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, this work typically 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, this provides no benefit if the traffic arrives at an overloaded server that will only queue the request.

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, completing a user request involves traversing many distributed nodes and network links. 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 could have been improved if the network had been aware of the server's busyness.

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 visibility into application-level queues, and integrates it with a receiver-driven congestion control protocol to adjust network scheduling in response to endhost queueing. We conclude with a discussion of the open problems in co-designing congestion control with CPU scheduling, including how CPU scheduling could be adjusted in response to network congestion (§5).

2 Benefits of Codesign

Here we identify two potential benefits of co-designing congestion control with CPU scheduling.

2.1 Reducing End-to-End Response Latencies

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

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 application.

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 [14]. 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 application busyness be measured? (§3.3)
- What should congestion control schemes do with this information? (§4.2)

3.1 Receiver-Driven 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 Potential to Make Progress as a Busyness Metric

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.

What is a quantitative metric for potential to make progress? 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.

Alternatively, we could observe how long an application queues requests before it handles them. For example, are incoming packets languishing in socket buffers or in userspace packet queues? However, queueing delay alone cannot distinguish applications based on how much progress a single request might enable; given two applications with equal queueing delay, the application with longer service times will make better use of the CPUs.

Therefore, a quantitative metric for potential to make progress should consider two factors: (1) how soon additional work will be handled and (2) how much that work will utilize the CPU for application-level work. One option that incorporates both of these is:

$$\text{potential to make progress} = \frac{cpu\_usage\_per\_request}{queueing\_delay}$$

where *cpu_usage_per_request* represents the average number of CPU cycles used by an application for each request, and *queueing_delay* represents how long requests queue before an application begins handling them.

3.3 Measuring Potential to Make Progress

Assuming that potential to make progress, as defined above, is the best metric for exposing application busyness, how can a receiving endhost expose this to congestion control algorithms? Queueing delay 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. Estimating queueing delay 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 how long items queue for (Shenango uses this information to decide how many cores to allocate to applications). Specifically, the IOKernel's congestion detector peeks into these queues every 5μs, enabling it to estimate queueing delays for threads and packets to a precision of 5μs; the largest delay across thread and packet queues for an application could be used as the queueing delay estimate. In Shenango, the IOKernel also tracks how many cores each application uses at any given time and runtimes could expose the number of processed requests to the IOKernel, enabling it to estimate *cpu_usage_per_request*.

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 packets 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 for different applications. Instead, the IOKernel processes bursts of packets from the NIC in FIFO order and application runtimes run their own network stacks which independently react to network congestion.

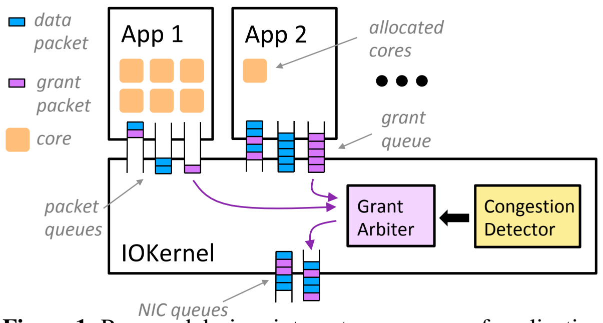

Figure 1: Proposed design: integrate awareness of application busyness, as in Shenango, with the GRANT feature of receiver-driven congestion control approaches. This enables congestion control to prioritize packets based on *application*-level objectives rather than local *network*-level objectives.

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's network stack to offer grants to the IOKernel for each pending incoming flow, without any rate-limiting. The IOKernel takes these GRANT packets and passes them to the *grant arbiter*, which then prioritizes the sending of GRANT packets based on the policies described below (§4.2). It relies directly on information about application busyness provided by the IOKernel's modified congestion detector (§3.3). 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 in addition to 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?

The grant arbiter 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. There may also 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.

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

5 Open Problems

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 about network congestion and demands. For example, if the network path for the outgoing traffic generated by an application is congested, CPU resources could be reallocated to another application running on the same server until the backlog drains. As another 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. These could both potentially be achieved in Chimera by providing feedback from the grant arbiter back to Shenango's core arbiter.

Can we prioritize requests in the application's critical path? A high-level user request within a datacenter typically results in many RPCs to different services such as caching tiers and databases [10]. 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 network flow and its destination application. However, a network that was able to prioritize requests 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 [28], rather than considering how each flow contributes to constructing a high-level user response. Other work has proposed using slack to prioritize server processing across different requests [10]. Chimera may provide a framework for using slack to determine request priority both in the network and at the endhosts. Furthermore, in estimating request slack, Chimera could incorporate information about the application-level queueing that requests will encounter once they arrive at their destination.

How should we handle congestion in the core of the network? Because recent work has argued that most congestion happens at the end links [15, 29], 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 prioritization 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, 11–13]) 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.

8 Discussion

We are interested to hear feedback regarding our proposed design. For example, are there simpler solutions that achieve the same goal? We anticipate that this paper will spark discussions about the potential benefits and drawbacks of co-design, what types of workloads may or may not benefit from co-design, and whether there are other ways to improve end-to-end application performance when considering the entire system holistically.

The most controversial part of the paper is the idea that better prioritization of network traffic could allow network links to be operated at higher utilization with similar or improved performance. The significant overprovisioning of network links in datacenters today suggests that this may be the case, but a quantitative study of real datacenter traffic would enable estimates of exactly how much benefit the co-design we propose could provide. This paper does not address how to optimize for performance of high-level user requests instead of individual RPCs, how to adapt OS scheduling to consider network congestion, or how to handle congestion in the core of the network (§5). Finally, Chimera focuses on settings in which both CPU and network are contended resources; in clusters in which one of these two resources is in ample supply, it will not be beneficial.

References

[1] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat. Sincronia: Near-Optimal Network Design for Coflows. In *SIGCOMM*, 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 *SIGCOMM*, 2013.

[5] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*, 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*, 2013.

[7] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 2007.

[8] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[10] 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*, 2014.

[11] M. Chowdhury and I. Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *HotNets*, 2012.

[12] M. Chowdhury and I. Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*, 2015.

[13] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014.

[14] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[15] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.

[16] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-Aware Scheduling for Data Center Networks. In *SIGCOMM*, 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 *CoNEXT*, 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*, 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 *SIGCOMM*, 2017.

[21] C. Y. Hong, M. Caesar, and P. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 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 *USENIX ATC*, 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 Energy Efficiency. Technical report, McKinsey & Company, 2008.

[25] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *SOCC*, 2012.

[26] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *EuroSys*, 2014.

[27] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[28] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. In *NSDI*, 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 *SIGCOMM*, 2016.

[31] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[32] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*, 2019.

[33] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *SoCC*, 2015.

[34] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-Aware Thread Management. In *OSDI*, 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 *SoCC*, 2012.

[36] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*, 2015.

[37] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). *SIGCOMM*, 2012.

[38] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 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 *EuroSys*, 2013.