This technical report is an extended version of a paper with the same title published in ACM SIGCOMM 2024.

Understanding the Host Network

Midhul Vuppalapati
Cornell University

Saksham Agarwal
Cornell University

Henry N. Schuh
University of Washington

Baris Kasikci
University of Washington

Arvind Krishnamurthy
University of Washington

Rachit Agarwal
Cornell University

ABSTRACT

The host network integrates processor, memory, and peripheral interconnects to enable data transfer within the host. Several recent studies from production datacenters have established that contention within the host network can have significant impact on end-to-end application performance. The goal of this paper is to build an in-depth understanding of the various regimes and root causes of such contention within the host network.

We present domain-by-domain credit-based flow control, a conceptual abstraction to study the host network. We show that the host network performs flow control over different domains (sub-networks within the host network), each having different numbers of credits and different unloaded latency. Different applications, depending on the source and the type of data transfer (compute or peripheral, and read or write), traverse different domains; thus, different applications may observe different performance degradation when the host network is contended. Exploring the host network from the lens of domain-by-domain credit-based flow control allows us to (1) near-precisely explain contention within the host network and its impact on networked application performance observed in previous studies; and (2) discover new, previously unreported, regimes of contention within the host network.

More broadly, our study shows that contention within the host network is not merely due to limited host network resources but rather due to the poor interplay between processor, memory, and peripheral interconnects within the host network. Moreover, contention within the host network has implications that are more far-reaching than the context of networked applications considered in previous studies: all our observations hold even when all applications are contained within a single host. Our study thus opens up many interesting avenues of future research at the intersection of computer networking, operating systems and computer architecture.

1 INTRODUCTION

In conversations on "networks", our community usually engages in discussions on the Internet, datacenter networks, mobile networks, etc. This paper is about a different network—*the host network*—that integrates processor, memory, and peripheral interconnects to enable data transfer between devices (processors, memory, network interface cards, storage devices, accelerators, etc.) within a host. Several studies from large-scale production datacenters [1, 42, 44] have demonstrated that contention within the host network can result in significant throughput degradation, tail latency inflation, and isolation violation for networked applications. As eloquently argued in [1, 2, 10, 42], the host network is becoming an increasingly prominent bottleneck due to unfavorable technology trends: performance of peripheral interconnects is improving much more rapidly than processor and memory interconnects, resulting in increasing imbalance of resources and contention within the host network. Designing future network protocols, operating systems, and hardware requires an in-depth understanding of the various regimes and root causes of such contention within the host network.

Processor, memory, and peripheral interconnects have been studied for decades in the computer architecture community [16, 23, 31, 36, 37, 47–54, 62–64]; however, these works primarily focus on the behavior of individual interconnects rather than the interplay between these interconnects that leads to contention within the host network. Recent work from the computer networking community [1, 2, 42, 44, 55] studies the impact of contention within the host network on the behavior of end-to-end network protocols (*e.g.*, packet queueing and drops at the host), rather than characterizing the root causes of contention within the host network. Thus, our understanding of the host network—especially the interplay between processor, memory, and peripheral interconnects that leads to contention within the host network—is rudimentary at best. The goal of this paper is to advance this status quo.

The key idea that drives our study is domain-by-domain credit-based flow control, a conceptual abstraction to study the host network. We demonstrate that the host network can be decomposed into multiple "domains" (sub-network of the host network)[1], each of which uses an independent credit-based flow control mechanism. Specifically, the sender of each domain is assigned credits that are used to limit the amount of data the sender can inject into the domain; the sender consumes a credit to send one message, and this credit is replenished when the message receipt is acknowledged by the receiver of the domain. Different domains within the host network have different numbers of credits and different unloaded latency. Each data transfer, depending on the source (compute or peripheral device) and on the type (read or write), traverses a different set of domains. The end-to-end performance for each transfer depends on the number of credits and the per-request latency of domains traversed by that transfer. Many details of existing host network hardware are not public; nevertheless, we reverse engineer Intel host architecture to characterize each domain, its credits, and its unloaded latency.

The lens of domain-by-domain credit-based flow control enables us to capture the subtle interplay between processor, memory, and peripheral interconnects that leads to nanosecond-scale latency inflation in certain domains. This, along with the knowledge of domains traversed by each data transfer, allows us to near-precisely explain how nanosecond-scale inefficiencies within the host network percolate through the host hardware, operating systems, and network protocols to negatively impact application-level performance. We do this

[1]The notion of a domain here is different from administrative domains in the Internet architecture, and in ATM networks [4, 13, 20, 21]. Importantly, unlike administrative domains, the domains in our study do not need to be non-overlapping.

both for applications generating peripheral-to-memory traffic (P2M apps, *e.g.*, networked and storage applications [11, 14, 61, 69, 71]) and for applications generating compute-to-memory traffic (C2M apps, *e.g.*, in-memory databases [17, 19, 57, 60] and systems for graph analytics [24, 40, 45]). More concretely:

- We reproduce the phenomenon of contention within the host network and its impact on networked applications with both in-kernel [2] and hardware-offloaded [42, 44] transport protocols. We find that P2M apps can indeed suffer from performance degradation, *e.g.*, when both C2M and P2M apps are doing writes. We provide precise root causes for the phenomenon. We also extend observations made in all prior studies [1, 2, 42, 44]: we demonstrate (and provide explanation for) degradation in C2M app performance along with P2M app performance.
- We identify new, previously unreported, regimes of contention within the host network: we find that—in sharp contrast to the phenomenon observed in previous studies [1, 2, 42, 44]—P2M apps can, in fact, cause severe performance degradation for C2M apps for most workloads, with minimal or no impact on the P2M app performance. These new regimes are reproducible across multiple generations of servers with different processors, different memory bandwidth to core count ratios, and different configurations (*e.g.*, with and without direct cache access [28], with and without prefetching, etc.).

Our study suggests that the impact of contention within the host network has implications that are more far-reaching than the context of networked applications considered in previous studies [1, 2, 42, 44]. In particular, all our observations hold even when all applications are contained within a single host (*e.g.*, using storage applications that generate P2M traffic using locally-attached storage devices). Thus, our work may be of independent interest to researchers and practitioners not only in computer networking but also in operating systems and computer architecture.

The code, along with the documentation necessary to reproduce our results, is available at https://github.com/host-architecture/understanding-the-host-network.

2 HOST NETWORK CONTENTION REGIMES

In this section, we broadly characterize the interplay of processor, memory, and peripheral interconnects within the host network using four "quadrants" (§2.2) that reveal different regimes in terms of contention within the host network and performance degradation for C2M and P2M apps. Our key findings are:

- The first regime, referred to as the blue regime, captures a new phenomenon: C2M apps observe performance degradation, while P2M apps observe minimal or no performance degradation. Surprisingly, this phenomenon can happen even when memory bandwidth is far from saturated.
- The other regime, referred to as the red regime, captures the phenomenon observed in previous studies [1, 2, 42, 44]: P2M apps observe severe performance degradation when memory bandwidth gets saturated. In addition, we find that C2M apps also observe significant performance degradation.

This section focuses on characterizing the host contention regimes; we discuss the root causes in §5. We first focus on a setup where all traffic is contained within the host (P2M traffic generated by locally attached storage devices)—this allows us to isolate the impact of contention within the host network from the impact of network protocol behavior (packet drops, queueing delays, and/or PFC pause frames) on application performance. We then discuss how our observations generalize to networked applications in §2.3.

	Ice Lake	Cascade Lake
CPU	Xeon Platinum 8362	Xeon Gold 6234
Cores	32 @ 2.8GHz	8 @ 3.3GHz
LLC	48MB	24MB
DRAM	4×3200MHz DDR4	2×2933MHz DDR4
DRAM BW	102.4GB/s	46.9GB/s
PCIe	8×PM173X NVMe	4×P5800X NVMe
PCIe BW	32GB/s	16GB/s

Table 1: Hardware configuration of our two servers. All of the specifications are for a single socket in each server. DRAM and PCIe bandwidths are theoretical maximum values.

We use two testbeds with different processors and different resource ratios (Table 1). The first testbed uses Intel Ice Lake processors and has roughly the same resource ratios (cores, memory bandwidth and PCIe bandwidth) as the testbed in the Google study [1]. The second testbed uses Intel Cascade Lake processors and has a lower core to memory bandwidth ratio. We run our experiments on a single socket. Each DRAM module is attached through a separate channel, and a simple sequential read microbenchmark saturates more than 90% of theoretical maximum memory bandwidth.

2.1 Host network contention with real applications

We first present the new phenomenon: C2M apps observing performance degradation, with minimal or no performance degradation for P2M apps. This phenomenon is reproducible for a variety of configurations: multiple C2M apps with different compute-to-memory bandwidth demands, multiple server configurations, with and without Intel Data Direct I/O (DDIO) [28] technology, and with and without prefetching.

C2M and P2M apps used in our experiments. We use two C2M apps, each with different compute-to-memory bandwidth demands and with different access patterns. The first C2M app is a popular in-memory database called Redis [57], and the second C2M app is a standard graph processing framework called the GAP Benchmark Suite (GAPBS) [8]. GAPBS is more memory bandwidth intensive and performs lighter-weight computations than Redis.

For Redis, we use the standard sharding-based multi-core deployment setup [12]—multiple independent Redis server instances (each with its own keyspace) running on a dedicated set of cores. Clients run on a different set of dedicated cores (1 client core per server core) and issue queries to the server instances using Unix domain sockets (the most efficient inter-process communication mechanism supported by Redis [58]). We use the standard YCSB-C (100% read) workload with a uniform random access pattern; as is standard, clients issue queries with parallelism given by the knee-point of the latency-throughput curve. Performance is measured in terms of the throughput (queries/sec). The working set size per server core is 1 million key-value pairs with a 1KB value size, exceeding the system Last-Level Cache (LLC) even for a single server core. As a result, the observed cache miss ratio is >95%, and a large number of C2M memory reads are generated. For GAPBS, we run the

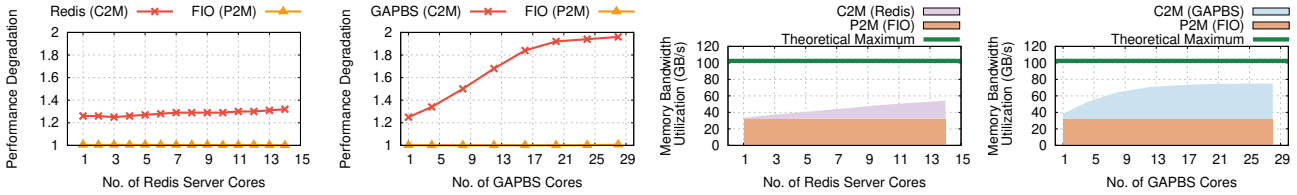

Figure 1: A new phenomenon of contention within the host network: C2M and P2M apps are colocated, C2M app performance degrades while P2M app performance is unaffected. This happens even though cores are isolated and memory bandwidth is far from saturated. (a-d; left-right) (a, b) Performance degradation observed by C2M and P2M when they are colocated (ratio of isolated throughput and colocated throughput for each data point; degradation for GAPBS is the slowdown—ratio of colocated execution time to isolated execution time); (c, d) Memory bandwidth utilization when C2M and P2M are colocated, broken down by C2M and P2M.

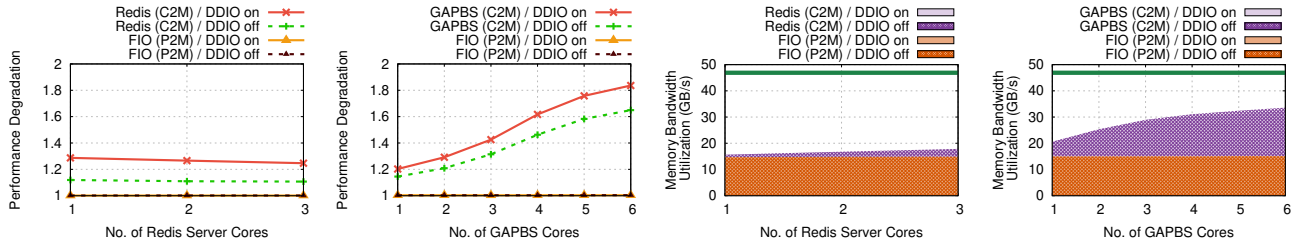

Figure 2: Enabling DDIO can worsen performance degradation for both C2M and P2M applications when the working set size does not fit in cache (left-right, a-d) (a, b) Performance degradation for C2M and P2M when they are colocated with DDIO on/off; (c, d) Memory bandwidth utilization when C2M and P2M are colocated with DDIO on/off.

PageRank workload on a random graph of 2^{25} nodes and degree 16, using the GAPBS default parameters. Performance is measured in terms of execution time (lower is better). A single graph instance is shared across all the cores. The workload has a ~5GB memory footprint, significantly larger than the cache, resulting in a large number of random C2M memory reads. We focus on non-cache-resident memory-intensive workloads since the phenomenon was observed in datacenters for similar workloads [1, 42].

We use a lightweight storage-based P2M app, FIO [7], that performs storage accesses with minimal computational overhead. We configure it to perform sequential reads with 8MB request sizes. This is representative of storage workloads that perform large sequential operations, for example, storage nodes of distributed data stores [61] for analytics workloads. The performance metric is throughput measured in IOPS. The reads result in direct memory access (DMA) writes to host memory from the storage device, leading to a large volume of P2M write traffic. While DDIO minimizes P2M traffic for many workloads by servicing DMA requests from the cache instead of memory, it is well known that it is not effective for all workloads [10, 18, 66]. Our P2M workload is in the latter category—due to the large sequential requests, the application buffers do not fit into the small portion of the cache that DDIO is allowed to use [18], thus leading to cache misses and evictions/writebacks for every DMA in steady state. Therefore, we observe nearly the same average memory bandwidth utilization for this workload with/without DDIO.

In the following experiments, we first run each of the C2M and P2M apps in isolation. We then colocate them and measure the performance degradation for each app. We start with the Ice Lake setup (Table 1). We partition cores between the applications by pinning each to a separate set of cores—we dedicate 4 cores to the P2M app (which is more than sufficient to saturate PCIe bandwidth without compute being a bottleneck) and run the C2M app on the remaining cores with a varying number of cores. We enable DDIO and hardware prefetching on this setup.

C2M app performance degrades even when memory bandwidth is far from saturated. When Redis (C2M) and FIO (P2M) are colocated, as shown in Figure 1(a), Redis observes throughput degradation ($1.25 - 1.32\times$) while FIO remains unaffected. The surprising observation here is that degradation is observed even though cores and PCIe bandwidth are isolated across the applications and memory bandwidth utilization is far from saturation as shown in Figure 1(c) (ranging between $33 - 53\%$ of the theoretical maximum, and the utilization curve has not flattened out). Although the LLC is shared between the applications, it does not play much of a role in determining performance, since both C2M and P2M traffic observe nearly 100% cache miss ratio even when they are run in isolation. We investigate the root cause of this performance degradation in §5.

Using GAPBS rather than Redis results in the same high-level observation—C2M performance degrades ($1.28 - 1.98\times$) while P2M is unaffected (Figure 1(b)). This again happens even when memory bandwidth is far from saturated (as shown in Figure 1(d) for fewer than 15 GAPBS cores). As one would intuitively expect, the magnitude of performance degradation is larger for GAPBS compared to the Redis since it is more memory intensive—Redis spends only a part of its time stalled on memory accesses, while GABPS is stalled on memory accesses nearly all of the time.

Impact of prefetching, processor generations, and resource ratios. Given the random access nature of the Redis and GAPBS workloads, hardware prefetchers have little to no impact on performance. For both workloads, we found $< 5\%$ difference in performance when comparing prefetch on/off configurations in both isolated and

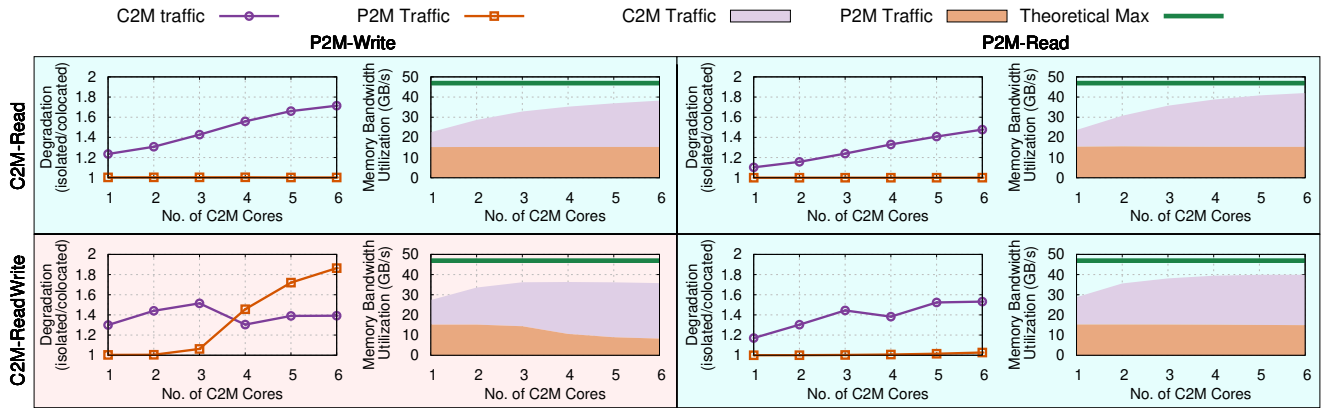

Figure 3: Blue and red regimes across four quadrants (1-4 are shown on top-left, top-right, bottom-left, bottom-right respectively). Quadrants are shaded with the color of the regime they show. For each quadrant, the left column shows the throughput degradation observed by C2M and P2M (ratio of throughput when run in isolation to the throughput when colocated) and the right column shows the memory bandwidth utilization when they are colocated broken down by C2M and P2M traffic.

colocated cases. We repeat the above experiments on the Cascade Lake setup (Table 1). Here, we dedicate 2 cores to FIO and run the C2M app on the remaining cores. The corresponding results are shown in the (DDIO on) curves of Figure 2. We see the same general observation as in the Ice Lake setup—C2M app performance degrades while the P2M app observes no degradation, thus showing that our observations apply across different processor generations and resource ratios. Similar observations apply even when using different read/write ratios for the C2M and P2M applications (results presented in Appendix B).

DDIO can worsen performance degradation when app working size does not fit in cache. As discussed previously, DDIO is not effective for our P2M workload and does not reduce its memory bandwidth utilization when run in isolation. To study if DDIO has any second-order impact when C2M/P2M apps are colocated, we re-ran the above experiments with DDIO disabled on our Cascade Lake setup (this was not possible on our Ice Lake setup because DDIO is permanently enabled there). The corresponding results, shown in Figure 2, reveal a surprising observation: DDIO results in worse performance degradation for C2M apps for both Redis and GAPBS (Figures 2(a), 2(b)). This is surprising because our C2M workloads already have ~100% cache miss ratio even when run in isolation, and thus should ideally not be impacted by cache evictions caused by DDIO. We do not know how to explain this observation.

2.2 The Blue and Red Regimes

Our experiments in §2.1 use real open-sourced apps that have fixed memory access patterns. We now switch to using lightweight apps with easy to control memory access patterns, enabling us to perform a deeper characterization of performance degradation trends and to study different combinations of read/write for C2M/P2M apps.

Workloads. To generate C2M traffic, we use a modified version of the STREAM [46] benchmark that supports different read/write ratios. We use two C2M workloads: (1) a read-only workload that sequentially reads data from a 1GB buffer (using 64-byte AVX512 load instructions). This results in 100% memory reads (*C2M-Read*). (2) a write workload that sequentially writes data to a 1GB buffer (using 64-byte AVX512 store instructions). This generates 50% read and 50% write memory traffic since every cacheline is first read into the CPU's cache before the store instruction can be serviced and is later written back to memory during cache eviction (*C2M-ReadWrite*). For P2M, we run FIO with (1) 100% storage reads which translates to 100% memory writes (*P2M-Write*) and (2) 100% storage writes which translates to 100% memory reads (*P2M-Read*). Both our C2M and P2M workloads perform sequential accesses.

We run experiments on our Cascade Lake setup while colocating each of the two C2M workloads with each of the two P2M workloads. This results in a total of four scenarios, which we refer to as the four "quadrants". We disable prefetching and DDIO for better explicability. We found that while enabling each of these leads to different absolute degradation numbers, the trends and takeaways remain the same (when memory bandwidth is not saturated, prefetching improves C2M throughput in both the isolated and colocated cases, but their ratio remains roughly the same). The degradation observed in the quadrants is shown in Figure 3. We classify the observations into two key regimes:

Blue regime: C2M throughput degrades while P2M throughput does not. In quadrant 1 (C2M-Read, P2M-Write), we observe $1.2 - 1.7\times$ degradation in C2M throughput while P2M throughput remains unaffected. This degradation happens when memory bandwidth is far from saturated (for example, with a single C2M core), and increasing load leads to worse C2M throughput. Similarly, in quadrant 2 (C2M-Read, P2M-Read), while C2M throughput degrades, P2M throughput remains unaffected. C2M throughput degradation is lower than quadrant 1. Quadrant 4 (C2M-ReadWrite, P2M-Read) observes the same trend as in quadrant 2.

Red regime: Both C2M and P2M throughput degrade. Quadrant 3 (C2M-ReadWrite, P2M-Write) shows a range of different performance degradation trends. With 2 or fewer C2M cores, similar to quadrants 1 and 2, C2M throughput degrades while P2M throughput does not. For 3 C2M cores and above, once memory bandwidth is saturated, we see a completely different trend—C2M traffic now antagonizes P2M by getting an increasingly larger share of the memory

bandwidth with increasing load, leading to larger throughput degradation for P2M traffic compared to C2M traffic. This captures the observations reported by recent works [1, 42]. P2M traffic, however, does not get starved at higher load—with 5 and 6 C2M cores, we observe a relative stabilization of the memory bandwidth shares of C2M and P2M traffic.

2.3 Networking Case Studies

Our characterization of host contention regimes in §2.2 generalizes to cases where P2M traffic is generated by a NIC instead of local storage devices, and networked transfers use either kernel-based or hardware-offloaded transport mechanisms. We briefly summarize these observations below; full details are presented in Appendix C.

RDMA. Using RDMA over Converged Ethernet with Priority Flow Control (RoCE/PFC), we observe the same blue and red regime trends from §2.2 for each of the C2M/P2M read/write combinations—RoCE/PFC throughput degrades in the red regime; on the other hand, in the blue regime, RoCE/PFC throughput remains unaffected and causes significant C2M app throughput degradation.

DCTCP. With Linux DataCenter TCP (DCTCP) over lossy fabric, we again observe the same blue and red regimes, although the observed application-level performance trends are slightly different. In particular, the networked application observes performance degradation in each regime. This is because, in addition to P2M traffic, the networked application also generates C2M traffic due to the data copy between application buffers and kernel socket buffers. In the blue regime, C2M throughput degradation slows down data copy processing, resulting in CPU bottleneck; this results in DCTCP flow control kicking in, reducing the P2M traffic load. In the red regime, P2M throughput degrades but no congestion signal is sent back to the sender until packets are dropped at the NIC; this results in further throughput degradation, latency inflation, and violation of isolation properties as outlined in [1, 2].

Given that both the setups—P2M traffic from within the host, and P2M traffic from datacenter network transfers—lead to similar observations, we focus on the former setup for the rest of the paper. It makes it easier to describe all our results. We present corresponding results for the RDMA and DCTCP scenarios in Appendix D, Appendix E.

3 BACKGROUND

In this section, we provide a brief primer on the host network architecture and the datapath for C2M and P2M requests.

Figure 4 shows the host architecture: it consists of cores (with private L1/L2 caches), the LLC, the Caching and Home Agent (CHA), the Integrated IO controller (IIO), and the Memory Controller (MC) all connected by the on-chip processor interconnect. The CHA abstracts away the LLC and memory from the rest of the system while maintaining cache coherence[2]. Peripheral devices are attached to the IIO through the peripheral interconnect (typically PCIe). DRAM consists of a set of modules (Dual Inline Memory Modules, or DIMMs), each of which is attached to the MC through a memory channel.

[2]Both the CHA and LLC are physically distributed into multiple slices. Based on the physical address, memory requests are routed to the correct slice. For simplicity, we represent the CHA/LLC as a single logical entity.

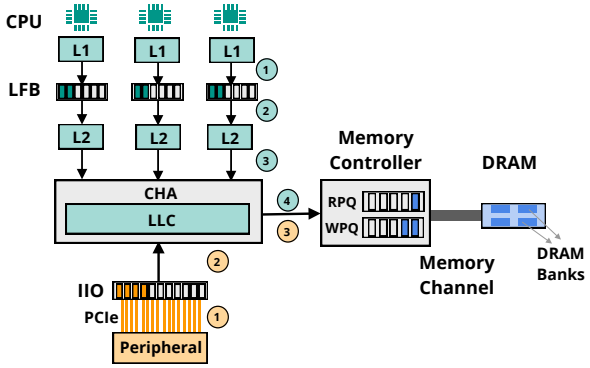

Figure 4: Host network architecture, and C2M and P2M datapaths. Details in §3.

These memory channels constitute the memory interconnect. For simplicity, in Figure 4, we show a single module attached through a single memory channel.

C2M datapath. CPU-to-memory reads are generated by cores upon a cache miss through the following data path:

1. Upon an L1 cache miss, an entry is allocated in the core's Line Fill Buffer (LFB), and a request is sent to the L2 cache.
2. Upon reaching the L2 cache, the cacheline is either served from the L2 cache upon a cache hit (and the LFB entry is freed) or the request is sent to the CHA upon a cache miss.
3. The CHA serves the cacheline from the LLC if there is an LLC hit (and the LFB entry is freed). Otherwise, the CHA sends the request to the MC, where it is queued in the Read Pending Queue (RPQ).
4. The MC fetches the required cacheline from DRAM over the memory channel and returns the data back to the core while populating the caches and ultimately freeing the corresponding LFB entry.

Memory writes are generated upon cache evictions and follow a similar datapath: a write-back from the L2 cache is sent to the CHA, which either services it from the LLC or sends a write request to the MC, where it is queued in the Write Pending Queue (WPQ). The MC eventually issues the write to DRAM over the memory channel. *Importantly, unlike reads, writes generated by cores are asynchronous*: the CPU only has to wait for the request to be admitted to the CHA, and the CHA only has to wait for the request to be admitted to the WPQ.

P2M datapath. Each peripheral-to-memory request (read/write) incurs the following datapath:

1. Peripheral device initiates a DMA request to the IIO, that allocates an entry in the IIO (read/write) buffer per cacheline.
2. IIO forwards the requests (at cacheline granularity) to the CHA. If DDIO is enabled and there is a cache hit, the CHA serves the request from the LLC (and the IIO buffer entry is freed). Otherwise, the CHA sends the request to the MC, where it is enqueued in the RPQ/WPQ.
3. The MC serves read/write requests in a manner similar to the C2M datapath. After a read is serviced from DRAM, the data is returned to the IIO, at which point the IIO buffer entry is cleared

and the data is sent back to the peripheral device. For writes, the IIO only needs to wait until the request is admitted to the WPQ before freeing its buffer entry.

The interconnects within the host physically implement hop-by-hop flow control mechanisms to ensure losslessness [29]. In the peripheral interconnect, this is implemented through the exchange of PCIe credits between the peripheral device and the IIO [2, 54]. The peripheral device needs a PCIe credit to send a request to the IIO; this credit is replenished once the corresponding IIO buffer entry is freed. In the memory interconnect, flow control happens implicitly through DRAM timing constraints [35, 41]. In the processor interconnect, implementation details of credit exchange are not public.

DRAM operation. MC reads/writes cachelines from/to DRAM over memory channels. Each memory channel can only transmit data in one direction (either reads or writes) at any point in time. The MC, therefore, operates in two separate modes, read mode and write mode, and maintains separate queues for reads and writes (RPQ and WPQ, respectively) per memory channel. Due to electrical constraints, switching between modes takes a certain delay (called the switching delay) during which the channel is idle [41]. The data in each DRAM module is organized into multiple banks. Each bank has multiple rows, each of which stores a fixed number of cachelines, and a row buffer that can buffer a single row at any time. In order to access a cacheline, its corresponding row needs to be present in the corresponding bank's row buffer. If not, this results in a row miss, which incurs additional processing delay at the banks: The row needs to be loaded into the row buffer using an Activate (ACT) operation. If the row buffer contains a different row (i.e., row conflict), then it needs to be flushed using a Precharge (PRE) operation before a new row can be loaded, which incurs additional overhead.

For the remainder of the paper, we focus on the scenario where C2M/P2M requests result in misses at all levels of the cache hierarchy (as is the case in the §2 experiments).

4 UNDERSTANDING THE HOST NETWORK

This section presents domain-by-domain credit-based flow control, a conceptual abstraction that captures the interplay between the processor, memory, and peripheral interconnects within the host network. We start by describing the abstraction along with the various domains and their characteristics in §4.1. We then describe in §4.2 how we reverse engineered the Intel host architecture to characterize each domain, its credits, and its unloaded latency.

4.1 Domain-by-domain credit-based flow control

We begin by defining domain-by-domain credit-based flow control. The host network is logically decomposed into multiple domains, each of which is a sub-network of the host network. Each domain uses an independent credit-based flow control mechanism. Specifically, the sender of each domain is assigned credits that are used to limit the number of in-flight requests that the sender can inject into the domain; the sender consumes a credit to send one request, and this credit is replenished when the request is acknowledged by the receiver of the domain. Depending on the number of credits, at any given point of time, there can be multiple concurrent in-flight requests within each domain.

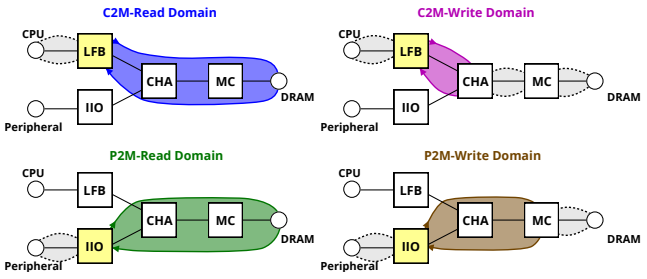

Figure 5: Domain-by-domain credit-based flow control in the host network: Different shaded regions within each sub-figure depict independent domains within the host network for respective C2M and P2M read/write datapaths. Data is transmitted between the CPU/Peripheral and the DRAM by traversing each domain using an independent credit-based flow control mechanism. The specific domain highlighted in color for each datapath is particularly interesting: these are the domains that will turn out to be the bottleneck within individual datapaths. Different domains can span different numbers of hops, leading to different domain latencies. Further, the number of domain credits (limited by the node marked as yellow) is also different for P2M vs C2M domains.

Intuitively, domain-by-domain credit-based flow control generalizes the two flow control mechanisms studied in classical computer networking literature. On the one hand, end-to-end flow control mechanisms (*e.g.*, used in TCP and in receiver-driven datacenter transport protocols [9, 22, 25, 26]) are a special case where the entire path between a sender-receiver pair is a single domain. On the other hand, hop-by-hop credit-based flow control mechanisms (*e.g.*, used in ATM networks [32, 38, 39] and PFC-enabled RDMA networks [43, 72]) are a special case where each hop along the path between the sender-receiver pair is a domain.

Different domains within the host network have different numbers of credits and different unloaded latencies. Each request, depending on the source (compute or peripheral device) and on the type (read or write), traverses a different set of domains. Figure 5 shows the domains within the host network for each of the C2M and P2M read/write datapaths, with cores, peripheral devices, and DRAM as endpoints and intermediate components (i.e., LFB, IIO, CHA, and MC) as network nodes. We now discuss the four domains that turn out to be the most important ones (in that, these will be the "bottleneck" domains in individual datapaths):

- **C2M-Read Domain:** This domain spans all hops from LFB to DRAM. For each request, credit is allocated at the LFB and replenished once the request is serviced by DRAM and the data is returned to the LFB.
- **P2M-Read Domain:** This domain spans all hops from IIO to DRAM. For each request, credit is allocated at the IIO and replenished once the request is serviced by DRAM and the data is returned to the IIO.
- **C2M-Write Domain:** This domain spans only a single hop from LFB to CHA. For each request, credit is allocated at the LFB and replenished once the request reaches the CHA.
- **P2M-Write Domain:** This domain spans two hops from IIO to MC. For each request, credit is allocated at the IIO and replenished once the request reaches the MC.

The maximum throughput (T) for any domain is bound by $T \leq \frac{C \times 64}{L}$, where C is a constant representing the hardware-specific number of credits available to the sender in the domain (in terms of cachelines), 64 is cacheline size in bytes, and L is a variable representing the latency required to traverse all hops within the domain. Each of these factors can be different depending on the domain:

Domain Credits (C). The number of credits for the C2M-Read and C2M-Write domains is limited by the LFB size. The number of credits for the P2M-Write domain is limited by the IIO write buffer size. The number of credits for the P2M-Read domain is limited by the IIO read buffer size. For our servers, these numbers are $10-12$, $\sim$92, and $>$164 cachelines, respectively.

Domain Latency (L). Different domains span a different subset of network hops; this could result in different domains having different latencies for two reasons. First, simply due to spanning a different subset of network hops, different domains may have different unloaded latencies. Second, when C2M and P2M traffic contend for host network resources, queueing at the contention point may have different impact on different domains (only those domains are impacted that contain the contention point). As a result, contention within the host network may result in latency inflation for some domains but not others.

Given the number of credits and latency for a domain, the maximum throughput of that domain is given by the expression $T \leq \frac{C \times 64}{L}$, as discussed above. The overall end-to-end throughput of a particular C2M or P2M app is the minimum throughput across all domains along the datapath for that app.

4.2 Evidence on domains and their characteristics

We now present evidence for domains and their characteristics, including a discussion of how we reverse-engineered several of the details by piecing together information from processor manuals [29, 30] and conducting careful measurements.

Measurement Methodology. We use the Intel uncore performance monitoring counters [29] to capture average queue/buffer occupancy (O) and average request arrival rate (R) metrics at different nodes in the host network. In particular, we program the counters so that their values are aggregated in hardware every clock cycle and sample them at runtime in software every 1 second, which entails very low overhead. To compute average latency, we apply Little's law on the measured O and R values ($L = O/R$). We use the umask and opcode filtering capabilities of CHA counters to classify requests based on their source (CPU/Peripheral) and type (read/write), allowing us to capture all the above metrics on a per-domain basis.

C2M-Read. The C2M-Read domain spans all hops from LFB to DRAM because an LFB entry (and corresponding credit), once allocated, is only freed (and corresponding credit replenished) once the memory read request is serviced from DRAM and returned to the core to prevent duplicate memory requests to the same cacheline [30, 67]. To validate this, we perform latency measurements while running the C2M-Read workload (§2.2) with varying number of cores. Figure 6(a) shows the measured LFB latency (time between allocation and replenishment of an LFB credit) alongside the CHA→DRAM read latency (time taken for request to traverse from CHA to DRAM and for response to return to CHA). As is evident from the figure, the LFB latency is always strictly greater than the CHA→DRAM read latency. Further, the inflation in LFB latency from 1 to 6 cores near perfectly matches inflation in CHA→DRAM read latency. This shows that the LFB latency is inclusive of the CHA→DRAM read latency, thus providing evidence that the C2M-Read domain includes all hops until DRAM. In all of our experiments, the maximum measured LFB occupancy is between $10-12$ providing evidence that this the number of domain credits (also corroborated in [15]). The unloaded domain latency is $\sim$70ns, as is evident from the single-core data point in Figure 6(a).

C2M-Write. It is clear that the C2M-Write domain includes the LFB, the CHA (since each domain must span at least two nodes), and that it does not include DRAM (since writes are serviced to DRAM asynchronously [29]). The key challenge lies in determining whether the MC is part of the domain. To do so, we perform latency measurements while running the C2M-ReadWrite workload (§2.2) with varying number of cores. For this workload, the LFB latency is equal to the sum of the C2M-Read and C2M-Write domain latencies (and thus must be strictly greater than each of them). If the C2M-Write domain included the MC, then the C2M-Write latency (and consequently LFB latency) would always be strictly greater than the CHA→MC write latency (time taken for the request to traverse from CHA to MC). However, as shown in Figure 6(b), the CHA→MC write latency can exceed the LFB latency (e.g., with 6 C2M cores), thus implying that the C2M-Write domain does not include the MC. Subtracting the unloaded C2M-Read domain latency from the LFB latency at the single core data point in Figure 6(b) gives us an estimate of $\sim$10ns unloaded latency for the C2M-Write domain.

P2M-Write. To understand P2M-Write domain, we run a low-load P2M workload performing 4KB storage read requests with queue depth of 1, colocated with C2M-ReadWrite workload. Figure 6(c) shows the IIO latency (time between credit allocation and replenishment at IIO) alongside the CHA→MC write latency. We make three observations. First, the unloaded domain latency is $\sim$300ns. Second, the IIO latency is always larger than the CHA→MC write latency. Finally, the inflation in IIO latency with increasing load near perfectly matches the inflation in CHA→MC write latency (Figure 6(d)), indicating that IIO latency is inclusive of the CHA→MC write latency. This provides evidence that unlike C2M-Write domain, P2M-Write domain includes the MC. To determine the P2M-Write domain credits, we run P2M-Write workload from §2.2 (which saturates PCIe bandwidth), and apply maximum possible C2M load. We find that IIO write buffer occupancy saturates at $\sim$92, giving us the size of the IIO write buffer.

P2M-Read. The P2M-Read domain spans all hops from IIO to DRAM because PCIe reads are non-posted transactions [54]—the IIO needs to wait until reads are serviced from DRAM and data is returned before issuing PCIe completion and replenishing the credits. We were not able to measure IIO read buffer occupancy on our server (and consequently IIO read latency), thus precluding us from determining the precise number of credits and unloaded latency of the P2M Read domain. However, we obtain a lower bound on the P2M-Read domain credits using measurements at the CHA (since the number of in-flight P2M-Read requests at the CHA cannot

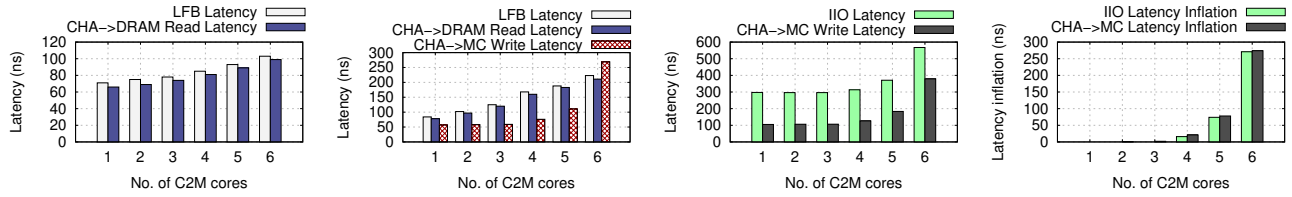

Figure 6: Evidence for domains, and per-domain characteristics (a-d, left-right). All y-axis values are on average. Discussion in §4.2.

exceed the P2M-Read domain credits). By introducing C2M load colocated with P2M-Read traffic, we found that the number of in-flight P2M-Read requests at the CHA saturates at ∼164 cachelines, providing evidence that the P2M-Read domain has a larger number of credits than the P2M-Write domain.

5 UNDERSTANDING CONTENTION WITHIN THE HOST NETWORK

We now provide an in-depth explanation for the two regimes observed in §2 using the lens of domain-by-domain credit-based flow control. We use the same measurement methodology as in §4.2. We observe no statistically noticeable change in application performance when counter sampling is enabled. We disable dynamic scaling of core frequency to avoid variation in measurements (we observe less than 1.5% variation across all runs for all counters).

5.1 Understanding the blue regime

We first focus on quadrant 1 (C2M-Read, P2M-Write) since it captures most of the takeaways in terms of explaining the blue regime. For quadrant 1, we first explain why C2M throughput degrades and then why P2M throughput does not degrade.

C2M throughput degrades because domain credits are fully utilized and domain latency increases. Even when the C2M workload is run in isolation, the corresponding domain credits are fully utilized. This is because each core can issue instructions fast enough to keep the LFB full (e.g., a core with 3GHz frequency can issue instructions every 0.3ns, which is more than two orders of magnitude smaller than the minimum C2M-Read domain latency). As a result, any non-zero increase in domain latency will result in throughput degradation. Indeed, when the P2M workload is colocated, we observe $1.26 - 1.8\times$ increase in domain latency (Figure 7(a)) due to queueing at the MC (Figure 7(b)) since DRAM is part of the C2M-Read domain. Interestingly, such queueing happens far before memory bandwidth is saturated; we now explain this phenomenon.

Queueing at the MC before memory bandwidth saturation happens due to a combination of two DRAM-level factors: (1) row misses and (2) load imbalance across banks. Row misses result in processing delays at the banks (due to precharge/activate operations), which must be completed before the data can be accessed and transmitted over the memory channel. Even for a workload with 100% row miss ratio, the bank-level processing delays can still be hidden behind data transmission over the memory channel, if requests are load balanced perfectly across the banks. If requests are perfectly distributed across N_b banks, then the bank processing delay can be hidden/overlapped behind transmission on the channel if $t_{\text{Proc}}/N_b < t_{\text{Trans}}$, where t_{Proc} is the per-request bank processing delay and t_{Trans} is the per-request transmission delay over the memory channel. This condition holds for the DRAM modules in our setup, where $t_{Proc} \approx 45ns$, $N_b = 32$, and $t_{Trans} = 2.73ns$. In reality, however, load balancing is far from perfect since memory addresses are mapped to banks through a static hash function [56], which does not guarantee perfect load balancing [70]. As a result, requests can be blocked on bank processing even when the memory channel is idle, thus causing queueing even when channel capacity (i.e., memory bandwidth) is not saturated. We quantify row misses and load imbalance, focusing on the single core C2M case in quadrant 1 next.

In the absence of P2M traffic, the row miss ratio for C2M-Read requests is very low (< 4%, shown in Figure 7(c)). This is because of the sequential access pattern resulting in a good row locality. Colocating the P2M workload causes a significant increase in row miss ratio (up to 4×). This is because the C2M and P2M workloads access different address spaces — intermixing them reduces row locality, leading to a higher row miss ratio. While row miss ratio also increases for C2M-only traffic from multiple cores since each core accesses a different address space, colocating P2M traffic leads to a larger increase in row miss ratio as is evident in Figure 7(c).

To measure load distribution, we sample the number of read requests mapped to each individual bank every 1000 requests[3]. Let the *bank deviation* of a given sample be the ratio of the load of the maximally loaded bank to the average load across banks. Figure 7(d) shows the CDF of bank deviation across 10000 samples. We see significant load imbalance, both with and without P2M traffic — the bank deviation is $\geq 1.5\times$ in $50 - 70\%$ of samples and $\geq 2\times$ in as many as $13 - 22\%$ of samples (although there is load imbalance even when C2M is run in isolation, it is not a problem since the row miss ratio is very low causing bank processing delays to be negligible).

Returning to our main discussion, P2M traffic observes different behavior compared to C2M traffic in quadrant 1 due to differences in: (1) domain latency and (2) domain credits.

Write domain does not include execution latency of DRAM, leading to smaller latency inflation relative to reads. Unlike the C2M-Read domain, where queueing at the MC leads to domain latency inflation, since the P2M-Write domain does not include traversing DRAM, its domain latency only increases when the MC write queue becomes full. As shown in Figure 7(f), the fraction of time the WPQ is filled is near zero when P2M-Write is colocated with a single C2M-Read core. Thus, there is no domain latency inflation for P2M-Write (Figure 7(e)). With increasing C2M cores, while we see a small increase (< 25ns) in domain latency for P2M-Write (Figure 7(e)), it is smaller than what is seen for C2M-Read. This is

[3]For these measurements, we use a dedicated core that busy polls on MC hardware counters [29]. Given constraints on the number of available hardware counters, we focus on 4 banks within a single DRAM module.

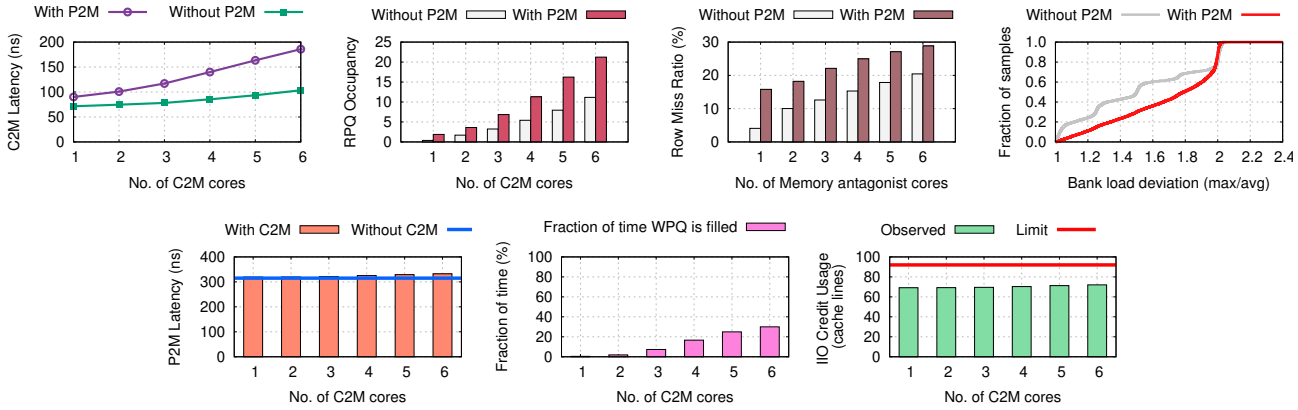

Figure 7: Results for understanding quadrant 1 (a-d, top/left-right and e-g, bottom/left-right). All y-axis values are on average. Discussion in §5.1.

because the WPQ starts to get filled up occasionally (< 30% of the time), as shown in Figure 7(f).

P2M domain can tolerate latency inflation due to availability of spare domain credits. Increased P2M-Write domain latency, however, does not lead to a reduction in P2M throughput. This is because, unlike C2M-Read, when the P2M workload is run in isolation, domain credits are not fully utilized (as described in §4, unloaded P2M-Write domain latency is ~300ns; therefore, to saturate PCIe bandwidth of ~14GB/s, ~65 credits are needed which is smaller than the ~92 available credits). P2M throughput degrades only after domain credits are exhausted. Indeed, we see a slight increase in domain credit utilization with increasing C2M cores, but it is well below the maximum limit (Figure 7(g)). Therefore, the P2M-Write domain is able to maintain enough in-flight write requests to mask the latency inflation and thus avoid throughput degradation.

Our explanation of quadrant 1 generalizes to quadrants 2 and 4 (the corresponding measurements are presented in Appendix A).

5.2 Understanding the red regime

We now turn our attention to quadrant 3, where both C2M and P2M observe throughput degradation. Before diving deep, we first briefly revisit the observations in quadrant 3 bearing similarities to §5.1.

With 2 or fewer C2M cores, similar to quadrant 1, C2M throughput degrades even before memory bandwidth is saturated (Figure 3). We see a similar trend of increase in row miss ratio when P2M is colocated with C2M (Figure 8(c)); this, in combination with load imbalance across banks, results in queueing at MC before memory bandwidth is saturated (Figure 8(b)). While there is a small ($\sim 20-30$ns) inflation in P2M-Write domain latency (Figure 8(d)), akin to quadrant 1, there is no P2M throughput degradation due to spare domain credits (Figure 8(f)).

With 3 or more C2M cores, when memory bandwidth becomes saturated, we observe two new trends in this quadrant (§2.2). First, from 3 to 4 C2M cores, we observe C2M antagonize P2M (i.e., reduction in C2M throughput degradation coupled with a large increase in P2M throughput degradation). Second, beyond 4 C2M cores, the rate of degradation of P2M throughput reduces with increasing C2M cores. We now discuss the underlying reasons for both observations.

Backpressure from MC impacts P2M-Write domain but not C2M-Write domain, leading to throughput degradation for the former. When memory bandwidth is saturated, unlike in quadrant 1 (Figure 7(f)), the MC write queue gets filled up persistently (75% of the time with 3 C2M cores, and nearly all the time with 4 or more C2M cores; Figure 8(e)), thus leading to backpressure and causing a backlog of writes at the CHA. Interestingly, this only impacts the P2M workload, but not the C2M workload, even though both are performing writes. This is because the P2M-Write domain spans the MC while the C2M-Write domain does not (§4); therefore, backpressure from the MC results in domain latency inflation of the P2M-Write domain but not C2M-Write domain. From 3 to 4 C2M cores, due to backlogging of writes, P2M-Write domain latency increases by 1.5× (Figure 8(d)) and results in significant P2M throughput degradation since the domain credits are fully utilized (Figure 8(f)). The C2M workload (C2M-ReadWrite), however, is only bound by C2M-Read domain latency (as C2M-Write domain latency does not increase) which only increases by ~12% (Figure 8(a)) since reads are not impacted by write backlogging as they can be processed concurrently at the CHA even when writes are blocked. As a result, C2M's share of memory bandwidth increases while P2M's share reduces.

Backpressure from CHA impacts both C2M and P2M domains, leading to increased degradation for both. As the write backlog at the CHA continues to increase with increasing C2M load, we observe a new phenomenon that is evident beyond 4 C2M cores: CHA begins to apply backpressure due to limited buffering resources. The trend in Figure 8(b) provides evidence of this—the average RPQ occupancy saturates beyond 4 C2M cores (and is lower than the corresponding without P2M values), showing that the number of in-flight read requests from the CHA to the MC has been capped despite the total number of in-flight read requests increasing with more C2M cores. This indicates that some requests are getting blocked at the cores even before being admitted into the CHA—a result of backpressure from the CHA. Under CHA backpressure, write backlogging no longer has a one-sided impact on the P2M domain. Latency inflation is now primarily determined by delay in admitting requests into the CHA itself, which impacts both C2M and P2M domains. We see a roughly equitable increase in domain latency (~50ns) when going

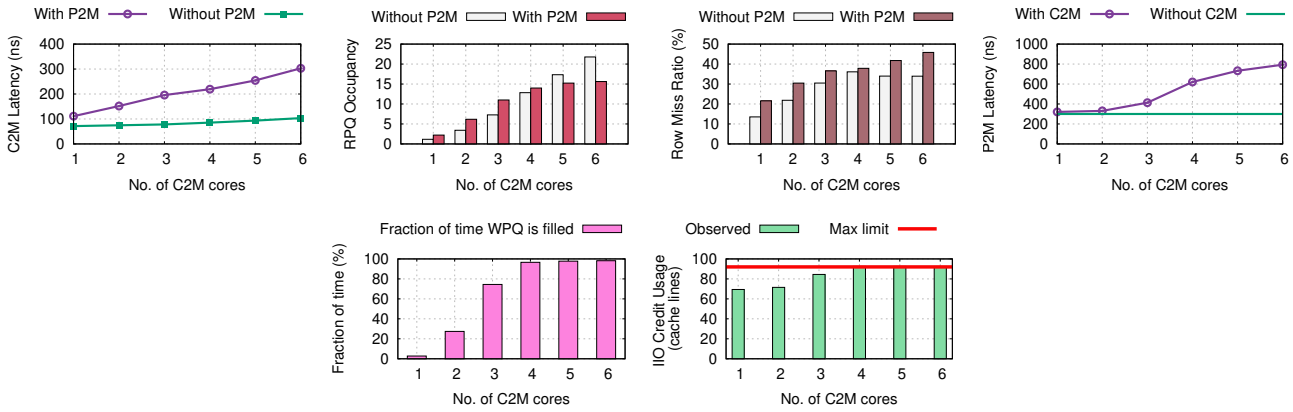

Figure 8: Results for understanding quadrant 3 (a-d, top/left-right and e-f, bottom/left-right). All y-axis values are on average. Discussion in §5.2.

from 5 – 6 cores for both the C2M and P2M domains, thus leading to a relative stabilization of their memory bandwidth shares (Figure 3).

6 QUANTITATIVE VALIDATION

In §5, we identified the root causes for performance degradation due to interplay between processor, memory, and peripheral interconnects, based on correlations with measurements from nodes in the host network. This, however, does not imply that these are the only factors impacting performance degradation. To close this gap and further validate our understanding, we now connect these measurements to the observed end-to-end throughput degradation. To this end, we develop an analytical formula that captures the average memory access latency observed by C2M/P2M traffic (which then directly connects to throughput using Little's law). Our analytical formula focuses on queueing delay at the MC (for reads) and at the CHA (for writes). While, in theory, there can be queueing delay at other points in the host network (e.g. in cache controllers, within the processor interconnect, etc.), we demonstrate that queueing delay at these other points contributes minimally to end-to-end latency (our analytical analysis captures end-to-end performance to a high degree of accuracy across all evaluated workloads). We first describe our analytical formula (§6.1), following which, we present results of applying it to the four quadrants (§6.2).

6.1 Analytical Formula

In designing our analytical formula, we exploit the insight that since we are analyzing latency for the purpose of understanding average throughput, we can focus on average-case behavior across a large number of memory requests rather than on individual request dynamics, which are difficult to capture. Before describing the analytical formula, we highlight that it is not designed to be perfect—it does not capture all the intricacies of DRAM operation, including out-of-order request scheduling, resource contention at some levels of the DRAM hierarchy (e.g., ranks and bank groups), low-level hardware optimizations (e.g., opportunistic processing of memory writes by Intel memory controllers while in read mode [59]), and a subset of DRAM timing constraints (e.g., write recovery delays, rank-level timing constraints, etc.). Despite its relative simplicity, as we will demonstrate, it still captures latency inflation to a reasonable degree of accuracy (within ~ 10% error) in most evaluated scenarios.

We first build the analytical expression for read domain latency following which we discuss write domain latency.

Read Domain Latency. Our formula for read domain latency (Figure 9) is applicable to both the C2M-Read and P2M-Read domains. As motivated at the start of the section, we focus only on the average queueing delay for reads (QD_{read}) at the MC. We, therefore, abstract away all latency in the end-to-end datapath other than QD_{read} into a constant ($\text{Constant}_{\text{read}}$). Naturally, $\text{Constant}_{\text{read}}$ is different for the C2M-Read and P2M-Read domains since they have non-shared hops in their datapaths (§4). QD_{read} is expressed as a sum of four additive components. The first three components capture the average delay for a given read request to reach the top of the RPQ (thus, they are all a function of the average RPQ occupancy O_{RPQ}), and the last component captures the additional delay that a request incurs after reaching the top of the queue before it is issued to DRAM. We now describe each of the individual formula components:

- Switching Delay: This component captures the average time a given read is blocked due to write-to-read switching delay (t_{WTR}). Since we focus on average case behavior, we can compute the total switching cost over a large number of switches (#switches) and average it over a large number of reads ($\text{lines}_{\text{read}}$).
- Write Head-of-Line Blocking: This component captures the average time for which a read is blocked because the channel is currently in write mode and cannot issue reads. Average case analysis allows us to compute the total time spent in write mode over a long time window ($\text{lines}_{\text{written}} \times t_{\text{Trans}}$) and average it across a large number of reads ($\text{lines}_{\text{read}}$).
- Read Head-of-Line Blocking: A given read request has to wait for the requests before it ($O_{\text{RPQ}} - 1$ on average) in the RPQ to get transmitted on the memory channel. In reality, while the MC may schedule requests out-of-order to maximize utilization, our evaluation of the formula indicates that this has very little impact, if any, on the end-to-end latency for the workloads we focus on.
- Top-of-queue delay: Even after reaching the top of the RPQ, a request might still have to wait for activate/precharge operations to complete before it is issued to DRAM. To capture this, we compute the total cost of activate and precharge operations ($\#\text{ACT}_{\text{read}} \times t_{\text{ACT}}$ and $\#\text{PRE}_{\text{read}}^{\text{conflict}} \times t_{\text{PRE}}$) and average them across a large window of requests ($\text{lines}_{\text{read}}$).

Analytical Formula Inputs	
$P_{\text{fill}}^{\text{WPQ}}$	Probability that WPQ is full
N_{waiting}	# write requests awaiting WPQ admission
#switches	# switches between read and write mode
$\text{lines}_{\text{read/write}}$	# cachelines read / written
O_{RPQ}	Average RPQ occupancy
$\text{PRE}_{\text{read/write}}^{\text{conflict}}$	# precharges due to row conflicts for reads / writes
$\text{ACT}_{\text{read/write}}$	# activations for reads / writes

Table 2: Inputs to the formula for computing latencies for C2M and P2M read/write domains (discussion in §6.1).

$$L_m^{\text{read}} = \text{Constant}_{\text{read}} + QD_{\text{read}} \quad \text{(Average read latency)}$$

$$QD_{\text{read}} = O_{\text{RPQ}} \cdot \frac{\#\text{switches}}{\text{lines}_{\text{read}}} \cdot t_{\text{WTR}} \quad \text{(Switching Delay)}$$

$$+ O_{\text{RPQ}} \cdot \frac{\text{lines}_{\text{written}}}{\text{lines}_{\text{read}}} \cdot t_{\text{Trans}} \quad \text{(Write HoL blocking)}$$

$$+ (O_{\text{RPQ}} - 1) \cdot t_{\text{Trans}} \quad \text{(Read HoL blocking)}$$

$$+ \frac{\#\text{ACT}_{\text{read}}}{\text{lines}_{\text{read}}} \cdot t_{\text{ACT}} + \frac{\#\text{PRE}_{\text{read}}^{\text{conflict}}}{\text{lines}_{\text{read}}} \cdot t_{\text{PRE}} \quad \text{(Top-of-queue delay)}$$

Figure 9: Read domain latency components (inputs defined in Table 2; t_{WTR}, t_{Trans} (transmission delay: time taken to transmit a single cacheline over the memory channel in either direction), t_{ACT} (t_{RCD}) and t_{PRE} (t_{RP}) are standard DRAM timing constraints).

$$L_m^{\text{write}} = \text{Constant}_{\text{write}} + AD_{\text{write}} \quad \text{(Average write latency)}$$

$$AD_{\text{write}} = P_{\text{fill}}^{\text{WPQ}} \cdot X_{\text{write}}$$

$$X_{\text{write}} = N_{\text{waiting}} \cdot \frac{\#\text{switches}}{\text{lines}_{\text{written}}} \cdot t_{\text{RTW}} \quad \text{(Switching Delay)}$$

$$+ N_{\text{waiting}} \cdot \frac{\text{lines}_{\text{read}}}{\text{lines}_{\text{written}}} \cdot t_{\text{Trans}} \quad \text{(Read HoL blocking)}$$

$$+ (N_{\text{waiting}} - 1) \cdot t_{\text{Trans}} \quad \text{(Write HoL blocking)}$$

$$+ \frac{\#\text{ACT}_{\text{write}}}{\text{lines}_{\text{written}}} \cdot t_{\text{ACT}} + \frac{\#\text{PRE}_{\text{write}}^{\text{conflict}}}{\text{lines}_{\text{written}}} \cdot t_{\text{PRE}} \quad \text{(Top-of-queue delay)}$$

Figure 10: Write domain latency components (inputs defined in Table 2; t_{WTR}, t_{Trans} (transmission delay: time taken to transmit a single cacheline over the memory channel in either direction), t_{ACT} (t_{RCD}) and t_{PRE} (t_{RP}) are standard DRAM timing constraints).

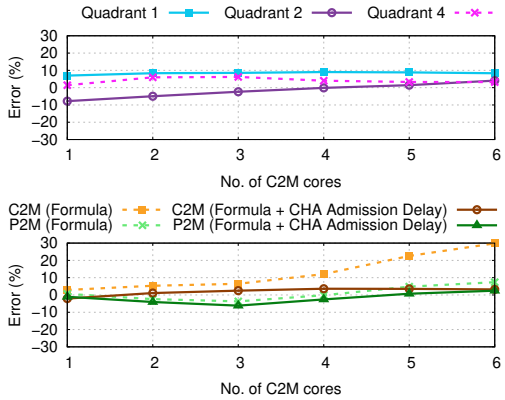

Figure 11: Accuracy of the analytical formulae: (top) Error in the formula's estimate of C2M throughput for quadrants 1, 2, 4. (bottom) Error in the formula's estimate of C2M and P2M throughput for quadrant 3 (both with/without adding CHA admission delay). Positive values indicate overestimation, and negative values indicate underestimation.

Write Domain Latency. Writes require slightly different analysis than reads: since writes do not have to wait until they are actually issued and processed in DRAM. For the P2M-Write domain, they are completed as soon as they are admitted into the MC WPQ. Thus, P2M-Write domain latency only inflates when the WPQ is filled, at which point requests will have to wait for some time until they are admitted (admission delay (AD_{write})).

Our write domain latency formula (Figure 10) captures AD_{write} via (1) the probability that a request is blocked due to the WPQ being full ($P_{\text{WPQ}}^{\text{fill}}$) and (2) the average waiting time for a request when the WPQ is full (X_{write}). For X_{write}, we use an expression analogous to read queueing delay, with the parameters for reads/writes swapped (since the corresponding components for write processing are exactly the duals of those for reads), and using N_{waiting}, the average number of writes (both C2M and P2M) waiting to be admitted into the queue, instead of O_{RPQ} (since admitting N_{waiting} requests requires processing an equal number of writes to make space in the queue). Unlike P2M writes, C2M writes do not have to wait until they are admitted to the MC. We do not capture inflation of C2M-Write domain latency and assume it to be a constant. We later discuss the implications of doing so.

6.2 Applying the Formula

All formula inputs can be captured or derived using programmable uncore performance counters available on Intel servers [29] using the same measurement methodology as §5. We use MC counters to capture all the inputs except $N_{waiting}$. For $N_{waiting}$, we use counters from the CHA, since this is where requests are backlogged when the MC write queues are full [29].

Applying the formula. We set $Constant_{read/write}$ based on unloaded latencies of domains (§4.2). For any given experiment, we then apply the formula using the measured inputs to obtain the average domain latency. Depending on the workload, we use either the read or the write domain latency expression. For C2M-Read and P2M-Read, we use the read domain latency expression. For P2M-Write, we use the write latency expression. For C2M-ReadWrite, we use the C2M-Read domain latency plus a constant (to account for C2M-Write). After obtaining average domain latency (L), we compute estimated throughput using the expression in §4.

Formula accurately captures end-to-end throughput. Figure 11 shows the error in the formula's estimate of throughput in all of the quadrants from §2.2. The formula captures throughput for the C2M workload (which is what degrades) within 10% error for all data points in quadrants 1, 2, and 4. For quadrant 3, with 4 or fewer cores, the error for both C2M/P2M is within 10%. However, beyond 4 C2M cores, the error increases (especially for C2M throughput). This is because our formula currently does not account for admission delay to the CHA, which manifests when CHA buffers get filled up as is the case for quadrant 3 with 4 or more C2M cores and causes latency inflation for both C2M and P2M domains (§5.2)—as a result, the formula underestimates latency leading to overestimation of throughput. When we add the measured CHA admission delay from the testbed to the output of the formula, the error reduces to $< 10\%$ for all data points, thus validating our understanding.

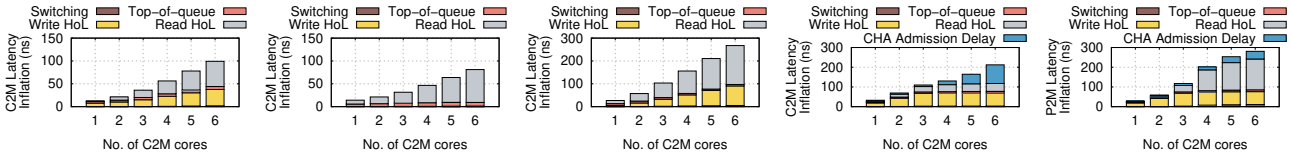

Figure 12: Breakdown of analytical formula components (a-e, left-right) (a, b, c) Breakdown of formula components for C2M in quadrants 1, 2, and 4 respectively; (d, e) Breakdown of formula components for C2M and P2M along with CHA admission delay in quadrant 3.

Breakdown of formula components. Figure 12 presents the breakdown of queueing delay into each of the individual formula components for all of the quadrants. For quadrant 1, with a single C2M core, WriteHoL is the dominant contributor. With increasing C2M cores, both WriteHoL and ReadHoL increase. Quadrant 2 has a larger Read HoL component due to higher average read queue occupancy, but it has no Write HoL component because there are no writes. In quadrant 4, ReadHoL is the dominant contributor for all data points. In quadrant 3, for C2M, WriteHoL is the dominant contributor up to 4 C2M cores, beyond which CHA admission delay starts to dominate. For P2M, WriteHoL is the dominant factor until 3 C2M cores, after which ReadHoL becomes dominant.

7 DISCUSSION AND FUTURE DIRECTIONS

Technology trends for the host hardware suggest that performance of peripheral interconnects is improving much more rapidly than processor and memory interconnects. This has led to an increasing imbalance of resources and contention within the host network, which in turn, negatively impacts application-level performance. We have presented a conceptual abstraction of domain-by-domain credit-based flow control that precisely captures the interplay between processor, memory, and peripheral interconnects within the host network. Using this abstraction, we have built an in-depth understanding of contention within the host network and its impact on application performance reported by previous studies, as well as identified new, previously unreported, regimes of contention within the host network. Our study opens up several interesting avenues of future research at the intersection of computer networking, operating systems and computer architecture. We outline some of these below.

Building an even deeper understanding of the host network. For instance, we focus on a simple setup: two generations of Intel processors with C2M and P2M apps contending on host network resources within the same socket, peripheral devices connected to a single IIO, and all peripheral transfers executed with DDIO disabled. A natural next step is to extend our study to hosts with multiple sockets, multiple IIOs, modern direct cache access mechanisms, and with a wider variety of Intel and AMD processors. Looking forward, the host network in modern datacenter hosts is becoming increasingly complex with new interconnects such as CXL and NVLink, deeper topologies with PCIe switches, different kinds of memory such as High-Bandwidth Memory, and new kinds of hardware accelerators and data movement engines. More work needs to be done to understand the behavior of the host network for such hosts.

Our analytical formula in §6 quantitatively validates our analysis. There are several possible extensions. First, our analytical formula requires inputs that are measured; it would be interesting to build an analytical model that can predict performance given a particular host network hardware configuration (*e.g.*, by extending [5]). Second, while we precisely capture the impact of contention within the host network on application performance for our storage and RoCE/PFC experiments, such is not the case for the lossy network setup—here, packet drops and resulting congestion response lead to complex modeling issues. Incorporating the behavior of end-to-end datacenter-level transport protocols within the analytical model is an important extension of our work. Finally, it would be interesting to build host network simulators that enable a deeper exploration of domain-by-domain credit-based flow and the host network in general.

Rearchitecting protocols, operating systems and host hardware. The host network has implications that are more far-reaching than the context of networked applications and datacenter congestion control—it impacts application-level performance even when all applications are contained within a single host. Our study thus opens up many interesting avenues of future research in the design of protocols, operating systems and even host hardware along several directions. For instance, it would be interesting to explore new mechanisms for host network resource allocation (*e.g.*, extending ideas in hostCC [2] to the case of all traffic contained within a single host), new memory controller scheduling mechanisms to better isolate C2M/P2M traffic (*e.g.*, extending ideas in heterogeneous memory scheduling architectures [6, 33, 34]), and new datapaths for peripheral traffic (*e.g.*, using dynamic direct cache access [3, 68], or even bypassing memory read/writes altogether [27, 65]).

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Michio Honda, the SIGCOMM reviewers, Qizhe Cai, and Shreyas Kharbanda for their insightful feedback. This research was in part supported by NSF grants CNS-2047283 and CNS-2212193, a Sloan fellowship, gifts from Intel and Google, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work does not raise any ethical concerns.

REFERENCES

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. 2022. Understanding Host Interconnect Congestion. In *ACM HotNets*.
[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *ACM SIGCOMM*.
[3] 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 *IEEE MICRO*.
[4] Anthony Alles. 1995. ATM Internetworking. In *Engineering InterOp*.
[5] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. 2023. Formal Methods for Network Performance Analysis. In *USENIX NSDI*.

[6] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ACM SIGARCH Computer Architecture News*.

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

[8] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. http://arxiv.org/abs/1508.03619

[9] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. 2022. dcPIM: Near-Optimal Proactive Datacenter Transport. In *ACM SIGCOMM*.

[10] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *ACM SIGCOMM*.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. In *IEEE Data Engineering Bulletin*.

[12] Justin Castilla. 2024. Clustering In Redis. https://developer.redis.com/operate/redis-at-scale/scalability/lustering-in-redis/.

[13] Robert Cole, David Shur, and Curtis Villamizar. 1996. IP Over ATM: A Framework Document. https://datatracker.ietf.org/doc/html/rfc1932.

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. In *Communications of the ACM*.

[15] Travis Downs. 2018. It's not write combining. https://github.com/Kobzol/hardware-effects/issues/1.

[16] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. 2010. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-core Memory Systems. In *ACM SIGPLAN Notices*.

[17] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. In *ACM SIGMOD Record*.

[18] 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 *USENIX ATC*.

[19] Cache Forge. 2024. memcached - A Distributed Memory Object Caching System. https://memcached.org/.

[20] Henry J Fowler. 1995. TMN-Based Broadband ATM Network Management. In *IEEE Communications Magazine*.

[21] Alex Galis, Dieter Gantenbein, Stefan Covaci, Carlo Bianza, Fotis Karayannis, and George Mykoniatis. 1996. Toward Multidomain Integrated Network Management for ATM and SDH Networks. In *Broadband Strategies and Technologies for Wide Area and Local Access Networks*.

[22] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. pHost: Distributed Near-Optimal Datacenter Transport Over Commodity Network Fabric. In *ACM CoNEXT*.

[23] Saugata Ghose, Hyodong Lee, and José F Martínez. 2013. Improving Memory Scheduling via Processor-Side Load Criticality Information. In *ACM/IEEE ISCA*.

[24] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX OSDI*.

[25] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *ACM SIGCOMM*.

[26] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. 2020. Aeolus: A Building Block for Proactive Transport in Datacenters. In *ACM SIGCOMM*.

[27] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The NanoPU: A Nanosecond Network Stack for Datacenters. In *USENIX OSDI*.

[28] Intel. 2012. Intel Data Direct I/O Technology (Intel DDIO): A Primer. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf.

[29] Intel. 2017. Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring. https://kib.kiev.ua/x86docs/Intel/PerfMon/336274-001.pdf.

[30] Intel. 2023. Intel 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200.

[31] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *ACM SIGMETRICS Performance Evaluation Review*.

[32] Raj Jain. 1996. Congestion Control and Traffic Management in ATM Networks: Recent Advances and a Survey. In *Computer Networks and ISDN systems*.

[33] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *ACM/IEEE DAC*.

[34] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *IEEE MICRO*.

[35] Yoongu Kim. 2015. Architectural Techniques to Enhance DRAM Scaling. https://kilthub.cmu.edu/articles/thesis/Architectural_Techniques_to_Enhance_DRAM_Scaling/7461695/1.

[36] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *IEEE HPCA*.

[37] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *IEEE MICRO*.

[38] HT Kung and Alan Chapman. 1993. The FCVC (Flow-Controlled Virtual Channels) Proposal for ATM Networks: A Summary. In *IEEE ICNP*.

[39] NT Kung and Robert Morris. 1995. Credit-Based Flow Control for ATM Networks. In *IEEE Network*.

[40] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX OSDI*.

[41] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2010. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. https://utw10235.utweb.utexas.edu/people/cjlee/TR-HPS-2010-002.pdf.

[42] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. 2023. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In *USENIX FAST*.

[43] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*.

[44] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. 2023. Hostping: Diagnosing Intra-Host Network Bottlenecks in RDMA Servers. In *USENIX NSDI*.

[45] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD*.

[46] John D McCalpin. 1995. STREAM: Sustainable Memory Bandwidth in High Performance Computers. https://www.cs.virginia.edu/stream/.

[47] David J Miller, Philip M Watts, and Andrew W Moore. 2009. Motivating Future Interconnects: A Differential Measurement Analysis of PCIe Latency. In *ACM/IEEE ANCS*.

[48] Thomas Moscibroda and Onur Mutlu. 2008. Distributed Order Scheduling and its Application to Multi-Core DRAM Controllers. In *ACM PODC*.

[49] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. 2011. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *IEEE MICRO*.

[50] Onur Mutlu and Thomas Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *IEEE MICRO*.

[51] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ACM/IEEE ISCA*.

[52] Thomas Moscibroda Onur Mutlu. 2007. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security*.

[53] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. 2006. Fair Queuing Memory Systems. In *IEEE MICRO*.

[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 *ACM SIGCOMM*.

[55] George P Nychis, Chris Fallin, Thomas Moscibroda, Onur Mutlu, and Srinivasan Seshan. 2012. On-Chip Networks From a Networking Perspective: Congestion and Scalability in Many-Core Interconnects. In *ACM SIGCOMM*.

[56] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*.

[57] Redis. 2024. Redis. http://www.redis.io.

[58] Redis. 2024. Redis Benchmark. https://redis.io/docs/management/optimization/benchmarks/.

[59] Bryan Spry, Nagi Aboulenein, and Steve Kulick. 2008. United States Patent Application Publication: Mechanism for Write Optimization to a Memory Device. https://patentimages.storage.googleapis.com/53/bf/04/667faa6c4e5278/US20080162799A1.pdf.

[60] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. In *Data Engineering Bulletin*.

[61] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of Temporary Storage in the NodeKernel Architecture. In *USENIX ATC*.

[62] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. 2014. The Blacklisting Memory Scheduler: Achieving High Performance

and Fairness at Low Cost. In *IEEE ICCD*.

[63] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *IEEE MICRO*.

[64] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. 2013. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *IEEE HPCA*.

[65] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NEBULA RPC-Optimized Architecture. In *ACM/IEEE ISCA*.

[66] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*.

[67] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *IEEE MICRO*.

[68] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan RK Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. 2023. RAMBDA: RDMA-Driven Acceleration Framework for Memory-Intensive μs-scale Datacenter Applications. In *IEEE HPCA*.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*.

[70] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality. In *IEEE MICRO*.

[71] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *ACM/IEEE ISCA*.

[72] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *ACM SIGCOMM*.

A QUADRANT 2 AND 4 RESULTS

Figure 13, and Figure 14 show measurements similar to those presented in §5 for Quadrants 2 and 4 respectively.

In Quadrant 2, similar to Quadrant 1, we observe C2M throughput degradation before memory bandwidth saturation. This is again due to a combination of row miss ratio increase (Figure 13(c)) and bank load imbalance leading to queueing at the memory controller (Figure 13(b)), causing domain latency inflation (Figure 13(a)) and ultimately C2M throughput degradation. P2M traffic in Quadrant 2 performs reads, and thus observes latency inflation due to queueing at the memory controller, similar to C2M (since the P2M-Read domain includes DRAM). Yet, its throughput does not degrade because there are enough spare domain credits available that allow it to tolerate inflation in domain latency. Moreover, the spare credit capacity is even larger for the P2M read domain (§4)—we observe that the number of in-flight P2M reads[4] is far below this limit even with the maximum number of colocated C2M cores (Figure 13(d)).

In Quadrant 4, C2M throughput degrades even though memory bandwidth is not saturated due to the same reason as in Quadrants 1 and 2—latency inflation (Figure 14(a)) due to queueing (Figure 13(b)) caused by a combination of an increase in row misses (Figure 14(c)) and load imbalance across banks. P2M throughput does not degrade for the same reason as in Quadrant 2—there are enough spare domain credits to tolerate the domain latency inflation—the measured number of in-flight P2M reads is well below the credit limit even with the maximum number of C2M cores (Figure 14(d)).

B APPLICATION RESULTS WITH DIFFERENT READ/WRITE RATIOS

The experiments with real applications presented in §2.1 capture the interaction between C2M reads and P2M writes. In this section, we extend these experiments to show all possible C2M/P2M read/write combinations for completeness.

For each of the C2M applications (Redis, GAPBS), in addition to the read-heavy workloads used in §2.1 (which we call Redis-Read and GAPBS-PR), we introduce corresponding workloads that generate C2M writes: Redis-Write and GAPBS-BC. In Redis-Write, the clients issue 100% set queries with all other parameters same as those used in §2.1. This workload generates $\sim$ 50% read 50% write C2M traffic on average and is slightly more memory intensive compared to Redis-Read (i.e., more memory bandwidth usage for same number of cores). In GAPBS-BC, we use the same GAPBS setup as in §2.1 but instead of using the page rank (PR) algorithm, we use the betweenness centrality (BC) algorithm. This algorithm is write-heavy during some iterations (performing random writes), and read-heavy during others. On average, it generates $\sim$ 80% read 20% write traffic (this is the most write-heavy algorithm available in the GAPBS suite), and is more compute intensive and less memory intensive compared to the page rank algorithm (i.e., smaller memory bandwidth usage per-core). For the P2M apps, we use the FIO-based P2MWrite and P2MRead workloads from §2.1.

We find that the performance degradation trends reported in §2.1 (Figure 2) generalize across all C2M/P2M and read/write combinations—as is evident in Figures 15,16,17, C2M applications suffers performance degradation (even when memory bandwidth is not saturated) while P2M application performance remains unaffected. The magnitude of C2M performance degradation generally correlates with the memory-intensity of the C2M workload. For a fixed P2M workload, and fixed number of cores, Redis-Write observes higher performance degradation than Redis-Read, and GAPBS-BC observes lower performance degradation than GAPBS-PR (an exception to this trend is the single core data point in Figure 15, where GAPBS-BC see higher performance degradation compared to what GAPBS-PR sees in Figure 2 when DDIO is on).

Similar to the observations in §2.1 (Figure 2), when the C2M apps are colocated with P2M Write, DDIO results in worse C2M app performance degradation (Figure 15). In contrast, when the C2M apps are colocated with P2M Read, the observed C2M app performance degradation is identical with/without DDIO (Figures 16, 17). With DDIO, the key difference between P2M Write and P2M Read is that in the former case, misses result in LLC allocations (which in turn can cause cache evictions), while in the latter case they do not [18]. Therefore, these observations provide some evidence that cache evictions are the cause of higher C2M degradation with DDIO. What remains unclear is why evictions have such a significant impact even though the C2M apps are not LLC-bound (i.e., miss ratio is near 100% even when the workload is run in isolation). As pointed out in §2, this requires further investigation.

C HOST NETWORK CONTENTION REGIMES: NETWORKING CASE STUDIES

Recent studies [1, 2, 42, 44] report the impact of host network contention on network traffic but lack a comprehensive characterization and in-depth understanding of the root causes of the problem. In this section, we extend our characterization of host network contention regimes (§2) to RDMA and TCP case studies.

C.1 RDMA

Our setup consists of two servers with RDMA-enabled NICs (Nvidia/Mellanox ConnectX-5) connected directly to each other through a 100Gbps network link. Each of the servers has the same Cascade Lake setup as in §2. We use standard RDMA configuration of RoCE v2 with PFC enabled and MTU size of 4096. RDMA traffic is generated using the standard `ib_write_bw` and `ib_read_bw` tools from the RDMA perftest benchmarking suite. The former generates P2M write traffic and the latter generates P2M read traffic on the server-side. On the server-side, we colocate C2MRead and C2MReadWrite workloads from §2. We disable CPU prefetchers and DDIO for better explainability.

With the above setup, we conduct experiments for all C2M/P2M read/write combinations analogous to the four quadrants in §2.2. Figure 18 shows the corresponding results. We observe the same performance degradation trends as in §2.2. In particular, Quadrants 1, 2, 4 exhibits the blue regime (where C2M degrades but P2M does not), and Quadrant 3 exhibits the red regime (where both C2M and P2M throughput degrade). The magnitudes of C2M throughput degradation (in Quadrants 1-4) and P2M throughput degradation (in Quadrant 3) are slightly lower that what is observed in §2.2. This is because the NIC generates a slightly lower P2M load than the SSDs

[4]We were not able to directly measure the IIO read buffer occupancy. Instead, we measure the number of in-flight P2M read requests at the CHA which is a lower-bound on the total number of in-flight requests in the P2M Read domain.

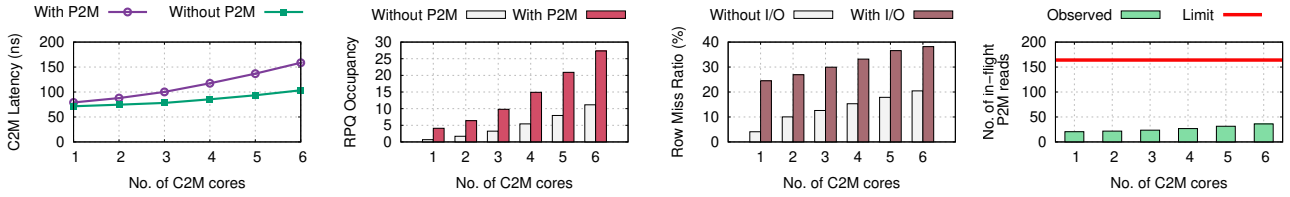

Figure 13: Results for understanding quadrant 2 (a-d, left-right). All y-axis values are on average. Discussion in Appendix A.

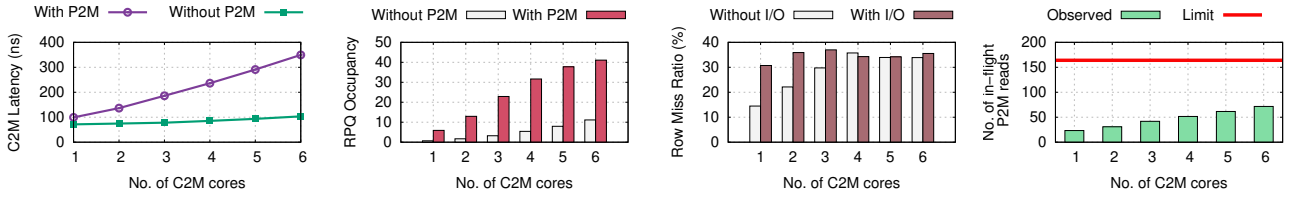

Figure 14: Results for understanding quadrant 4 (a-d, left-right). All y-axis values are on average. Discussion in Appendix A.

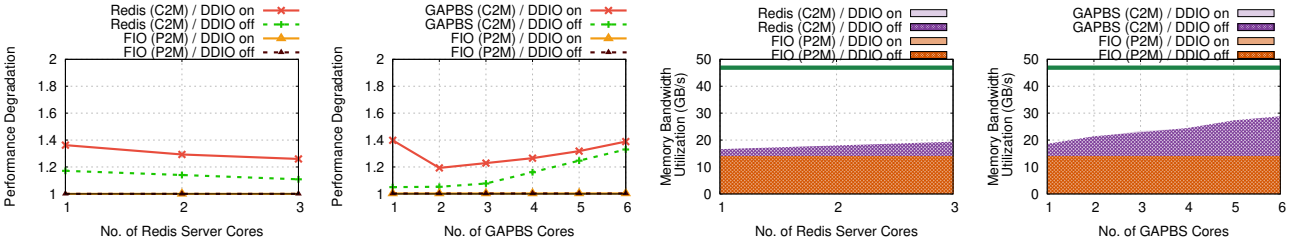

Figure 15: Results with each of Redis-Write and GAPBS-BC (C2M-ReadWrite) colocated with P2M write. (Discussion in Appendix B)

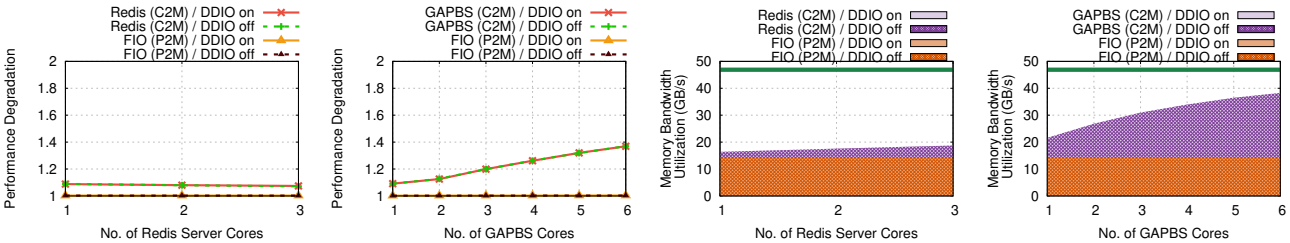

Figure 16: Results with each of Redis-Read and GAPBS-PR (C2M-Read) colocated with P2M read. (Discussion in Appendix B)

(with no memory contention, the SSDs generate ~112Gbps, while the NIC generates ~98Gbps).

C.2 DCTCP

In this section, we present results for the TCP case study. While we observe the same underlying C2M/P2M degradation regimes as in previous SSD and RDMA experiments, the resulting impact on application throughput is different. This is because, with TCP, (1) the network application generates C2M traffic in addition to P2M traffic due to data copy between application buffers and socket buffers that is executed on CPU cores (2) P2M throughput degradation can result in buffer overflow and packet drops at the receiver-side NIC triggering congestion control response at the sender-side.

We use similar setup as in [2]: Two servers each with Cascade Lake setup (§2) are connected via 100Gbps link. Both run Linux kernel version 5.4.0 with DCTCP and all optimizations enabled (TSO/GRO, aRFS, and Jumbo frames with 9K MTU). We send traffic from one server to another by running the standard `iperf` tool on 4 CPU cores on the sender and receiver each (this is sufficient to saturate 100Gbps link when there is no memory contention). Each sender core sends one long flow to its corresponding receiver core. Similar to [2], we focus on the receiver-side (Rx) as that is where the key bottlenecks are [10]. We colocate the C2MRead and C2MReadWrite workloads on the receiver-side server where P2M write traffic is generated by the NIC. Hardware prefetchers and DDIO are kept disabled for better explainability.

C2MRead + TCP Rx. Figure 19 (a, b) show the results of co-locating C2M Read (Memory app) on the TCP receiver-side. Both the Memory app and Network app observe throughput degradation. The magnitude of degradation for the memory app is relatively

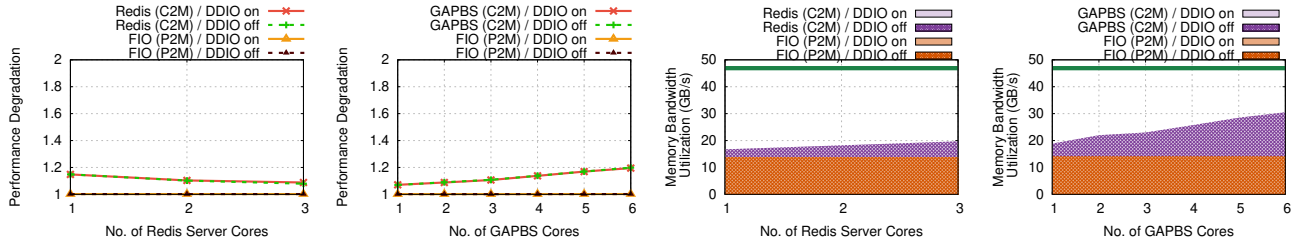

Figure 17: Results with each of Redis-Write and GAPBS-BC (C2M-ReadWrite) colocated with P2M read. (Discussion in Appendix B)

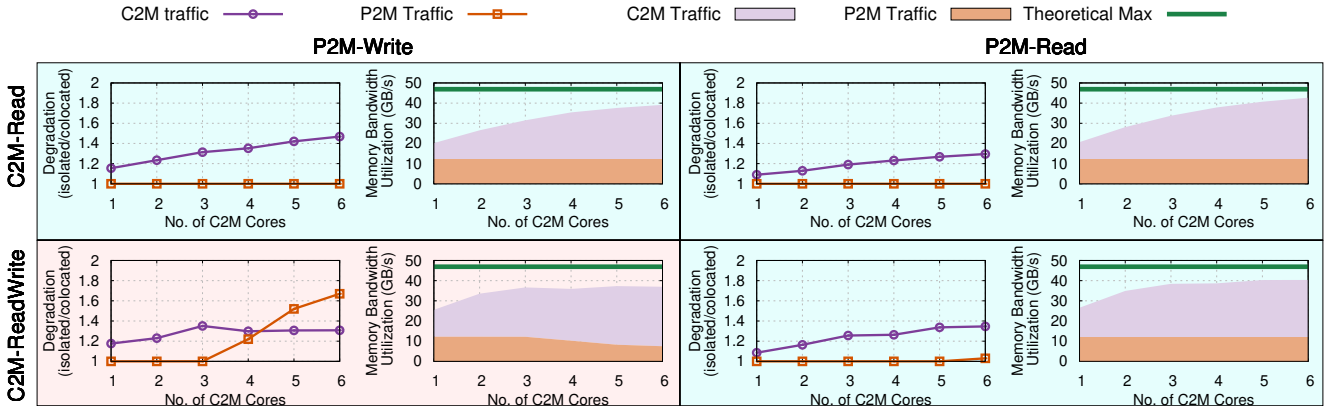

Figure 18: Blue and red regimes across four quadrants in the RDMA case study (1-4; clockwise) Each quadrant is shaded with the color of the regime it shows. For each quadrant, (left) shows throughput degradation observed by C2M and P2M workloads (ratio of throughput when run in isolation to the throughput when colocated) and (right) shows memory bandwidth utilization when they are colocated broken down by C2M and P2M traffic.

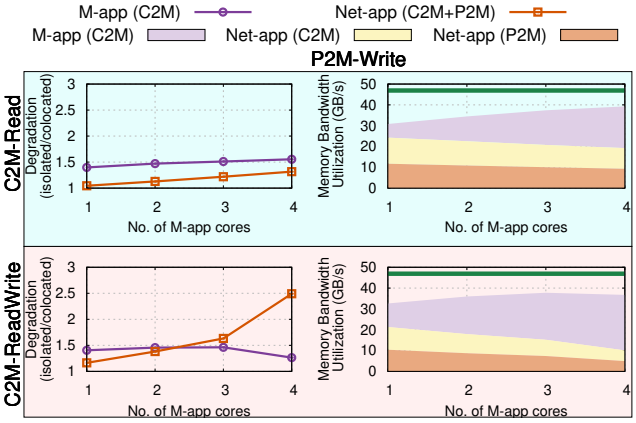

Figure 19: TCP case study results (a-d; clockwise). (a, b) respectively show throughput degradation observed when C2MRead and TCP Rx are colocated, and corresponding breakdown of memory bandwidth utilization (c, d) respectively show throughput degradation observed when C2MReadWrite and TCP Rx are colocated, and corresponding breakdown of memory bandwidth utilization.

higher. With increasing C2M load, the gap between Network app degradation and Memory app degradation reduces.

C2MReadWrite + TCP Rx. Figure 19 (c, d) show the impact of co-locating C2MReadWrite on the TCP receiver-side. Both Memory app and Network app observe throughput degradation. At lower load (2 or fewer C2M cores), Memory app observes larger degradation relative to the Network app. At higher load (beyond 2 C2M cores), the Network app observes larger throughput degradation relative to the Memory app, reaching up to ~2.5× degradation with 4 C2M cores.

D UNDERSTANDING CONTENTION WITHIN THE HOST NETWORK: NETWORKING CASE STUDIES

D.1 RDMA

Applying the methodology from §5 allows us to precisely explain all the above observed performance degradation trends. We discuss each of the regimes below.

Blue regime. In Quadrant 1 (C2M-Read, P2M-Write), C2M throughput degrades because C2M domain credits are fully utilized and domain latency increases (Figure 20(a)) due to queueing at the memory controller (Figure 20(b)) as C2M Read domain includes DRAM execution. P2M throughput does not degrade because (1) at low C2M load, P2M write latency does not increase (Figure 20(d)), since P2M write domain does not include DRAM execution, and the memory controller write queue does not get filled (Figure 20(e)) and (2) at higher C2M loads, even when there is a slight increase in P2M write latency (~15ns; which is not visible in the Figure 20(d)), the P2M write domain has enough spare credits to mask this latency inflation (Figure 20(f)). The explanations for Quadrants 2 and 4 from §5 similarly generalize as well, and the corresponding results are shown in Figure 21, 24.

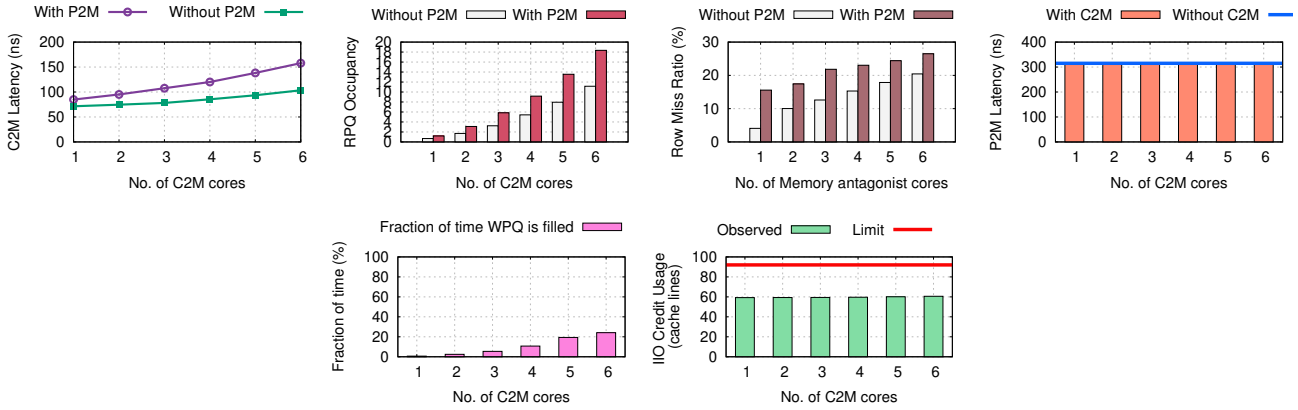

Figure 20: Results for understanding Quadrant 1 in RDMA case study (a-f, clockwise). All y-axis values are on average.

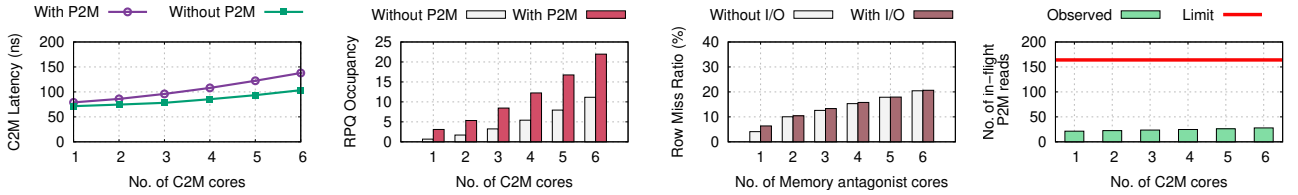

Figure 21: Results for understanding Quadrant 2 in RDMA case study (a-d, clockwise). All y-axis values are on average.

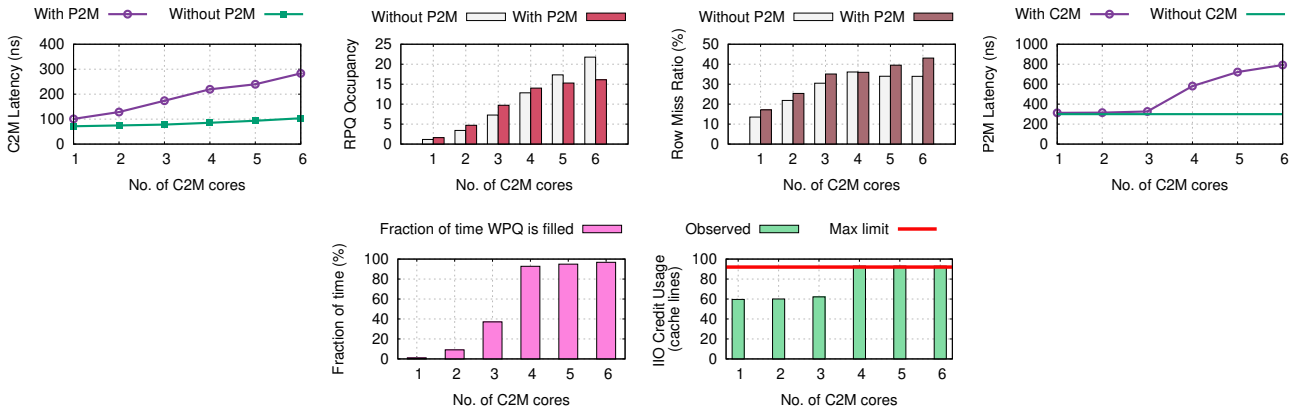

Figure 22: Results for understanding Quadrant 3 in RDMA case study (a-f, clockwise). All y-axis values are on average.

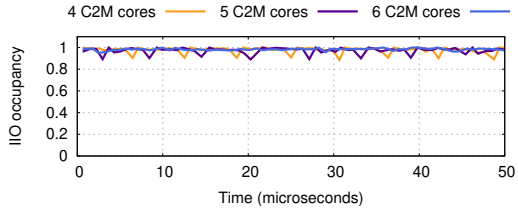

Figure 23: Microsecond-scale measurements of IIO buffer occupancy in Quadrant 3 of RDMA case study.

Red regime. In Quadrant 3 (C2M-ReadWrite, P2M-Write), with 3 or fewer C2M cores, C2M throughput degrades because of domain latency inflation (Figure 22(d)) and P2M throughput does not degrade due to either no P2M latency inflation (Figure 22(d)) or P2M domain having enough space credits (Figure 22(f)). With 4 or more C2M cores, P2M write latency increase significantly (Figure 22(d)) due to memory controller write queue being almost always full (Figure 22(e)) which has one-sided impact on P2M write domain. In conjunction, P2M write domain credits get exhausted (Figure 22(f)) thus leading to P2M throughput degradation. This results in queue build-up at the NIC which triggers PFC pauses—we observe PFC pause fractions of 22%, 39%, 43% with 4, 5, 6 C2M cores respectively. For each of these data points, Figure 23 shows microseconds-scale measurements of the IIO buffer occupancy. We observe that the occupancy remains close to full capacity throughout indicating that PFC is able to maintain enough in-flight data in the NIC queue in order to keep the IIO buffer full.

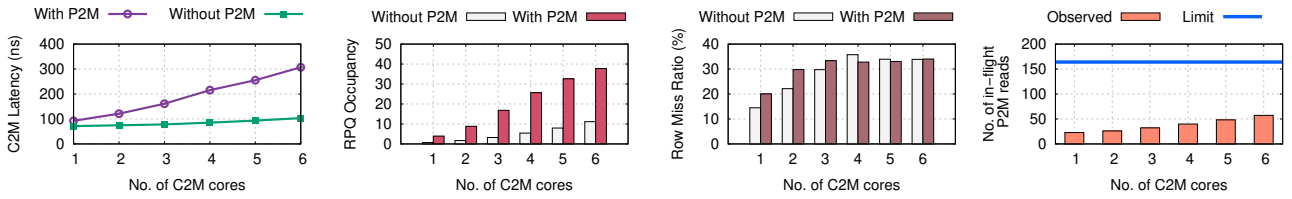

Figure 24: Results for understanding Quadrant 4 in RDMA case study (a-d, clockwise). All y-axis values are on average.

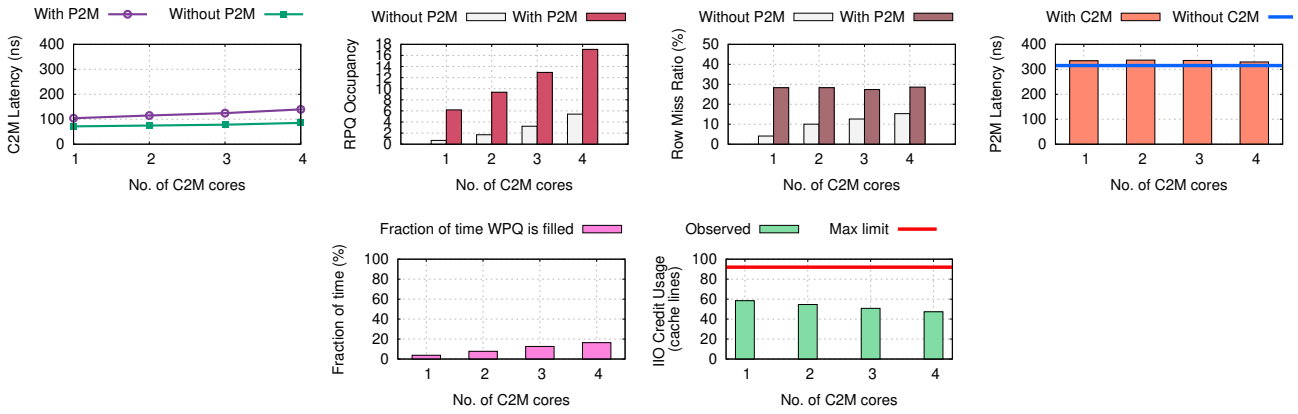

Figure 25: Results for understanding C2MRead + TCP Rx in TCP case study (a-f, clockwise). All y-axis values are on average.

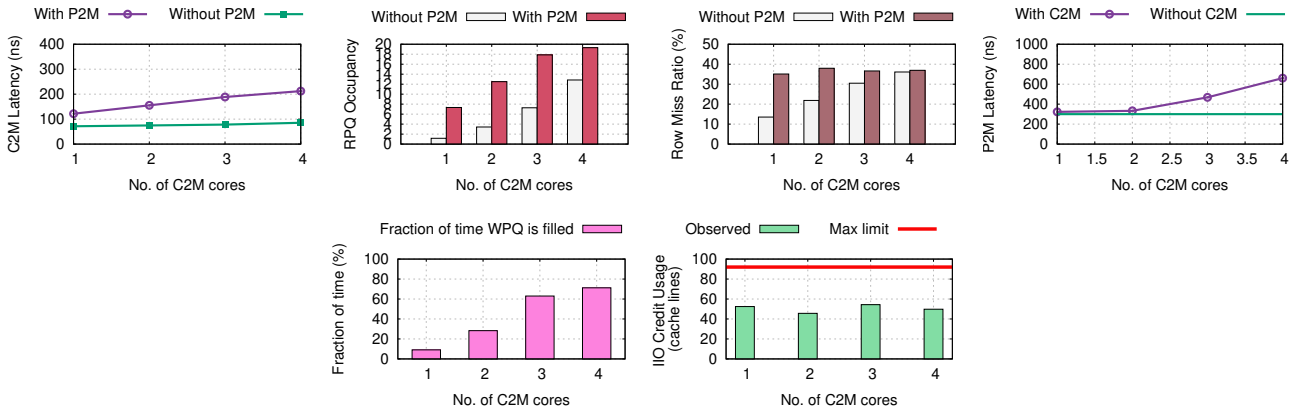

Figure 26: Results for understanding C2MReadWrite + TCP Rx in TCP case study (a-f, clockwise). All y-axis values are on average.

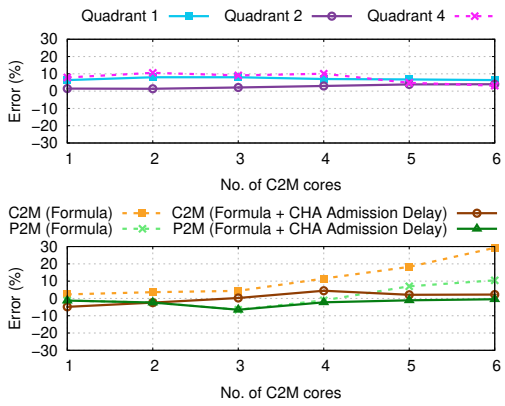

Figure 27: Accuracy of the analytical formulae in capturing observed C2M and P2M throughput in the RDMA case study: Positive values indicate overestimation, and negative values indicate underestimation.

D.2 DCTCP

We now apply our methodology from §5 to analyze the performance degradation trends in the TCP case study.

Blue Regime. The observations in C2MRead + TCP Rx are explained by C2M Read domain latency inflation (Figure 25(a)), due to queueing at the memory controller read queue (Figure 25(b)). In addition to causing degradation for the memory app, increase in C2M Read domain latency leads to CPU being bottlenecked for the network app, due to data copy slowing down, thus causing

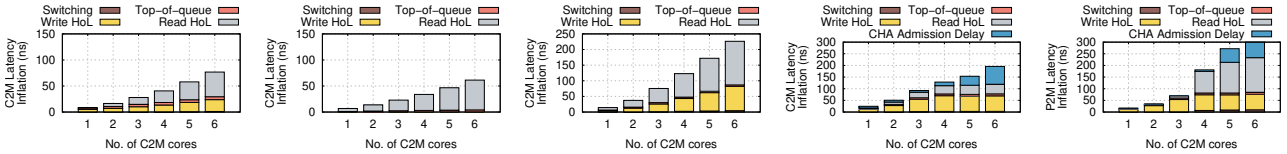

Figure 28: Breakdown of analytical formula components in the RDMA case study (left-right) Breakdown of formula components for C2M in Quadrants 1, 2, and 4 followed by breakdown of formula components for C2M and P2M in Quadrant 3 in that order.

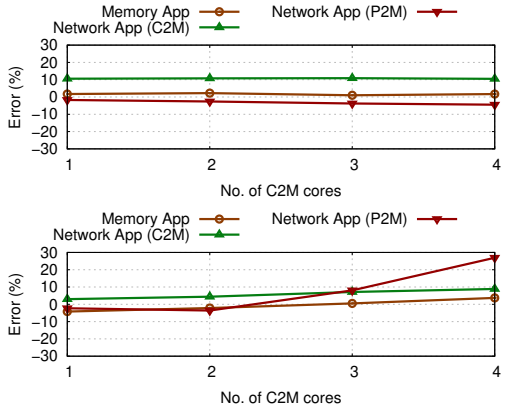

Figure 29: Accuracy of the analytical formulae in capturing Memory app and Network app throughput in the TCP case study: Positive values indicate overestimation, and negative values indicate underestimation.

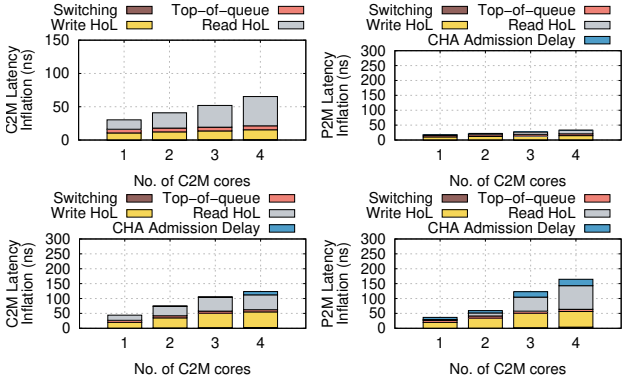

Figure 30: Breakdown of formula components when applied to TCP case study. (top-left) C2M latency breakdowns for C2MRead + TCP Rx (top-right) P2M latency breakdown for C2MRead + TCP Rx (bottom-left) C2M latency breakdown for C2MReadWrite + TCP Rx (bottom-right) P2M latency breakdown for C2MReadWrite + TCP Rx.

throughput degradation. The throughput degradation is larger for the memory app since it is relatively more memory-intensive (performing memory reads all the time, in contrast to network app which spends ~50% of it's time doing processing other than data copy when run in isolation [10]). Indeed, with increasing C2M load, the gap between network app degradation and memory app degradation reduces as data copy consumes an increasing fraction of CPU cycles for the Network app. P2M write latency inflation is negligible (Figure 25(d)) because the memory controller write queues get filled for only a small fraction of time (Figure 25(e)). Due to the CPU bottleneck at the receiver, TCP flow-control reduces the sending rate. With increasing C2M load, the rate becomes increasingly lower. Consequently, the P2M write domain can sustain the given rate while utilizing fewer credits; hence the average IIO occupancy reduces with increasing C2M load (Figure 25(f)).

Red Regime. In C2MReadWrite + TCP Rx case, similar to the C2MRead + TCP Rx case, degradation with 1 and 2 C2M cores is explained by C2M Read domain latency inflation which impacts both the memory app and network app (due to data-copy processing). At higher load (beyond 2 C2M cores), the memory controller write queues start getting filled up for large fractions of time (Figure 26(e)). This results in black log of writes at the CHA which impacts the P2M Write domain (but not the C2M Write domain), thus resulting in significant P2M write latency inflation (Figure 26(d)) causing P2M throughput degradation, resulting in packet drops at the NIC and congestion control response at the sender-side. We observe a packet loss rate of 0.02% with 3 C2M cores, and 0.36% with 4 C2M cores. This translates to throughput degradation for the Network app. The precise throughput observed by the Network app depends on the congestion control behavior. (Note that the IIO occupancy measurements in Figure 26(f) are long-term average values—these are difficult to directly reason about since congestion control response results in dynamic P2M load at RTT timescales).

E QUANTITATIVE VALIDATION: NETWORKING CASE STUDIES

E.1 RDMA

To close the loop in explaining the observations, we apply our analytical formula from §6 (using the same methodology) to all the experiments in the RDMA case study. As shown in Figure 27, the formula captures both C2M and P2M throughput within 6.5% accuracy for all data points across all the quadrants. Figure 28 shows the corresponding breakdowns of formula components; the trends match those observed in Figure 12 for the SSD experiments.

E.2 TCP

We apply our analytical formula from §6 to capture Memory app and Network app throughput in the TCP case study experiments. Since the Network app throughput depends on both C2M throughput and P2M throughput, we compute and show each of them separately. To compute Network app C2M throughput, we divide the measured average LFB occupancy of the Network app cores by the C2M latency value obtained by applying the formula. Similarly, for P2M throughput, we divide the measured average IIO occupancy by the P2M latency value obtained by applying the formula.

Figure 29 (top) shows that we are able to capture both Memory app and Network app C2M and P2M throughput for all data points

within 10% error for the C2MRead + TCP Rx case. Figure 29 (bottom) shows the results for the C2MReadWrite + TCP Rx case. The formula captures Memory app throughput and Network app C2M throughput within 10% across all data points. Network app P2M throughput is within 10% error for all data points except 4 C2M cores, where we see ∼26% error. At this data point, we observe relatively high packet loss rate (0.36%); therefore congestion control behavior plays a dominant role in determining Network app throughput. Precisely understanding and modeling the impact of such congestion control behavior (as explored in [2]) on application throughput is an interesting avenue for future work.