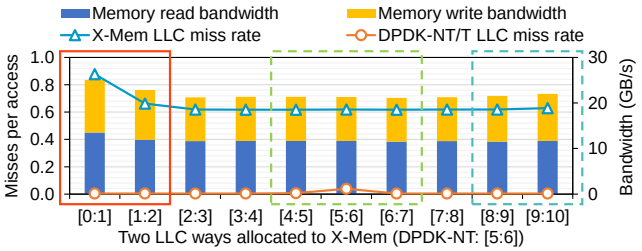
3 Newly-discovered I/O-driven LLC Contentions

In this section, we first uncover a previously unrecognized I/O-driven contention within inclusive ways coupled with two directory ways in a non-inclusive cache architecture. Second, we identify an emerging contention between network and storage I/O within DCA ways, driven by the increasing storage-I/O bandwidth.

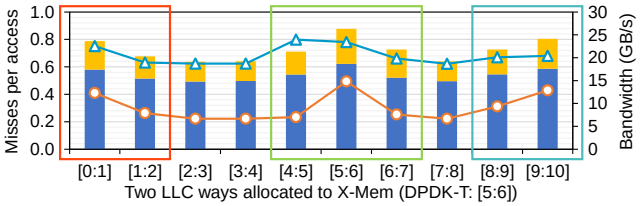
3.1 Hidden Directory Contention

Setup. We use the following two DPDK-based microbenchmarks as network-I/O workloads: (1) DPDK-T and (2) DPDK-NT. DPDK-T **T**ouches and then drops received network packets (*e.g.*, deep packet inspection [12]). DPDK-NT does **N**ot **T**ouch network packets and simply drops them (*e.g.*, packet classification and access control [13]). We run a given microbenchmark (*e.g.*, DPDK-T) on four CPU cores, each with a 2K-entry ring buffer (*e.g.*, totaling 8MB for storing 1KB packets), after allocating specific two consecutive LLC ways (*i.e.*, way[5:6]) to the microbenchmark. As a non-I/O workload, we run cache-sensitive X-Mem [41] with a 4MB working set on two CPU cores after allocating two consecutive LLC ways to X-Mem. The working set of X-Mem is larger than the aggregate capacity of two MLCs coupled with the two CPU cores but smaller than that of two LLC ways that we will allocate to X-Mem. See §6 for our system setup and control/monitor methods.

Contentions between I/O and non-I/O workloads. Fig. 3a shows the latent contention between DPDK-NT and X-Mem within DCA ways, where we sweep the allocation of two consecutive LLC ways to X-Mem from the two leftmost LLC ways (*i.e.*, way[0:1] or DCA ways) to the two rightmost LLC ways (*i.e.*, way[9:10] or inclusive ways), using Intel CAT [23]. This shows that X-Mem suffers higher LLC miss rates (also evidenced by higher consumption of memory bandwidth), when allocated to way[0:1] and way[1:2], which are fully and partially overlapped with DCA ways (red box in Fig. 3a), respectively. This contention is caused by packets DMA-written to cache lines (referred to as network-I/O cache lines hereafter) that evict the cache lines of X-Mem in DCA ways. Meanwhile, X-mem does not experience notably higher LLC miss rates when allocated to way[5:6] which DPDK-NT is also allocated to. This is because DPDK-NT does not bring I/O cache lines from DCA ways into its four MLCs since it does not consume them, preventing its working set from bloating beyond the capacity of four MLCs; in other word, DPDK-NT does not evict I/O cache lines to way[5:6].



(a) DPDK-NT (w/o touching I/O data) vs. X-Mem



(b) DPDK-T (w/ touching I/O data) vs. X-Mem

Figure 3: Contention between I/O-intensive DPDK and cache-sensitive X-Mem allocated to LLC way[*m*:*n*].

In comparison, Fig. 3b shows that X-Mem suffers higher LLC miss rates at three distinctive groups of two consecutive LLC ways allocated to X-Mem. The first group (red box in Fig. 3b) is caused by latent contention between DPDK-T and X-Mem within DCA ways, following the same contention mechanism as DPDK-NT (red box in Fig. 3a). The second group is incurred by contention between X-Mem and DPDK-T, both of which are (fully or partially) allocated to way[5:6] (green box in Fig. 3b). The source of this contention is DMA bloat caused by DPDK-T bringing fresh I/O cache lines from DCA ways into its four MLCs and subsequently evicting stale (or consumed) I/O cache lines to way[5:6]. This contrasts with DPDK-NT, which does not cause DMA-bloat to way[5:6] (dotted green box in Fig. 3a). Meanwhile, the third group (blue box in Fig. 3b) cannot be explained by the previously known contentions within the LLC.

Since the contention depicted by the third group is not observed when DPDK-NT co-runs with X-Mem (dotted blue box in Fig. 3a), it must be caused by I/O cache lines brought into MLCs and then consumed by DPDK-T. Nonetheless, these cache lines should have been

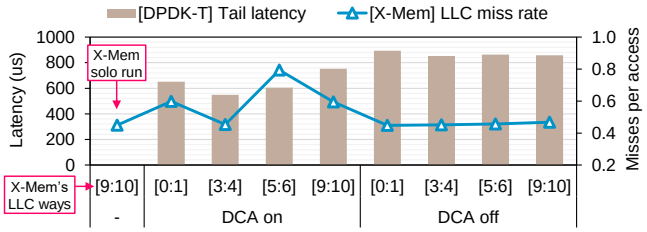
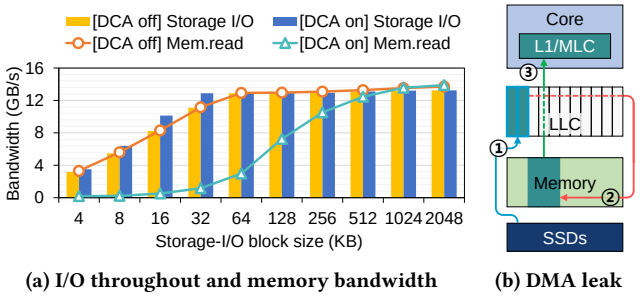


Figure 4: Validating the directory contention with DCA on.



(a) I/O throughout and memory bandwidth **(b) DMA leak**

Figure 5: Impact of storage block size and DCA on storage-I/O throughput, memory bandwidth, and DMA leak.

evicted only to way[5:6] allocated to DPDK-T, not to way[9:10]. We delve into this unexpected contention, inspired by the recent discovery of a microarchitectural property of a non-inclusive cache architecture with an inclusive directory (§2.1). Based on this, we hypothesize that LLC-exclusive cache lines in DCA ways are migrated to inclusive ways once they are read into MLCs. This is supported by past work [60] showing that the cache coherence state of DMA-written cache lines changes from *modified* LLC-exclusive to *shared* LLC-inclusive when these cache lines are brought into MLCs and consumed by CPU cores.

To validate this, we first disable DCA, forcing the CPU cores running DPDK-T to get all I/O cache lines through the device-memory-MLC path (§2.2). Second, we allocate X-Mem to way[0:1] (DCA ways), way[3:4] (standard ways), way[5:6] (ways allocated to DPDK-T), and way[9:10] (inclusive ways), respectively. Fig. 4 shows that disabling DCA avoids the contention between X-Mem and DPDK-T within inclusive ways because it does not bring I/O cache lines into MLCs from DCA ways. Note that disabling DCA also eschews the latent contention at DCA ways, but it considerably increases the p99 latency of DPDK-T (§3.2). Based on this analysis, we make the following **O**bservation:

(O1) When read by CPU cores, DMA-written I/O cache lines in DCA ways are migrated to inclusive ways, where I/O and non-I/O workloads contend.

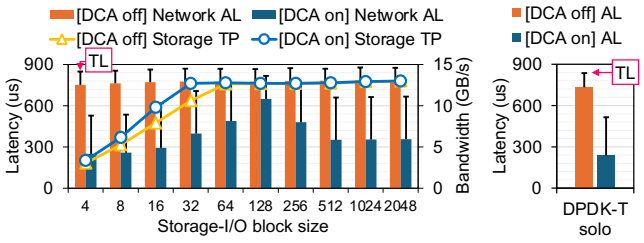
3.2 Storage-IO-Driven DCA Contention

Setup. We use Flexible I/O Tester (FIO) [30] as a storage-I/O workload, with four libaio (asynchronous I/O) threads. Each libaio thread performs random read accesses to local storage-I/O devices, with the `O_DIRECT` flag on (Direct I/O) and an I/O depth of 32. They run on four CPU cores while allocated to way[2:3]. We modify FIO such that each thread also performs regular expression matching [10] on storage blocks to make sure these blocks are brought into MLCs of the four CPU cores. Note that many datacenter servers deploy network-connected disaggregated storage servers but they still rely on local storage-I/O devices for fast access to performance-sensitive metadata or intermediate values [17, 35].

Storage-I/O characteristics. Fig. 5 shows the storage-I/O throughput and corresponding memory bandwidth consumption when DCA is enabled and disabled. This demonstrates two important characteristics of storage I/O when the size of storage blocks is larger than 32KB. First, the storage-I/O throughput is little affected by whether DCA is enabled or disabled. Second, although DCA is enabled, storage I/O consumes a considerable amount of memory bandwidth due to the DMA leak described below. As storage-I/O devices DMA-write large blocks to DCA ways at high rates (① in Fig. 5b), I/O cache lines, which stores previously DMA-written blocks, are evicted from these DCA ways before they are consumed by CPU cores (②). Later, these I/O cache lines must be brought back from memory into MLCs (③). We also observe that storage-I/O characteristics with employing deep I/O queues are similar to those with using large storage blocks. Note that while we may mitigate the DMA leak caused by storage I/O by choosing smaller blocks and/or shallower queues, such an approach is not desirable for high-throughput storage-I/O applications [7, 38, 46, 57].

Lastly, the effectiveness of DCA heavily depends on the I/O workload's temporal locality (*i.e.*, the delay between I/O devices DMA-writing I/O data and CPU cores accessing/processing them). Meanwhile, our experiment uncovers that storage-I/O-intensive workloads exhibit poor temporal locality for the following reasons. Storage-I/O workloads often transfer one to three orders of magnitude more data per I/O transaction than network-I/O workloads. This results in a longer transfer time per I/O transaction while CPU cores can start processing the data only after the entire I/O block has transferred to DCA ways. Additionally, with more complex processing for larger amounts of data, it takes longer time for CPU cores to process the data, and subsequent DMA writes likely evict cache lines storing the data before all of it is accessed and processed (*i.e.*, DMA leak). Since our experiment with FIO assumes an optimistic scenario with minimal processing per block, other workloads may not benefit from DCA even at smaller block sizes.

Contention between I/O workloads. Noting that high-throughput storage I/O alone can incur a substantial amount of DMA leak, we hypothesize significant contention between storage- and network-I/O cache lines within DCA ways, as both can simultaneously DMA-write a large number of storage blocks and network packets to these DCA ways. To demonstrate such contention, we co-run FIO with various storage block sizes and DPDK-T with a 2K-entry ring buffer per CPU core, and allocate them to way[2:3] and way[4:5], respectively. Fig. 6 shows that co-running DPDK-T with FIO increases the average latency of DPDK-T by 5–175%, compared to running DPDK-T alone, when DCA is enabled. As the storage block size increases, the amount of DMA leak—and consequently, the latency of DPDK-T—also increases, peaking at a storage block size of 128KB, where storage-I/O throughput saturates. However, as the storage block size increases over 128KB, the latency of DPDK-T begins to decrease for the following reason. In fact, we observe that the DMA leak occurs not only from DCA ways but also from inclusive ways, where migrated I/O cache lines are write-updated by subsequent DMA write (§3.1) but evicted before consumed. As



(a) DPDK-T + FIO (b) DPDK-T

Figure 6: Impact of FIO on DPDK-T latency. 'AL', 'TL', and 'TP' denote average latency, tail (p99) latency, and throughput.

the storage block size increases over 128KB, DMA leak begins to decrease, because most storage-I/O cache lines are evicted from DCA ways far before they are consumed and migrated to inclusive ways. This makes most storage-I/O data brought into MLCs directly from the memory (*i.e.*, no migration of storage-I/O cache lines to inclusive ways), which allows network-I/O data to use most of the capacity of inclusive ways. To avoid the contention between FIO and DPDK-T within DCA and inclusive ways, we may choose to disable DCA, which does not affect the throughput of FIO. However, doing so unacceptably increases the latency of DPDK-T (Fig. 6).

(O2) Storage I/O can cause intense contention between storage-I/O and network-I/O cache lines within both DCA and inclusive ways, unacceptably increasing the latency of network I/O.

4 Strategies to Mitigate LLC Contentions

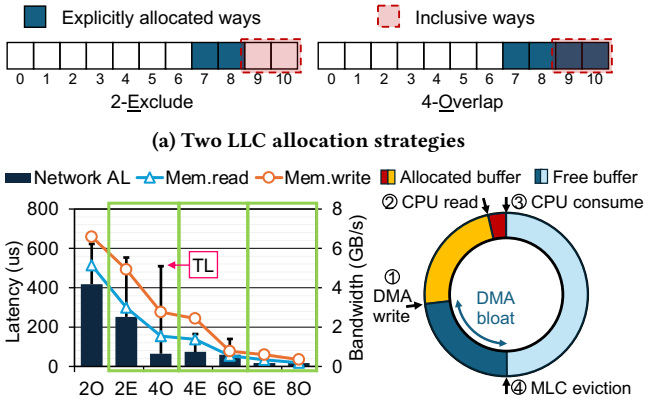
In this section, we present strategies to mitigate the newly discovered LLC contentions, using both well-known and hidden knobs provided by modern server CPUs.

4.1 Directory Contention-aware LLC Allocation

We have shown that I/O workloads consuming I/O cache lines cause these cache lines to migrate to inclusive ways, evicting cache lines of workloads explicitly allocated to those ways, *i.e.*, directory contention (§3.1). We can avoid this contention by not allocating any workloads to inclusive ways using CAT, but at the cost of reducing the number of LLC ways available for workloads by almost 20% (2 out of 11 ways). To address the inefficiency of such a naïve LLC allocation strategy, we propose explicitly allocating I/O workloads to LLC ways that encompass (or overlap) inclusive ways.

Consider two LLC allocation strategies, `n-Exclude` and `n-Overlap`, where DPDK-T is explicitly allocated to `n` LLC ways Excluding and Overlapping inclusive ways, respectively. `n-Exclude` intends to completely avoid the directory contention with the naïve strategy, while `n-Overlap` aims to maximize the LLC-caching efficiency. For instance, Fig. 7a illustrates allocated ways of `2-Exclude` and `4-Overlap`. Both `2-Exclude` and `4-Overlap` effectively use the same number of LLC ways, since DPDK-T with `2-Exclude` caches migrated I/O cache lines in inclusive ways.

Fig. 7b shows that `(n+2)-Overlap` consumes less memory bandwidth (*i.e.*, fewer LLC misses) and presents lower DPDK-T latency than `n-Exclude` although both implicitly use the same number of LLC ways (green boxes) for two reasons. First, for example,



(a) Two LLC allocation strategies

(b) Overlapping/Excluding inclusive ways (c) Life cycle of network Rx ring buffer

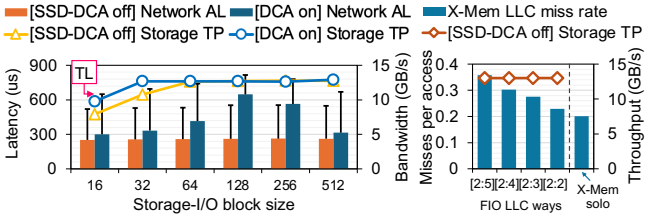
Figure 7: Impact of LLC allocation strategy on DPDK-T latency. 'AL', and 'TL' denote average and tail (p99) latency.

as `4-Overlap` and `2-Exclude` DMA-bloat 50% and 100% of consumed I/O cache lines back to way[7:8], respectively, `4-Overlap` incurs fewer conflict misses within way[7:8], thereby consuming less memory bandwidth. Second, another 50% of these I/O cache lines within the inclusive ways are soon reused/write-updated within those ways, which can provide these I/O cache lines for CPU cores more efficiently than I/O cache lines from other LLC ways for the following reason. Generally, `n-Overlap` presents three types of I/O cache lines in inclusive ways (Fig. 7c): ❶ DMA-bloated (subset of dark blue), ❷ write-updated (subset of yellow), and ❸ migrated I/O cache lines (consumed ones and freed-after-consumed ones in subset of red and light blue). Considering the life cycle of the network Rx ring buffer experiencing DMA bloat (Fig. 7c): ① DMA write, ② CPU read, ③ CPU consumption, and ③ MLC eviction, we note that ❶ is likely write-updated by DMA write and then becomes ❷. In contrast, ❸ is migrated to inclusive ways after fresh I/O cache lines are write-allocated to DCA ways or DMA-bloated I/O caches lines write-updated within standard ways, both of which go through longer paths than ❶ before CPU read. Therefore, as the percentage of ❶ in inclusive ways is higher with `n-Overlap`, CPU cores can get I/O cache lines faster with fewer conflict misses within DCA ways. We also observe analogous trends in LLC allocation strategy encompassing DCA ways.

(O3) The allocation of inclusive ways (and DCA ways) only to I/O workloads eases directory contention while maximizing LLC-caching efficiency.

4.2 I/O Device-aware DCA and LLC Allocation

To address storage-I/O-driven contention within DCA ways, we may disable DCA since DCA does not affect storage-I/O throughput (Fig. 5a) while only causing contention between storage- and network-I/O cache lines within DCA ways (Fig. 6a). A well-known knob to disable DCA is through a BIOS option [22]. However, this option disables DCA for all I/O devices including network-I/O devices, which significantly increases the p99 of network-I/O workloads compared to enabling DCA (Fig. 6b).



(a) DPDK-T + FIO (b) FIO + X-Mem

Figure 8: Impact of I/O device-aware DCA disabling and LLC allocation on DPDK-T latency, FIO throughput, X-Mem LLC miss rate, where X-Mem is allocated to way[2:5].

Tackling this challenge, we exploit a hidden feature that allows us to selectively disable DCA for a specific I/O device at runtime. Specifically, we can disable DCA, setting `NoSnoopOpWrEn` and unsetting `Use_Allocating_Flow_Wr` in register `perfctrlsts_0` for each PCIe port [26]. Using this feature, we propose to disable DCA only for storage-I/O devices. Fig. 8a plots the DPDK-T latency and FIO throughput when DCA for network-I/O devices is enabled but DCA for storage-I/O devices is disabled (`[SSD-DCA off]`), demonstrating the effectiveness of `[SSD-DCA off]` compared to when DCA is enabled for both network-I/O and storage-I/O devices (`[DCA on]`). For storage blocks larger than 32KB, `[SSD-DCA off]` offers 17–60% and 18–29% lower average and p99 latency for DPDK-T than `[DCA on]`, respectively, while providing uncompromised throughput for FIO. That is, `[SSD-DCA off]` practically eliminates the latency increase of DPDK-T caused by co-running FIO, compared to running DPDK-T alone, providing only 3% and 9% higher average and p99 latency, respectively.

However, with `[SSD-DCA off]`, FIO DMA-bloats consumed storage-I/O cache lines back to other LLC ways, incurring contention within those ways. To address such contention, we propose allocating as few standard ways as possible to storage-I/O workloads for the following reasons. The size of storage-I/O buffers often exceeds 10MB, which is far larger than that of network-I/O buffers and comparable to LLC. Therefore, consumed storage-I/O cache lines DMA-bloated back to standard ways cannot benefit from LLC caching as they will be evicted to the memory before they are reused/write-updated, unlike network-I/O cache lines. Fig. 8b plots the LLC miss rate of X-Mem co-running with FIO, where X-Mem is allocated to way[2:5] and FIO is allocated to way[2:n], with n gradually reduced from 5 to 2. This shows that as n decreases from 5 to 2, the LLC miss rate of X-Mem decreases from 36% to 23% due to fewer standard ways overlapped between FIO and X-Mem, while the throughput of FIO remains almost constant.

So far, we have shown that storage-I/O workloads heavily pollute all LLC way by DMA-write and DMA bloat. Furthermore, disabling DCA and restricting the LLC ways of such workload effectively addresses the pollution. The effect on the storage-I/O workload is negligible as large block data have poor locality.

(O4) Disabling DCA for storage-I/O devices eliminates DCA contention. **(O5)** Allocating only one standard way to storage-I/O workloads eases DMA bloat, without compromising storage-I/O throughput.

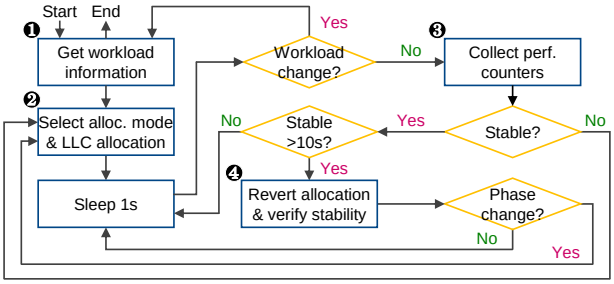


Figure 9: *A4* execution flow.

5 A4: Share More, Interfere Less

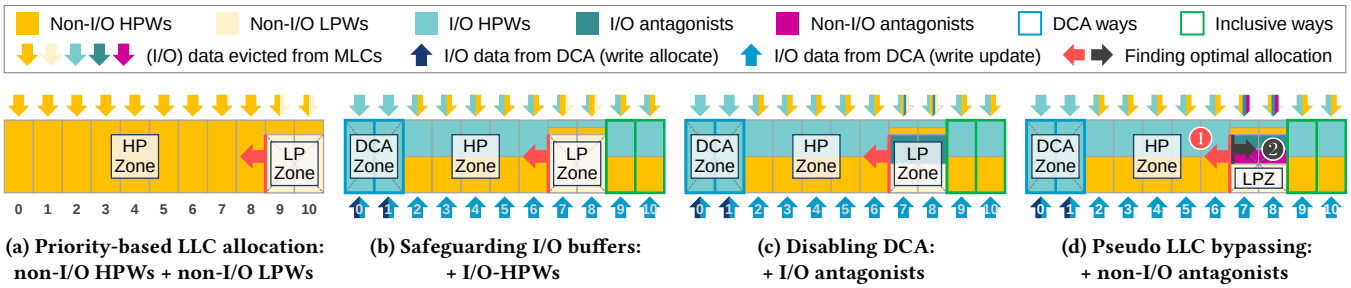
5.1 Overview

Leveraging the observations made thus far, we introduce *A4*, a holistic LLC management framework that orchestrates LLC resources among diverse co-running workloads. Determining the optimal LLC allocation for individual workloads is not trivial, as their data access patterns and working set sizes heavily depend on the dynamic behavior of workloads and systems, such as I/O stack characteristics and OS paging. Furthermore, datacenter servers, typically equipped with more CPU cores than available LLC ways, enforce the sharing of LLC ways among multiple workloads, further complicating this allocation process. To address these challenges, rather than statically allocating LLC ways to individual workloads, *A4* empirically and iteratively finds optimal LLC allocation by monitoring their relevant hardware performance counters.

Fig. 9 shows the execution flow of *A4*. When a workload is launched or terminated, *A4* gathers workload information, like QoS requirements (*i.e.*, priorities) and associated I/O devices (❶). This information can be provided by users or cluster management software [55, 67]. Given the common practice in modern datacenters to co-locate workloads with varying priorities on the same server for better resource utilization [20, 40], *A4* finds optimal LLC allocations for workloads (❷), considering (1) priority levels (§5.2), (2) contention between I/O and non-I/O workloads (§5.3), (3) contention between network-I/O and storage-I/O workloads (§5.4), and (4) other forms of contentions (§5.5). Subsequently, *A4* periodically monitors hardware performance counters to identify any antagonistic behavior or changes in workload execution phases (❸, §5.6). For other phase changes that are harder to detect, if *A4* remains stable for 10 seconds, it reverts temporarily to the initial allocation to check if workloads are still in the same phase (❹, §5.6). After ❸ and ❹, *A4* initiates LLC reallocation if necessary. Guidelines for tuning *A4* parameters are provided in §5.7.

5.2 Priority-Based LLC Allocation

A4 starts with allocating LLC ways based on the priorities of individual workloads (Fig. 10a): HP Zone shared for HPWs and LP Zone shared for LPWs. Without active I/O workloads, *A4* allocates all available LLC ways (way[0:10]) to HP Zone, while initially confining LP Zone to two LLC ways (way[9:10]), which we call *initial partitions*. Then, *A4* expands LP Zone one LLC way at a time towards the left side (red arrow in Fig. 10a), aiming to prevent overprovisioning of the LLC resources to HPWs beyond their working set sizes while making the best effort for LPWs. To achieve this, *A4*



(a) Priority-based LLC allocation: non-I/O HPWs + non-I/O LPWs **(b) Safeguarding I/O buffers: + I/O-HPWs** **(c) Disabling DCA: + I/O antagonists** **(d) Pseudo LLC bypassing: + non-I/O antagonists**

Figure 10: *A4* operations with different workload combinations. We assume that, from left to right, different types of workloads are launched (+) one by one. Colored rectangles indicate LLC ways allocated to individual workload types (LPZ: LP Zone).

monitors the LLC hit rates of individual HPWs every second and expands LP Zone every 2 seconds based on stable post-expansion values unless any HPWs exhibit a decrease in LLC hit rates surpassing the predefined threshold (`HPW_LLC_HIT_THR`). Instead of LLC hit rates, alternative performance-related metrics, such as Instructions Per Cycle (IPC), can also be used based on specific Service-Level Objectives (SLOs).

5.3 Safeguarding I/O Buffers

Once I/O HPWs are launched (Fig. 10b), *A4* intelligently rearranges LLC Zones to prevent contentions between I/O and non-I/O workloads. Specifically, it exclusively reserves DCA ways (way[0:1]) as DCA Zone, allowing only I/O HPWs to allocate these LLC ways, to eliminate latent contention while maximizing LLC-caching efficiency for I/O buffers (O3 in §4.1). Meanwhile, it prevents LP Zone from allocating inclusive ways (way[9:10]) as well, since the cumulative eviction traffic generated by LPWs could incur significant directory contention (O1 in §3.1). That is, *A4* re-evaluates the optimal LLC allocation for LP Zone (as detailed in §5.2), starting with new initial partitions that designate way[7:8] for LP Zone and way[2:10] for HP Zone, while reserving way[0:1] for DCA Zone.

Note that *A4* still allocates inclusive ways as a part of HP Zone for three reasons. First, HPWs utilize 9 LLC ways (way[2:10]), so their eviction traffic toward inclusive ways tends to be less intensive than that of LPWs which utilize fewer LLC ways and involve more antagonistic workloads (§5.5). Second, non-I/O HPWs can effectively utilize the remaining space in inclusive ways, preventing potential waste. Inclusive ways are less crowded compared to DCA ways, as a subset of the I/O traffic is served directly by LLC (*i.e.*, migrating to inclusive ways) while the rest is evicted to the memory and subsequently read through MLCs (*i.e.*, DMA leak). Third, even though some I/O data are evicted from inclusive ways by HPWs, most of them are already consumed and far from being reused, and thus, less critical than those in DCA ways.

5.4 Disabling DCA to Prevent DMA Leak

Intensive storage I/O can cause significant contention between storage-I/O and network-I/O cache lines within both DCA and inclusive ways (O2 in §3.2). Based on this observation, after launching storage-I/O workloads, *A4* monitors their I/O behavior and, upon detecting storage-I/O-driven contention, temporarily disables DCA for SSDs to mitigate DMA leak (O4 in §4.2). Furthermore, such a storage-I/O workload is treated as an LPW even if it was initially set to HPW, to mitigate pollution in HP Zone caused by heavy MLC-eviction traffic generated by storage-I/O data (*i.e.*, DMA bloat). At this point, LP Zone is reallocated (as detailed in §5.2), now including DCA-disabled storage-I/O workloads.

A4 dynamically detects storage-I/O-driven contention by monitoring: (1) frequent eviction of I/O cache lines from the LLC, *i.e.*, DCA miss rate surpasses `DMALK_DCA_MS_THR`; (2) significant DMA leak, *i.e.*, average LLC miss rate of storage-I/O workloads surpasses `DMALK_LLC_MS_THR`; and (3) substantial contribution of storage I/O to the DMA leak, *i.e.*, storage-I/O portion in overall system I/O's read throughput (*aka* PCIe write throughput) exceeds `DMALK_IO_TP_THR`. These collectively suggest storage I/O is causing considerable DMA leak without deriving any benefits from DCA. Therefore, we can safely decide to disable DCA.

5.5 Pseudo LLC Bypassing

DCA-disabled storage-I/O workloads extensively read/write low-locality (I/O) data from/to the memory and cause contention within LP Zone. Similar behavior is observed in memory-intensive, non-I/O workloads accessing data with poor temporal locality, such as KSM [53] and zswap [61] that are widely used for system resource management. To alleviate these contentions, we introduce pseudo LLC bypassing (Fig. 10d), which manages both DMA-bloated and cache-unfriendly cache lines together by allocating as few standard ways as possible to such workloads (O5 in §4.2). After LP Zone (LPZ) settles (❶), *A4* monitors the MLC and LLC miss rates of each non-I/O workload. If both metrics exceed `ANT_CACHE_MISS_THR`, *A4* speculates that this workload is an antagonist that derives minimal benefit from LLC caching. Such a non-I/O antagonist is also treated as LPW. For all identified I/O and non-I/O antagonists, *A4* progressively reduces the allocated LLC ways (*i.e.*, trash ways) from the current LP Zone down to the rightmost standard way (way[8], ❷), directing (DMA-bloated) dead cache lines to a subset of the LLC ways allocated to LP Zone. To prevent drastic performance drop, memory bandwidth abuse, or incorrect antagonist detection, this process ceases if any of the following exhibit instability (*e.g.*, fluctuations greater than 10%) after LLC way reductions: (1) LLC miss rates of non-I/O antagonists, (2) I/O throughput of storage-I/O antagonists, or (3) system-wide memory bandwidth.

Note that changing LLC way affinity using CAT only affects newly allocated LLC lines. Therefore, we can alleviate the LLC contention by selectively applying pseudo LLC bypassing to a workload

only when it exhibits antagonistic behavior, while still maintaining its performance-critical data in excluded LLC ways.

5.6 Reacting to Execution Phase Changes

Once the LLC allocation is complete (§5.2, §5.3, §5.4, §5.5), *A4* performs continuous, per-second monitoring of each workload's key performance metrics to cope with any detected changes in workloads and execution phases, as follows.

Reallocating LP Zone. *A4* reallocates LP Zone under the following three conditions: (1) new HPW combinations: occurs at the launch or termination of HPWs, or active workload's transition between HPW and LPW states (§5.4, §5.5); (2) execution phase changes: triggered if any HPW shows a fluctuation in LLC hit rate exceeding `HPW_LLC_HIT_THR` compared to that recorded in the initial partitions; and (3) uncapturable changes: activated if any HPW exhibits a fluctuation in LLC hit rate exceeding `HPW_LLC_HIT_THR` compared to the highest attainable hit rate at the moment, estimated by temporarily reverting to the initial allocation at every 10 seconds in a stable state.

Re-assigning priorities. *A4* detects antagonistic workloads either after their launch (for HPWs) or when they reach a stable state (for both HPWs and LPWs). However, their behavior may vary over time, such as when antagonistic data accesses complete. To adapt to these changes, *A4* restores LLC allocation for non-I/O antagonists if fluctuations appear in their LLC miss rate compared to that observed during initial antagonistic detection. For those originally classified as LPWs, the LLC allocation is reverted to the current LP Zone. Otherwise, they return to the HPW pool, triggering priority-based LLC reallocation. For storage-I/O antagonists, a significant fluctuation in storage-I/O throughput signals major phase changes, prompting a restoration of LLC allocations based on initial QoS requirements and reactivating DCA for the SSDs.

5.7 Guidelines for Tuning A4 Parameters

A4 employs five threshold values (T1—T5 in Table 1), and two timing parameters. Fine-tuning these values is crucial for maximizing the efficacy of *A4*, aligning with the needs of users and service providers. Below, we offer guidelines for setting these values.

T1. This threshold determines the extent to which LP Zone can expand, balancing it against the effective capacity available to HP Zone. With a lower threshold, HPWs could benefit from lower LLC miss rates, whereas a higher threshold allows LPWs to utilize more LLC resources. Note that *A4* can be readily extended in coordination with existing system monitoring tools [42, 48, 66, 67] to dynamically adjust this threshold according to specific SLOs. For instance, it can begin with the lowest value and then gradually increase the threshold as long as HPWs' specific SLOs remain satisfied.

T2–T4. T2 and T3 are used together to detect whether I/O workloads suffer from DMA leak, while T4 specifically determines whether storage I/O is the source of the DMA leak. These threshold values should be configured considering the I/O devices attached to the system, their relative bandwidth, and the extent to which the DMA leak impacts the workloads utilizing these devices. With lower thresholds, storage-I/O workload may be mistakenly classified as an antagonist. However, *A4* provides a correction mechanism for misclassifications, allowing these parameters to be set aggressively.

Table 1: *A4* evaluation setup.

Server machine	
OS (kernel)	Ubuntu-20.04.6 LTS (5.15.0-91-generic)
CPU	Intel® Xeon® Gold 6140 CPU @2.30 GHz, 18 cores, 32KiB private L1 I/D$, 1MiB private L2$, 25MiB shared LLC (11 ways, non-inclusive)
Main memory	16GB, 32GB DDR4 DRAM per channel (Total: 288GB, 6 channels)
Network device Storage device	100Gbps Nvidia ConnextX-6 NIC (part of the BlueField-2 SoC) 4× Samsung 980 PRO 1TB M.2 SSDs (4TB, PCIe Gen 3 ×16)
Client machine (network packet generator)	
OS (kernel)	Ubuntu-20.04.5 LTS (5.15.0-78-generic)
CPU	Intel® Xeon® E5-2650 v4 @2.20GHz, 12 cores, 30MiB LLC (20 ways)
Main memory	16GB DDR4 DRAM (Total: 16GB, 1 channel)
Network device	100Gbps Nvidia ConnextX-6 NIC (part of the BlueField-2 SoC)
***A4* threshold values**	
1. `HWP_LLC_HIT_THR`: 20%, 2. `DMALK_DCA_MS_THR`: 40%, 3. `DMALK_IO_TP_THR`: 35% 4. `DMALK_LLC_MS_THR`: 40%, 5. `ANT_CACHE_MISS_THR`: 90%	

Table 2: Real-world workloads for evaluation.

Name	Description	Parameters
Fastclick [6]	Network I/O workload Simple packet processing	1024-B packets, 2048-entry ring buffer per CPU core, 4 CPU cores
FFSB-H (heavy) and FFSB-L (light) [19]	Storage I/O workload Regular expression matching	Heavy: 2MB I/O block, 3 CPU cores Light: 32KB I/O block, 1 CPU core
Redis-S (server) and Redis-C (client) [49]	In-memory database workload Persistent key-value store	YCSB workload A (Update heavy), 1 CPU core each [9]
SPEC CPU2017 [8]	General-purpose workloads	SPECrates with reference input sets, 1 CPU core each

Table 3: X-Mem instances for microbenchmark evaluation.

Instance	Working set size (MB)	Access pattern	Operation type
X-Mem 1	4MB	Sequential	Read
X-Mem 2	4MB	Sequential	Write
X-Mem 3	10MB	Random	Read

T5. This threshold is used to identify potential non-I/O antagonists. The threshold should be set sufficiently high to ensure the workload gains no benefit from using the LLC. A low threshold may force the workload to sacrifice its performance for others.

Timing parameters. *A4* reverts to the initial partition for 1 second (revert interval) at every 10 seconds (stable interval) of a stable state (§5.6). Tuning a stable interval has a trade-off between responsiveness to an uncapturable phase change and performance. A short stable interval will result in frequent revert to the initial partition which yields lower performance gain. *A4* measures hit rates for a sufficiently long revert interval. Averaging the counter values for a long time ensures that *A4* to obtain stable values from workloads that frequently, and repeatedly change their phases. This guarantees that *A4* avoids frequent searches for the optimal allocation, except when an actual phase change occurs.

6 Experimental Methodology

Testbeds. Table 1 outlines our testbeds. The server machine serves as our focal system for LLC orchestration, running all the (micro)benchmarks along with a *A4* daemon. The client machine generates and sends network packets to the server by running DPDK Pktgen [15, 16]. Two machines are connected via a 100Gbps network interface, driven by Nvidia ConnextX-6 NICs. We verified that Pktgen generates sufficient network traffic to stress the server, reaching up to 100Gbps. The server's local storage consists of a RAID-0 array of four 1TB SSDs plugged into a PCIe Gen3 ×16 slot. Hyper-Threading and Turbo Boost are disabled on both machines.

System monitor/control. Hardware performance counters, such as LLC/DCA hits/misses and I/O throughputs, are monitored using Intel Performance Counter Monitor (PCM) [25]. We use Intel CAT [23] to allocate LLC way(s) to individual CPU cores. Note that CAT only allows contiguous LLC way allocation for each CPU core. *A4*'s fundamental operations (system monitor and cache control) closely resemble those in [67], which was confirmed to add negligible overhead to the system even with context switches for accessing hardware performance counters. We also verify that the execution time of *A4* does not exceed 800μs, imposing minimal overhead on the system given the 1-second monitoring interval.

Benchmarks. We evaluate *A4* using both microbenchmarks (§7.1) and real-world workloads (§7.2). For microbenchmarks, we use the same workloads described in §3 but with varying configurations to offer a comprehensive analysis of *A4*'s efficacy. The real-world workloads are described in Table 2. Workloads are pinned to specific CPU cores, while *A4* daemon occupies one CPU core, ensuring its isolation from other CPU cores while remaining within the same NUMA node. Result values are averaged over five iterations, and each of them lasts 70s, including the initial 10s of warm-up time and the last 10s of result-collecting time.

Evaluated designs. We compare *A4* with two baseline LLC management schemes: (1) a Default model, where all workloads share the entire LLC without explicit CAT allocation; and (2) an Isolate model, which statically assigns distinct LLC ways to each workload in proportion to the number of pinning CPU cores, *i.e.*, static workload-wise LLC isolation. In both baseline models, DCA is enabled for every I/O device.

7 Results and Analysis

7.1 Microbenchmark Results

Setup. Besides DPDK-T (HPW) and FIO (LPW), each running on four CPU cores (way[2:3] and way[4:6], respectively, in the Isolate model), we run three X-Mem variants as depicted in Table 3. We conduct experiments with (1) varying network packet sizes from 64B to 1,514B with 2MB storage I/O blocks, and (2) varying storage I/O block sizes from 4KB to 2MB with 1,514B packets.

Non-I/O workload analysis. Fig. 11 depicts the IPCs and LLC hit rates of three X-Mem variants with varying network packet sizes. IPC values are normalized to those of the Default model with the smallest packet size. In the Default model, three X-Mems show degraded performance with increasing packet sizes, because larger packets pose more pressure on the LLC as the I/O buffer size increases (*i.e.*, DMA bloat). On the other hand, in the Isolate model, their performance seems to be less affected by the packet sizes since LLC ways allocated to each workload are isolated from each other. Nevertheless, this performance isolation is imperfect because it can be easily broken by the directory and latent contentions. As a result, X-Mem 1 and X-Mem 2 show performance degradation with larger packets. In addition, as it statically allocates LLC ways without considering cache sensitivity and working set sizes, a cache-sensitive X-Mem 1 occupies less effective LLC capacity than the Default model, resulting in lower performance. In *A4*, X-Mem 1 (HPW) and X-Mem 2 (LPW) maintain their initial QoS requirements, while X-Mem 3 is detected as an antagonist. As a result, X-Mem 1 achieves speedups of 1.3×–1.78× over the Default

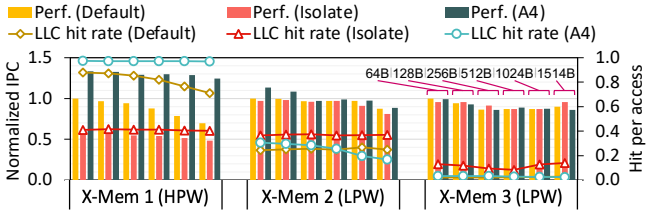


Figure 11: IPC and LLC hit rates of three X-Mem variants with varying network packet sizes (storage I/O size: 2MB).

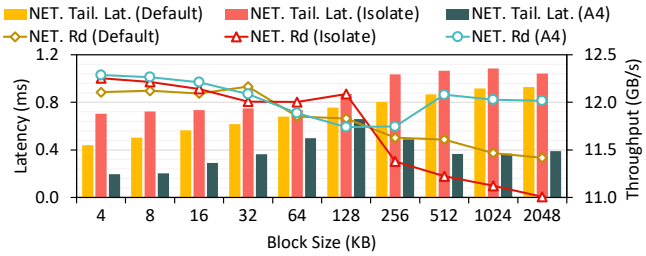


Figure 12: Network performance metrics with varying storage I/O block sizes (network packet size: 1,514B).

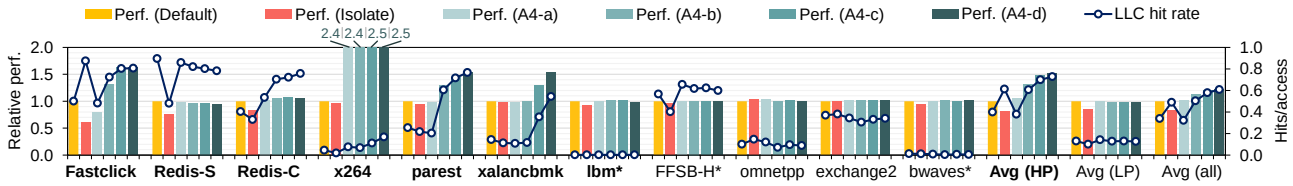
model by maintaining consistent LLC hit rates of 97% across varying packet sizes. Meanwhile, LPWs (X-Mem 2 and X-Mem 3) retain the performance within acceptable ranges.

I/O workload analysis. Fig. 12 depicts the average latency and throughput of DPDK-T with varying storage I/O block sizes. With increasing I/O block sizes, the Default and Isolate models exhibit increasing network latency and decreasing network throughput due to the exacerbated storage-I/O-driven contentions. These trends are more pronounced in the Isolate model, as network I/O can benefit from reusing DMA-bloated I/O buffers only within smaller isolated LLC ways. On the other hand, *A4* achieves a 58% reduction in network latency and a 5% increase in network throughput at the largest block size compared to the Default model. Note that *A4* experiences gradual network performance degradation until the block size reaches 128KB, as FIO is not detected as an antagonistic workload within this range.

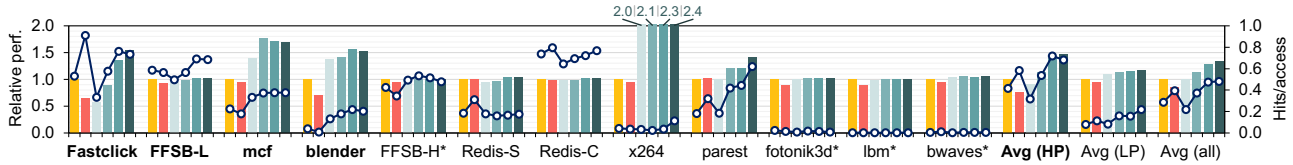
7.2 Real-World Workload Results

Besides Default and Isolate models, we evaluate four variants of *A4*, applying the techniques proposed in §5 (Fig. 10a–10d) one by one to Default model (denoted as *A4*-a–d, respectively).

HPW-heavy scenario. Fig. 13a shows the relative performance (normalized to Default model) and LLC hit rates of co-running workloads listed in Table 2, with seven workloads assigned high priority and four assigned low priority. We denote HPWs as boldface and LPWs as normal legends, and the workloads detected as antagonists are highlighted with asterisks. We observe that IPCs and LLC hit rates do not always align with actual performance in multi-threaded I/O workloads. This is because these metrics are often inflated by synchronization, I/O syscall, and idle loops [1]. For instance, LLC hit rates of Fastclick in the Isolated model tend to be inflated even though they exhibit lower throughput and higher latency compared to Default model. Thus, we measure throughput (inverse of latency per request) for multi-threaded I/O workloads (Fastclick and FFSB-H/L) and IPC for single-threaded workloads (Redis-S/C and SPEC



(a) HPW-heavy scenario: 7 HPWs + 4 LPWs (3 antagonists detected). Speedups: 52% (HPWs), 22% (All)



(b) LPW-heavy scenario: 4 HPWs + 8 LPWs (4 antagonists detected). Speedups: 47% (HPWs), 33% (All)

Figure 13: Performance and LLC hit rates of real-world workloads with various LLC orchestration schemes. Workloads with an original QoS requirement of HPW are in bold, and asterisks (*) indicate workloads experiencing pseudo LLC bypassing.

CPU). These metrics are inversely proportional to the execution time, accurately representing their performance.

In general, Isolate model yields lower performance than Default model. Isolate model confines each worklaod to specific LLC ways, irrespective of its QoS requirement and cache sensitivity. Such rigid allocation impedes flexible LLC sharing among workloads, often resulting in mismatches where their working set sizes either exceed or fall short of the assigned LLC capacity.

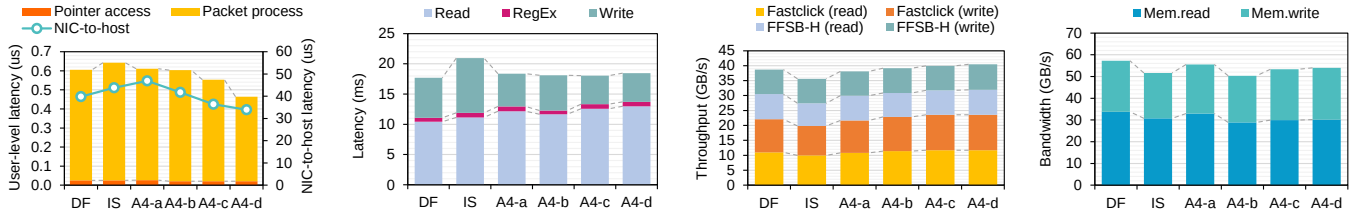
Even with LLC sharing enabled among workloads of the same priority, *A4*-a (Fig. 10a) fails to demonstrate a significant performance improvement, except for x264. This is primarily due to I/O-driven contentions, as both network-I/O and storage-I/O devices inject high-bandwidth I/O traffic into the LLC. The I/O cache lines from these workloads, distributed across all LLC ways, significantly interfere with co-running non-I/O workloads, offsetting the benefit of priority-based partitioning. Conversely, the accumulated MLC eviction traffic from LPWs to LP Zone (including inclusive ways) creates significant contention with Fastclick, leading to a noticeable performance degradation. Nevertheless, FFSB-H is less affected by these contentions as its heavy storage I/O operations hardly benefit from DCA and LLC caching. By safeguarding I/O buffers, *A4*-b (Fig. 10b) enhances the LLC hit rate and performance of Fastclick by 50% and 67%, respectively, compared to *A4*-a. Non-I/O HPWs also gain from improved LLC-caching efficiency for I/O buffers, as a substantial portion of I/O cache lines now resides in either DCA or inclusive ways, alleviating DMA bloat to standard ways. Consequently, *A4*-b demonstrates 32% (29%) performance improvements for I/O (non-I/O) HPWs and an overall 52% increase in LLC hit rates across all HPWs compared to Default model.

Even with these optimizations, Fastclick still suffers from DMA leak and bloat caused by FFSB-H. *A4*-c (Fig. 10c) intelligently disables DCA for FFSB-H by detecting its antagonistic behavior, resulting in a 11% improvement in LLC hit rate and a 24% performance increase for Fastclick compared to *A4*-b. However, this increases the eviction traffic from its MLCs to LP Zone as FFSB-H can no longer utilize DCA and inclusive ways. *A4*-d (Fig. 10d) relieves such DMA-bloated pressure on LP Zone, by directing the consumed I/O cache lines to limited trash ways. As bwaves and lbm are also identified as an antagonist due to their high MLC/LLC miss rates, *A4*-d further improves the performance and LLC hit rates of HPWs by 2% and 4% compared to *A4*-c. After mitigating latent, directory, and DCA contention, I/O HPW performance relies less on standard ways, yielding only marginal gains for *A4*-d over *A4*-c.

The sensitivity of non-I/O HPWs performance to LLC capacity is explained by the analysis of SPEC CPU2017 Suite [50]. It suggests that x264 reaches diminishing returns beyond a certain cache size, but parest and xalancbmk steadily benefit from increased cache size. This aligns with our result that the LLC hit rates of parest and xalancbmk increase progressively, unlike x264. To sum up, *A4* successfully eradicates various types of I/O-driven contentions. *A4* improves LLC hit rate and performance of all (HPW) workloads by 79% (83%) and 22% (51%), respectively, compared to the Default model, without compromising the performance of LPWs.

LPW-heavy scenario. We verify the robustness of *A4* with a different combination of workloads, more focused on LPWs, described in Fig. 13b. While the overall trend is similar to the HPW-heavy scenario, the LPW-heavy scenario demonstrates distinct trends emerging from a smaller HP Zone and a larger LP Zone. First, non-I/O HPWs performance saturates right after applying *A4*-a. Second, non-antagonistic LPWs, x264 and parest, benefit from extensive LP Zone capacity. However, their performance gains are lower than in the HPW-heavy scenario where they are set as HPWs. Note that for x264, while the LLC hit rate gradually increases, the L2 hit rate increases sharply. In addition, *A4* distinguishes antagonistic workload among multiple storage I/O workloads, and selectively disables DCA for FFSB-H only. Overall, *A4* improves LLC hit rate and performance of all (HPW) workloads by 68% (63%) and 33% (47%), respectively, compared to the Default model.

I/O latency and throughput. We use a modified Fastclick binary exclusively in Fig. 14a to measure network latency breakdown, as timestamping adds delay on the critical path. We capture three parts: queueing in the Rx ring, accessing a packet pointer, and processing it. Fig. 14a shows that *A4*-d reduces these three parts by 15%, 20%, and 23% (vs. Default model). In Fastclick, computation latency is reduced the most, indicating that I/O HPWs are well benefiting from DCA. This matches the result depicted in Fig. 14c, *i.e.*, reduced



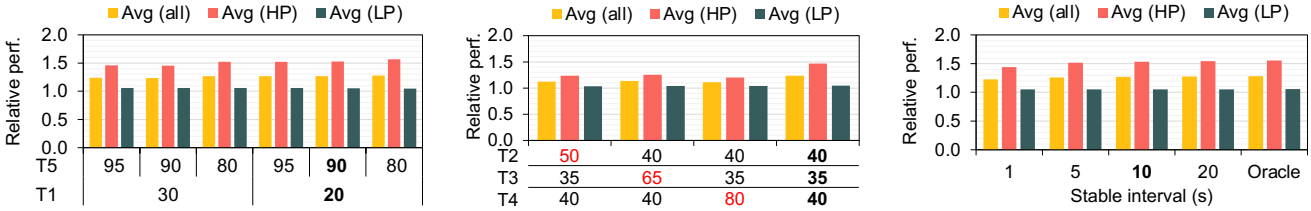
(a) Breakdown of network I/O (Fastclick) average latency

(b) Breakdown of storage I/O (FFSB-H) average latency

(c) System-wide I/O throughput (Fastclick + FFSB-H)

(d) System-wide memory bandwidth consumption

Figure 14: I/O latency breakdown and system-wide performance metrics.



(a) Partitioning thresholds

(b) Antagonist detection thresholds

(c) Timing parameter

Figure 15: Sensitivity study on *A4* parameters. Parameter values used in the main experiment are highlighted in boldface.

latency translates into increased I/O throughput in network I/O device. Since the I/O devices have reached the maximum available bandwidth, the gain in I/O throughput is less compared to that of I/O latency. On the contrary, latency and throughput are largely unchanged in FFSB-H, supporting that it's insensitive to DCA or LLC caching. The storage I/O breakdown in Fig. 14b suggests that read latency is even larger when DCA is enabled (Default model) than when DCA is disabled (*A4*-c).

Memory bandwidth. Pseudo LLC bypassing forces the dead cache lines to be written back to the memory, but the total number of write-backs remains largely unchanged (O5 in §4.2). This is depicted as a negligible difference in memory bandwidth in Fig. 14d, comparing *A4*-c and *A4*-d. Disabling DCA for intensive storage I/O workload does increase the memory write bandwidth by 9% from *A4*-b to *A4*-c. However, overall *A4*-d reduces memory bandwidth by 6% (vs. Default model). Furthermore, despite the 5% increase in the I/O throughput, *A4*-d reduces the memory read bandwidth by 11%. (vs. Default model). It is attributed to the improved LLC-caching efficiency for high locality data, and alleviating DMA leak. Note that, the low memory bandwidth of the Isolate model originates from reduced I/O throughput.

Sensitivity study. We investigate the sensitivity of thresholds and parameters by categorizing them into three groups: (1) partitioning thresholds (T1 and T5), (2) I/O antagonist detection thresholds (T2—T4), and (3) timing parameters. We perform a sensitivity study using an HPW-heavy configuration, highlighting main experimental parameters in bold. Fig. 15 illustrates relative performance of *A4* normalized to the Default model.

First, Fig. 15a depicts the impact of T1 and T5, which establish the optimal partition of *A4*. T1 defines the acceptable range of HPW performance. As T1 decreases, the *A4* constrains the LP Zone, thereby enhancing the performance of HPWs while diminishing that of LPWs. T5 is used to dynamically detect non-I/O antagonistic workloads. At T5 values of 90% and 95%, *A4* identifies two non-I/O antagonists, whereas, at that of 80%, it detects three, which leads to additional performance gain in HPWs. However, this comes at the cost of a non-I/O HPW (*i.e.*, xalancbmk), sacrificing its performance and failing to meet its QoS. Therefore, despite an overall performance increase, a T5 threshold of 80% does not appear optimal.

Second, in Fig. 15b, T2, T3, and T4 jointly indicate that the certain I/O workload is not leveraging the advantage of DCA, causing severe DMA leak. As the threshold values increase, FFSB-H is no longer identified as an antagonist, leading to suboptimal performance. By gradually increasing each value, we pinpoint the threshold values at which FFSB-H ceases to be detected as an antagonist, and these critical thresholds are marked in red in Fig. 15b.

Finally, in Fig. 15c, we intentionally chose a stable phase in our experiment to investigate the overhead of periodic reverting. *A4* periodically reverts to the initial partition, which might yield inferior performance if triggered too frequently. As shown in Fig. 15c, with a fixed revert interval of 1 second, as the stable interval increases, performance approaches the oracle policy where *A4* never reverts and stays in a stable state. In particular, 10 seconds of stable interval achieves 99.2% performance of the oracle policy.

8 Related Work

Handling DMA bloat. Existing solutions to address DMA bloat are (1) self-invalidating cache lines that invalidates consumed I/O buffers without writing them back to the LLC [2, 59] and (2) replacing LLC-unfriendly cache lines earlier based on re-reference interval prediction [29, 47, 52, 63, 64] or dead block prediction [32, 33, 37, 39]. However, these approaches either entail a high overhead of detecting dead cache lines or require significant changes across the computing stack (*e.g.* program codes, ISAs, and cache controllers). As an alternative solution, we introduce pseudo LLC bypassing, which is readily applicable to commodity servers.

Mitigating latent contention. ShRing [44] proposes reducing the aggregate memory footprint of receive rings by sharing them among CPU cores, while DMA Cache [54] suggests better isolation by separating on-chip storage for I/O data. Some studies [56, 67]

address the problem without such software or hardware modification by utilizing CAT-based cache way partitioning. Our work provides a holistic LLC orchestration framework that addresses not only these contentions but also two new I/O-driven contentions.

9 Conclusion

This work uncovers two new I/O-driven LLC contentions: between I/O and non-I/O workloads within inclusive ways and between storage-I/O and network-I/O workloads within DCA ways. Then, this work presents *A4* that orchestrates LLC sharing to holistically address these new I/O-driven LLC contentions when co-running diverse workloads, as well as previously identified ones. It improves the performance of high-priority workloads by 51% without notably compromising that of low-priority workloads.

Acknowledgments

This work was supported in part by the MSIT, Korea, under the Global Scholars Invitation Program (No. RS-2024-00456287) supervised by the IITP; by the National Research Foundation of Korea (NRF) (No. RS-2025-00554650); by the Technology Innovation Program (No. RS-2024-00420541, 2410000802) funded by the Ministry of Trade, Industry and Energy of Korea; by the Yonsei University Research Fund of 2025-22-0104; and by gift grants from Intel Corp. Ipoom Jeong is the corresponding author.

References

[1] Alaa R Alameldeen and David A Wood. 2006. IPC considered harmful for multiprocessor workloads. *IEEE Micro* 26, 4 (2006), 8–17.

[2] Mohammad Alian, Siddharth Agarwal, Jongmin Shin, Neel Patel, Yifan Yuan, Daehoon Kim, Ren Wang, and Nam Sung Kim. 2022. IDIO: Network-Driven, Inbound Network Data Orchestration on Server Processors. In *MICRO*. 480–493.

[3] 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 *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 160–169.

[4] AMD. Accessed in 2025. Smart Data Cache Injection (SDCI) White Paper . https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/white-papers/58725.pdf.

[5] Arm. Accessed in 2025. Arm DynamIQ Shared Unit Technical Reference Manual. https://developer.arm.com/documentation/100453/latest/.

[6] Tom Barbette. Accessed in 2024. tbarbette/fastclick: FastClick - A faster version of the Click Modular Router featuring batching, advanced multi-processing and improved Netmap and DPDK support (ANCS'15). Check the metron branch for Metron specificities (NSDI'18). PacketMill modifications (ASPLOS'21) as well as MiddleClick(ToN, 2021) are merged in main. https://github.com/tbarbette/fastclick.

[7] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. 2003. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*. 371–386.

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] cppreference.com. Accessed in 2024. Regular expressions library since (C++11). https://en.cppreference.com/w/cpp/regex.

[11] Crucial. Accessed in 2024. Crucial® T705 Gen5 NVMe® M.2 SSD. https://www.crucial.com/content/dam/crucial/ssd-products/t705/flyers/b2c/crucial-t705-b2c-product-flyer-en.pdf.

[12] Luca Deri. Accessed in 2024. Using nDPI over DPDK to Classify and Block Unwanted Network Traffic. https://www.dpdk.org/wp-content/uploads/sites/35/2018/12/LUCADERI_NDPI.pdf.

[13] DPDK. Accessed in 2024. 59. Packet Classification and Access Control — Data Plane Development Kit 24.03.0 documentation. https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html.

[14] DPDK. Accessed in 2024. Data Plane Development Kit (DPDK). https://www.dpdk.org.

[15] DPDK. Accessed in 2024. GitHub - pktgen/Pktgen-DPDK: DPDK based packet generator. https://github.com/pktgen/Pktgen-DPDK. (Accessed on 01/04/2023).

[16] DPDK. Accessed in 2024. The Pktgen Application — Pktgen 23.06.1 documentation. https://pktgen-dpdk.readthedocs.io/en/latest/. (Accessed on 06/24/2024).

[17] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. 2019. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere!. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 647–660.

[18] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX ATC*. 673–689.

[19] FFSB-Prime. Accessed in 2024. FFSB-Prime/ffsb: The Flexible Filesystem Benchmark (FFSB) is a cross-platform filesystem performance measurement tool. https://github.com/FFSB-Prime/ffsb.

[20] Jaeung Han, Seungheun Jeon, Young-ri Choi, and Jaehyuk Huh. 2016. Interference management for distributed parallel applications in consolidated clusters. *ACM SIGPLAN Notices* 51, 4 (2016), 443–456.

[21] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. 2005. Direct cache access for high bandwidth network I/O. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 50–59.

[22] Intel. Accessed in 2024. 11th Gen Intel® Core™ Processors Real-Time Tuning Guide. https://webdls.ieiworld.com/data/_prod-detail-feature/DRPC-DEV-KIT/Real-Time-Tuning-Guide-11th-Gen-Intel-Core-Processors-1.4.pdf.

[23] Intel. Accessed in 2024. GitHub - intel/intel-cmt-cat: User space software for Intel(R) Resource Director Technology. https://github.com/intel/intel-cmt-cat.

[24] Intel. Accessed in 2024. Intel Xeon Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[25] Intel. Accessed in 2024. intel/pcm: Intel® Performance Counter Monitor (Intel® PCM). https://github.com/intel/pcm.

[26] Intel. Accessed in 2024. Intel® Xeon® Skylake Processor Scalable Family Datasheet. https://www.intel.com/content/www/us/en/content-details/614073/intel-xeon-processor-scalable-family-datasheet-volume-two-registers.html.

[27] Intel. Accessed in 2025. Intel® Data Direct I/O Technology (Intel® DDIO): Technology Brief. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html.

[28] Ravi Iyer, Vivek De, Ramesh Illikkal, David Koufaty, Bhushan Chitlur, Andrew Herdrich, Muhammad Khellah, Fatih Hamzaoglu, and Eric Karl. 2021. Advances in microprocessor cache architectures over the last 25 years. *IEEE Micro* 41, 6 (2021), 78–88.

[29] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH computer architecture news* 38, 3 (2010), 60–71.

[30] Jens Axboe. Accessed in 2024. Flexible I/O Tester. https://github.com/axboe/fio.

[31] Ipoom Jeong, Jiaqi Lou, Yongseok Son, Yongjoo Park, Yifan Yuan, and Nam Sung Kim. 2023. LADIO: Leakage-Aware Direct I/O for I/O-Intensive Workloads. *IEEE Computer Architecture Letters* (2023).

[32] Daniel A Jiménez. 2010. Dead block replacement and bypass with a sampling predictor. In *JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship*.

[33] Samira M Khan, Daniel A Jiménez, Doug Burger, and Babak Falsafi. 2010. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 489–500.

[34] KIOXIA. Accessed in 2024. Data Center SSD | KIOXIA - Asia Pacific (English). https://apac.kioxia.com/en-apac/business/ssd/data-center-ssd.html.

[35] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding ephemeral storage for serverless analytics. In *2018 USENIX annual technical conference (USENIX ATC 18)*. 789–794.

[36] 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)*. IEEE, 20–38.

[37] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction & dead-block correlating prefetchers. *ACM SIGARCH Computer Architecture News* 29, 2 (2001), 144–154.

[38] Sunwoo Lee, Qiao Kang, Kewei Wang, Jan Balewski, Alex Sim, Ankit Agrawal, Alok Choudhary, Peter Nugent, Kesheng Wu, and Wei-keng Liao. 2021. Asynchronous I/O Strategy for Large-Scale Deep Learning Applications. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 322–331.

[39] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 222–233.

[40] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International*

Symposium on Microarchitecture. 248–259.
[41] Microsoft. Accessed in 2024. GitHub - microsoft/X-Mem: Extensible Memory Benchmarking Tool. https://github.com/microsoft/X-Mem.
[42] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. 237–250.
[43] Jiwoong Park, Cheolgi Min, and HeonYoung Yeom. 2017. A new file system I/O mode for efficient user-level caching. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 649–658.
[44] Boris Pismenny, Adam Morrison, and Dan Tsafrir. 2023. ShRing: Networking with Shared Receive Rings. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 949–968.
[45] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2022. Double trouble: Combined heterogeneous attacks on {Non-Inclusive} cache hierarchies. In *31st USENIX Security Symposium (USENIX Security 22)*. 3647–3664.
[46] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and André Brinkmann. 2024. Combining Buffered {I/O} and Direct {I/O} in Distributed File Systems. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 17–33.
[47] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News* 35, 2 (2007), 381–391.
[48] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.
[49] Redis. Accessed in 2024. Redis. https://redis.io/.
[50] Sarabjeet Singh and Manu Awasthi. 2019. Memory centric characterization and analysis of spec cpu2017 suite. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 285–292.
[51] SPDK. Accessed in 2024. Storage Performance Development Kit. https://spdk.io/.
[52] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 63–74.
[53] John Peter Stevenson. 2013. *Fine-grain in-memory deduplication for large-scale workloads*. Stanford University.
[54] 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 *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.
[55] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next generation. In *Proceedings of the fifteenth European conference on computer systems*. 1–14.
[56] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *NSDI*. 283–297.
[57] Shu-Mei Tseng, Bogdan Nicolae, Franck Cappello, and Aparna Chandramowlishwaran. 2021. Demystifying asynchronous i/o interference in hpc applications. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 391–412.
[58] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. 2022. Memory performance of AMD EPYC Rome and Intel Cascade Lake SP server processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 165–175.
[59] Marina Vemmou, Albert Cho, and Alexandros Daglis. 2022. Patching up Network Data Leaks with Sweeper. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 464–479.
[60] Minhu Wang, Mingwei Xu, and Jianping Wu. 2022. Understanding I/O Direct Cache Access Performance for End Host Networking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–37.
[61] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. 2022. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 609–621.
[62] WikiChip. Accessed in 2025. Zen 4 - Microarchitectures - AMD - WikiChip. https://en.wikichip.org/wiki/amd/microarchitectures/zen_4.
[63] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 430–441.
[64] Yuejian Xie and Gabriel H Loh. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 174–183.
[65] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy 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)*. IEEE, 888–904.
[66] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 607–618.
[67] 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 *ISCA*. IEEE, 112–125.

A Artifact Appendix

A.1 Abstract

In this section, we provide detailed instructions to reproduce the main experimental results presented in the paper. The artifact includes three primary sets of experiments: (1) experiments revealing new I/O-driven LLC contentions (Fig. 3, 4, 5, and 6); (2) experiments demonstrating the effectiveness of contention mitigation solutions (Fig. 7, and 8); and (3) experiments evaluating A4 with real-world workloads (Fig. 13, and 14).

A.2 Artifact check-list (meta-information)

- **Program:**
  - Microbenchmarks: DPDK-T, DPDK-NT, Flexible I/O Tester (FIO) [30], and X-Mem [41]
  - Real-world workloads: Fastclick [6], FFSB [19], Redis [9, 49], and SPEC CPU2017 [8]
  - Tools: Intel Performance Counter Monitor (PCM) [25], Intel CAT [23], and ddio-bench [18]
- **Compilation:** GCC-10.0.5. Python 2.7 and Scons 2.3.0
- **Binary:** DPDK-(N)T, X-Mem, and ddio-bench
- **Run-time environment:** Ubuntu 20.04 LTS
- **Hardware:** A server machine with Intel Xeon Gold 6140 CPU, and a client machine connected to 100Gbps DPDK-compatible NICs. Four M.2 SSDs and a M.2 NVMe RAID Controller.
- **Metrics:** Throughput, average and p99 latency, LLC hit rate, memory bandwidth consumption, and IPC.
- **Output:** Text files with raw data, and figures.
- **Experiments:** Fig. 3, 4, 5, 6, 7, 8, 13, and 14
- **How much disk space required (approximately)?: ≤ 20GB**
- **How much time is needed to prepare workflow (approximately)?: 1h**
- **How much time is needed to complete experiments (approximately)?: ~ 10h**
- **Publicly available?: https://github.com/ece-fast-lab/ISCA-2025-A4**
- **Code licenses (if publicly available)?: MIT License**
- **Archived (provide DOI)?: 10.5281/zenodo.15105163**

A.3 Description

A.3.1 How to access. The artifact can be accessed on Github (https://github.com/ece-fast-lab/ISCA-2025-A4). Most of the benchmarks and tools are publicly available via the following links.

- **FIO:** https://github.com/axboe/fio
- **X-Mem:** https://github.com/microsoft/X-Mem
- **Fastclick:** https://github.com/tbarbette/fastclick
- **FFSB:** https://github.com/FFSB-Prime/ffsb
- **Redis:** https://github.com/redis/redis
- **YCSB:** https://github.com/brianfrankcooper/YCSB
- **PCM:** https://github.com/intel/pcm
- **CAT:** https://github.com/intel/intel-cmt-cat
- **DDIO-bench:** https://github.com/aliireza/ddio-bench

A.3.2 Hardware dependencies. The experiment requires two machines equipped with DPDK-compatible 100 Gbps NICs connected to each other. The server machine should have I/O devices in the same socket with at least 18 cores. We conducted an evaluation on an Intel Xeon Gold 6140 CPU (Skylake) with Nvidia BlueField-2 and four Samsung 980 PRO 1TB M.2 SSDs with a RAID controller installed in the same node.

A.3.3 Software dependencies. Please set up the dependencies for benchmarks and tools according to each GitHub repository listed in §A.3.1. Python 3 is required to generate figures along with the matplotlib, numpy, and pandas libraries. SPEC CPU2017 suite is required to run real-world workload experiments.

A.4 Installation

First, clone the artifact repository.

```
$ git clone https://github.com/ece-fast-lab/ISCA-2025-A4
$ cd ISCA-2025-A4
```

Second, install benchmarks and tools. Follow the detailed instructions in `app/README.md` and `tools/README.md`. Third, set the client machine as in `client/README.md`, and corresponding environment variables in `scripts/utils/env.sh` properly.

Some benchmarks are modified, and we provided a forked repository of the official benchmark, which is also publicly available. Modifications of each benchmark are indicated in `app/README.md`. Also, we provide pre-compiled binaries of some benchmarks and tools along with the source codes, such as DPDK micromenchmarks (DPDK-(N)T), X-Mem with different core affinities, and ddio-bench.

A.5 Experiment workflow

Run the initial setup scripts every time the machines are rebooted.

```
# in server machine
$ cd scripts/utils
$ source ./env.sh
$ sudo ./init.sh
# in client machine
$ sudo ./setup_basic.sh
```

We provide guidelines on `scripts/README.md` for running experiments. `run_motivation.sh` runs figures 3 to 8, and `run_evaluation.sh` runs figures 13 and 14.

```
$ scripts/run_motivation.sh
$ scripts/run_evaluation.sh
```

A.6 Evaluation and expected results

Results generated from §A.5 demonstrate I/O-driven new contentions, suggest a mitigation technique, and evaluate *A4* on the real-world benchmarks. It improves the performance of HPWs significantly without notably compromising that of LPWs. Detailed observations and key takeaways for each experiment can be found under the corresponding folders in the repository. Since experiments are conducted on real hardware, results may vary due to several influencing factors. Specifically, X-Mem tends to exhibit inconsistent performance, which could result in trends not precisely matching those presented and necessitate repeated runs.