Just In Time Delivery: Leveraging Operating Systems Knowledge for Better Datacenter Congestion Control

Amy Ousterhout
MIT CSAIL

Adam Belay
MIT CSAIL

Irene Zhang
Microsoft Research

Abstract

Network links and server CPUs are heavily contended resources in modern datacenters. To keep tail latencies low, datacenter operators drastically overprovision both types of resources today, and there has been significant research into effectively managing network traffic [4, 19, 21, 29] and CPU load [22, 27, 32]. However, all of this work looks at the two resources in isolation.

In this paper, we make the observation that, in the datacenter, the allocation of network and CPU resources should be *co-designed* for the most efficiency and the best response times. For example, while congestion control protocols can prioritize traffic from certain flows, it is wasted work if the traffic arrives at an overloaded server that will only enqueue the request. Likewise, allocating more CPU resources to an application will not improve its performance if network congestion is causing a bottleneck in its requests.

This paper explores the potential benefits of such a co-designed resource allocator and considers the recent work in both CPU scheduling and congestion control that is best suited to such a system. We propose a *Chimera*, a new datacenter OS that integrates a receiver-based congestion control protocol with OS insight into application queues, using the recent Shenango operating system [32].

1 Introduction

In modern datacenters, user requests must traverse many distributed nodes and network links to complete. For example, constructing a Facebook page may require contacting up to 2,000 memcached nodes [31]. If any of these links are congested, or if requests must wait before being handled by a busy CPU core, the user response is delayed.

To keep user response times low, especially the long tail of response times, datacenter operators drastically overprovision both the network and CPU. For example, average link utilization in datacenters over intervals of 1-5 minutes is typically less than 1% [8, 36]. CPUs are similarly under-utilized, operating at utilizations of 10-66% [6, 7, 22, 24–26, 35, 39]. Because a large fraction of the total cost of ownership of a datacenter can be attributed to servers and network links [6], overprovisioning in this way is quite costly.

Much research has explored how to maintain good performance at higher utilization; however, the existing work focuses on only one of the network and the CPU. For example, Heracles [27], PerfIso [22], and Shenango [32] maintain good performance at higher CPU utilization by increasing isolation between colocated applications. In the network domain, several approaches [4, 19, 21, 29] enable networks to be run at higher utilization while preserving good performance, by prioritizing traffic that is presumed to be more urgent (typically shorter flows).

This paper takes a more holistic approach. We argue that networks should not optimize to deliver traffic as fast as possible, but they should aim to deliver it *just in time* for the CPU to process it. Today, a congested network may prioritize a given flow, forcing other flows to queue, only to have that flow arrive at a server that is busy handling other requests. In this case, overall performance would be improved if the network was aware of the server's busyness. Similarly, if the network is congested, a busy CPU should not waste cycles generating a backlog of packets only to have them delayed by the network.

In this paper, we consider the question *How can we co-design congestion control with operating system CPU scheduling to optimize for both utilization and end-to-end response latencies?* We discuss the benefits of such a co-design (§2) and explore the design options in achieving it (§3). Then, we propose *Chimera*, a new datacenter OS (§4). Chimera leverages the recent Shenango operating system [32], which has insight into application-level queues, and integrates it with a receiver-driven congestion control protocol to achieve better end-to-end application latency. We conclude with a discussion of the open problems in Chimera's design and implementation (§5).

2 Benefits of Codesign

Here we identify two opportunities where co-designing congestion control with CPU scheduling could be beneficial.

2.1 Reducing End-to-End Response Latencies

Consider two applications running in two different VMs on the same server. Application A has a short backlog of incoming requests; Application B has a long backlog of requests waiting to be handled. Now consider two requests destined for this server: a large request for A and a small request for B. Any congestion control scheme that implements shortest remaining processing time (SRPT), such as pFabric [4] or Homa [29], will prioritize the small request for B, causing it to arrive at the server first. Assuming that B processes requests in FIFO order, the request will sit in B's queue until B is able to drain the backlog of requests ahead of it. At the same time, the large request for A is needlessly delayed. In this case, switching the order of these two requests would improve the end-to-end response time for the large request and would have no impact on the end-to-end response time of the small request.

This example demonstrates that optimizing flow scheduling for conventional network-centric objectives such as minimizing flow completion time with SRPT [4, 5, 18, 29], meeting flow arrival deadlines [21, 37, 38], or achieving fairness across flows [20] may be a reasonable heuristic in many cases, but can produce suboptimal response times for the end-to-end system.

2.2 Increasing Resource Utilization

Suppose a compute-heavy application, such as Spark or MapReduce, and a network-heavy application, such as a video upload application, run on the same server. We would like to allocate enough network bandwidth to the compute-heavy application in order for it to keep the CPUs on the server busy while leaving all remaining bandwidth available for the network-intensive application. With existing approaches such as Differentiated Services [9], we could strictly prioritize the network traffic of the compute-heavy application over that of the network-heavy application. This would ensure that the compute-heavy application was always able to receive incoming messages with more work to do and could keep its CPUs occupied. However, this may waste bandwidth on requests that will just sit in queues, or worse be cancelled entirely if they take too long to complete [15]. With QJump [19], we could prioritize the compute-heavy traffic while also rate limiting it, but how would we decide on the correct rate limit? Too high of a limit and we return to the situation described above, and too low of a limit would cause CPU cores to sit idle. The best rate likely varies over time depending on the workload.

Ideally applications receiving network traffic would be able to apply backpressure to senders, indicating when they were congested. In TCP, receive windows were originally designed for this purpose, in order to indicate to senders when a receiver did not have enough available memory to accept more traffic. However, servers today typically allocate ample memory for receive socket buffers, and applications rapidly dequeue requests from these buffers and move them to internal queues. Therefore, in practice, receive windows rarely limit transmission rates. Worse still, the amount of data queued in a network socket is only loosely related to the amount of computational work needed to process it. In summary, there is a significant opportunity to apply scheduler information about the busyness of CPUs to adjust network-level flow control.

Today, datacenter network operators address these types of problems by overprovisioning network resources, deploying 10, 40, or 100 Gbits/s links that sit underutilized. If instead networks could prioritize traffic based on applications' ability to handle it, networks could be provisioned with lower capacity, significantly reducing costs.

3 Design Decisions

To achieve these benefits, we propose leveraging OS information about how busy an application is to improve congestion control. Rather than reinvent the wheel, we build upon existing congestion control schemes. Many congestion control algorithms today make explicit decisions about which flows to prioritize over others, but they do so using heuristics about what will produce the best *network*-level performance [4, 5, 17–21, 29], not end-to-end application performance. Instead, we modify these algorithms to use information about the busyness of servers to choose which flows to prioritize. To do so, we must make the following design decisions:

- What types of congestion control schemes are best-suited for this purpose? (§3.1)
- What metric(s) should endhosts use to expose application busyness? (§3.2)
- How can information about server busyness be exposed to congestion control? (§3.3)
- What should congestion control schemes do with this information? (§4.2)

3.1 What Type of Congestion Control?

Congestion control schemes can be broadly grouped into four categories, based on how they decide which flows get to use network resources at any given time:

- **Implicit**: Schemes such at TCP, DCTCP [2], and HULL [3] do not explicitly choose to prioritize one flow over another; flow rates are adjusted based on packet drops or marks.
- **Sender-driven:** Sender-driven schemes set the priorities of packets at sending endhosts, based on application priorities (e.g., QJump [19]) or on flow sizes (e.g., pFabric [4], PIAS [5]). Network switches then enforce these priorities, but the priorities themselves are determined by the senders without input from the network or the receiving endhosts.
- **In-network:** In in-network approaches, senders advertise some information about their demands, and switches along network paths decide what rate to allocate to each flow based on the congestion they observe locally [17, 21, 23, 30].
- **Receiver-driven:** In receiver-driven approaches such as pHost [18], NDP [20], and Homa [29], receiving endhosts decide how to mediate between competing traffic demands from different senders. In these approaches, senders must still make decisions about how to prioritize different flows, for example when two receivers both permit transmissions from the same endhost simultaneously. However, only in receiver-driven approaches do the receivers participate in prioritization at all.

Of these four types, receiver-driven approaches are uniquely suited to our need because only receivers have the potential to access information about the busyness of all destination applications for all flows traversing a given congested downlink. This protocol design makes it possible to decide how to prioritize different flows using this information.

3.2 What is the Best Metric for Application Busyness?

Ideally, how should endhosts measure application busyness? Equivalently, what metric for application busyness best enables the example benefits described in Section §2?

For both examples, what we really want to know is, if each application could be given more work to do, which would make the most progress in the short run? We call this an application's *potential to make progress*. In the first example, knowing that application *B* could make little progress in the near future with an additional request would allow congestion control to prioritize flows for application *A* first. For the second example, knowing at what times the compute-bound application had enough work to keep its cores occupied and at what times it didn't would allow congestion control to allocate just enough bandwidth to the compute-bound application and grant the rest to the network-intensive application.

3.3 How to Expose Application Busyness?

Assuming that *potential to make progress* is the best metrics of application busyness, how can a receiving endhost expose this metric to congestion control algorithms?

In virtualized or containerized environments in which applications are granted a dedicated set of cores to run on, estimating CPU utilization over those cores may provide a good proxy for potential to make progress; applications with many idle cycles can probably process incoming requests in a more timely manner.

Other systems such as IX [33], Arachne [34], and Shenango [32] dynamically reallocate cores across applications. These systems themselves have visibility into the busyness of different applications; the difference between the number of cores allocated to an application and the maximum number of cores it is allowed to have allocated indicates its potential to make use of additional cores by processing incoming requests.

However, neither of these approaches will work for applications that are bottlenecked on resources other than CPU. For example, if an application is bottlenecked on disk accesses, additional requests will not allow it to make more progress. Similarly, real applications such as Memcached often suffer from lock contention at high enough request rates; allowing more traffic to reach such an application similarly does not enable it to complete more work.

Thus, perhaps a better signal of potential to make progress is to observe the rate at which an application is able to process incoming requests. Are incoming packets languishing in socket buffers or in userspace packet queues? How does the rate of outgoing responses change in response to higher rates of incoming requests? The former signal may be hard to measure if packets for a given application are distributed across many different sockets or if applications dequeue packets from socket buffers only to let them wait in application-level queues. The latter signal suffers from time lag, in that a change in the rate of incoming requests will only yield a change in the rate of outgoing responses after the request service time has elapsed; this may or may not be useful, depending on whether service times are on the order of microseconds or milliseconds.

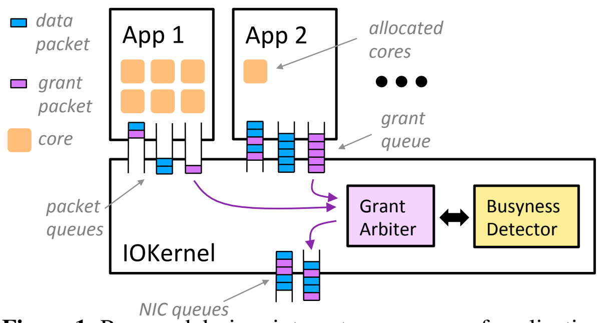

Figure 1: Proposed design: integrate awareness of application busyness, as in Shenango, with the GRANT feature of receiver-driven congestion control approaches. This enables congestion control to prioritize packets based on *application*-level objectives rather than local *network*-level objectives.

Understanding whether an application could make more progress if given more work to do requires visibility into all sources of application-level work, whether they are queued packets, queued requests, or queued threads. Unlike commodity operating systems, Shenango [32] exposes this information. In Shenango, a centralized component called the IOKernel has visibility into thread and packet queues for all applications, and can observe whether these queues are making progress. If a request is stuck in a queue over a certain time interval, the IOKernel estimates the application is backlogged. By contrast, if all outstanding requests have drained from their queues, the IOKernel detects that the application is underloaded. Shenango uses this information to decide how to allocate cores across different applications, but these signals could also be leveraged to decide if an application could make more progress if given more requests.

4 Co-designing Congestion Control and CPU Scheduling in a Datacenter OS

We propose a new datacenter OS, Chimera, that includes a congestion control protocol informed by the CPU scheduler. Chimera is based on Shenango [32], a recent research operating system with insight into application-level queuing.

4.1 OS Design

While Shenango exposes information about application busyness, it does not provide support for receiver-driven congestion control. At the same time, existing receiver-driven congestion control algorithms have no visibility into application busyness. Chimera integrates the key features of these two sets of existing work, thereby enabling the co-design of congestion control and CPU scheduling.

Existing receiver-driven congestion control algorithms such as pHost [18], NDP [20], and Homa [29] vary in the mechanisms they employ. However, at their core, all of these protocols share a key feature, which is that receiving endhosts send GRANT packets (called PULL packet in NDP or tokens in pHost) in order to control the flow of incoming packets.

Shenango's IOKernel monitors application progress, but does not provide any mechanism for controlling the relative priority of traffic from different applications. Instead, it uses the IOKernel to dequeue bursts of packets from application packet queues in a round-robin manner without enforcing an explicit policy across applications.

Figure 1 shows how Chimera integrates the GRANT mechanism described above into Shenango's IOKernel. Chimera adds an additional queue for each application called the *grant queue*. The grant queue provides the IOKernel with information about the amount of ingress data each application would like to receive. This design allows each application to offer grants to the IOKernel for each pending incoming flow, without any rate-limiting.[1] The IOKernel takes these GRANT packets and pass them to the *grant arbiter*, which would then prioritize the sending of GRANT packets based on the policies described below (§4.2). It relies directly on information about application busyness provided by the existing IOKernel busyness detector. These GRANT packets clock the flow of incoming traffic, as in existing receiver-driven approaches to congestion control. In this way, Chimera is able to schedule network traffic according to end-to-end application metrics, rather than based on network-level objectives.

4.2 Policy

Once empowered with the knowledge of each applications' *potential to make progress*, as well as a list of pending flows, how should Chimera's grant arbiter decide which flows to prioritize?

When faced with incoming traffic for multiple applications, the grant arbiter determines the potential to make progress of each application. It then sorts GRANTs into priority order, based on the potential to make progress of the corresponding application, with those with the greatest potential to make progress assigned the highest priority. At the same time, there may be multiple incoming flows for each application. To determine the relative priority of these flows, the grant arbiter falls back on a simple policy such as FIFO. An alternative is to apply a network-centric heuristic such as SRPT, as used by prior work.

Note that sending endhosts may receive GRANT packets from multiple receivers simultaneously, and must also decide how to prioritize flows relative to each other. In existing receiver-driven congestion control algorithms,

[1]Note that in Shenango, transport-layer functionality is implemented by a runtime library that runs in each application's address space.

senders typically address this by implementing the same policy as receivers (e.g., SRPT). In our approach, GRANT messages could be augmented to also include the flow's potential to make progress, so that at sending endhosts, the IOKernel could also prioritize outgoing flows based on potential to make progress.

5 Open Problems

Chimera raises several questions.

Can we prioritize for requests in the critical path? A high-level user request within a datacenter typically results in many RPCs to different services such as caching tiers and databases [11]. Some of these RPCs will be on the *critical path* for that user request, meaning that increasing the latency of that RPC by X would increase the response time for the user request by X as well. However, many of these requests will not lie on the critical path. These requests have some associated slack, $S > 0$, meaning that they could take up to S microseconds longer to complete without impacting the response time for the high-level request at all.

Our discussion of Chimera so far has focused on an individual RPC and its destination application. However, a network that was able to prioritize RPCs that were on the critical path of a high-level request ($S = 0$) over those that were not could improve end-to-end response latencies. While some prior work has used slack in network scheduling, it computes slack time for each individual network flow in order to mimic a given scheduling algorithm's behavior and does not consider how each flow contributes to constructing a high-level user response [28]. Other work has proposed using slack to determine the relative priorities of different requests [11], but estimates slack *a priori* and does not consider how a request's slack may change based on the congestion it encounters during its handling.

How should we modify the CPU scheduling policy? So far, our focus has been on controlling and prioritizing traffic flows in the datacenter network to meet the demands of the application running on a set of CPU cores. However, we could also adjust the number of cores dedicated to each application based on information from the congestion control algorithm about the amount of traffic in each flow. For example, if the congestion control protocol decides to give more priority to flow A because flow B is heading to an application that already has many queued requests, then the OS could preemptively allocate more cores to flow A in response.

Likewise, if the network is forced to drop a packet due to congestion, it could reserve that request's place in application queues by sending a marker to the application. For example, this could be accomplished using packet trimming, which drops only the packet payload but forwards the header on, as in CP [10] and NDP [20]. This would ensure that when the network had enough bandwidth to re-send the packet, the application would have an available CPU to be able to promptly process the request.

How should we handle congestion in the core of the network? Because recent work has shown that most congestion happens at the end links, we focus on balancing contending traffic to and from end hosts. However, networks that are less overprovisioned may also experience congestion at core switches; these networks may benefit from in-network priorization of traffic across applications based on application busyness. For example, if some of the traffic through a core router was for a high-priority application but arriving at a server that was not able to keep up with its incoming requests, we could prioritize a lower-priority application running on a different server that was able to make more progress. How can we communicate application busyness to switches, and how should they prioritize traffic based on this information?

6 Related Work

Many existing congestion control schemes carefully prioritize some flows over others [4, 5, 17–21, 29], but the objectives that they optimize for, such as shortest flow first, flow deadlines, and fairness across flows, only consider the *network*. Unlike these approaches, we consider the behavior of the network and the endhosts when deciding how to schedule network traffic, in order to optimize the end-to-end *application*. Existing approaches to coflow scheduling (e.g., [1, 12–14]) or to scheduling all of the RPCs associated with a high-level user request together (e.g., Baraat [16]), consider groups of related flows, but similarly only focus on the network dynamics and do not consider how they may be impacted by queueing that occurs at endhosts.

7 Conclusion

Significant existing work has explored how to balance demands on limited CPU and network resources. However, none of this work has considered the two resources together. In this paper, we describe the benefits of a more holistic approach to reducing end-to-end application latency and increasing resource utilization. We explore the design for such a system and propose a solution, called Chimera, that uses receiver-driven congestion control and the Shenango OS scheduler. While there are still open problems to be solved, we believe the co-design of network congestion control and OS CPU scheduling has significant advantages over solutions that only consider these layers in isolation.

References

[1] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat. Sincronia: near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 16–29. ACM, 2018.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[5] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 455–468, 2015.

[6] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[7] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[8] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, Dec. 1998.

[10] P. Cheng, F. Ren, R. Shu, and C. Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data center. In *NSDI*, pages 17–28, 2014.

[11] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *OSDI*, pages 217–231, 2014.

[12] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[13] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 393–406. ACM, 2015.

[14] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 443–454. ACM, 2014.

[15] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[16] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 431–442. ACM, 2014.

[17] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM CCR*, Jan. 2006.

[18] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 1. ACM, 2015.

[19] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *NSDI*, pages 1–14, 2015.

[20] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42. ACM, 2017.

[21] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 127–138. ACM, 2012.

[22] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. R. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 519–532, 2018.

[23] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. High Speed Networks Need Proactive Congestion Control. In *HotNets*, 2015.

[24] J. M. Kaplan, W. Forrest, and N. Kindler. Revolutionizing data center efficiency. In *Uptime Institute Symposium*, 2008.

[25] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 9. ACM, 2012.

[26] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, page 4. ACM, 2014.

[27] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[28] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 501–521, 2016.

[29] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[30] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 188–201. ACM, 2016.

[31] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[32] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: achieving high cpu efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[33] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 342–355. ACM, 2015.

[34] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[35] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[36] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

[37] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[38] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[39] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi 2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.