Workload Characterization of Interactive Cloud Services on Big and Small Server Platforms

Shuang Chen*, Shay GalOn†, Christina Delimitrou*, Srilatha Manne†, José F. Martínez*
*Computer Systems Laboratory, Cornell University, Ithaca, NY 14853, USA
{sc2682, delimitrou, martinez}@cornell.edu
†Cavium Inc, San José, CA 95131, USA
{shay.galon, bobbie.manne}@cavium.com

***Abstract*—Key-value stores (e.g., Memcached) and web servers (e.g., NGINX) are widely used by cloud providers. As interactive services, they have strict service-level objectives, with typical 99^{th}-percentile tail latencies on the order of a few milliseconds. Unlike average latency, tail latency is more sensitive to changes in usage load and traffic patterns, system configurations, and resource availability. Understanding the sensitivity of tail latency to application and system factors is critical to efficiently design and manage systems for these latency-critical services.**

We present a comprehensive study of the impact a diverse set of application, hardware, and isolation configurations have on tail latency for two representative interactive services, Memcached and NGINX. Examined factors include input load, thread-level parallelism, request size, virtualization, and resource partitioning. We conduct this study on two server platforms with significant differences in terms of architecture and price points: an Intel Xeon and an ARM-based Cavium ThunderX server. Experimental results show that latency on both platforms is subject to changes of several orders of magnitude depending on application and system settings, with Cavium ThunderX being more sensitive to configuration parameters.

I. Introduction

Warehouse-scale systems host many interactive online services, including search, social networking, and online navigation. These are hosted either as monolithic, single-tier applications, or as part of multi-tier configurations, for example consisting of a webserver front-end, a memory caching middle-tier, and a database backend.

These services are typically compute-intensive, and operate under strict *service-level objectives* (SLO). Rather than optimizing for low average latency, SLOs are defined with respect to tail latency, such as 95^{th}, 99^{th}, or 99.9^{th} latency percentiles. This makes interactive services much more sensitive both to application parameters like the intensity of input load, or the size of incoming requests, and to system parameters, such as the underlying architecture, the availability of memory and network resources, and the existence of effective isolation mechanisms, in the presence of multi-tenancy. As resource isolation techniques, such as cache and network bandwidth partitioning, become integrated to more production datacenter servers [1]–[3], and an increasing amount of research goes towards low-power hardware for cloud services [4]–[6], it is critical to quantify the impact these choices have on application latencies.

In this paper, we present a detailed study of two representative interactive cloud services, focusing on the latency impact of several application, hardware, and resource sharing configuration parameters. With respect to application factors, we study the sensitivity to input load, request and dataset size, and thread-level parallelism. With respect to system parameters, we study the sensitivity to hardware ISA, comparing an Intel Xeon and a Cavium ThunderX server platform, as well as the sensitivity to virtualization technologies, such as containers. Finally, with respect to resource sharing, we study the latency impact of a set of OS and hardware partitioning techniques in the presence of multi-tenancy.

In terms of applications, we focus on NGINX and Memcached. **NGINX** [7], a high-performance HTTP server, is one of the most popular open-source web servers globally. It is responsible for serving over 33% of online web requests [8] as of August 2017, making it the second most popular web server platform in production. It owes its popularity to its simplicity, generality, high performance, and scalability. **Memcached** [9] is a high-performance object caching system and it is used for speeding up web requests by caching data and objects in memory. Such distributed, in-memory key-value stores have become a critical tier in modern cloud services, and directly impact their throughput, latency, and efficiency [10]–[12]. Memcached is used extensively by several large companies, such as Facebook, Twitter, and YouTube [13], [34].

These two applications are quite different: NGINX is designed as a stateless front-end service, while Memcached is a stateful middle-tier cache. NGINX requests involve more user-space processing, while Memcached requests are much simpler, and mostly processed in kernel-space. Their target QoS is also different; NGINX typically targets a 99^{th} latency percentile of a few tens of milliseconds [5], while Memcached has slightly more stringent requirements, in hundreds of microseconds up to a few milliseconds [11], [34].

Processors of specifications similar to those of Intel Xeon have been traditionally used by cloud providers. More recently, however, there has been renewed interest for chips with many, relatively small cores, which target highly parallel workloads, and can offer power, area, and price advantages, under the right application and system conditions [14], [15].

In our study, we use a 22-core Intel Xeon server, and a 48-core Cavium ThunderX server, with simpler ARM cores.

We show that, while latency on both platforms is influenced by the examined application and system factors, the Cavium ThunderX shows more sensitivity to application and resource changes. For instance, the overhead of virtualization on ThunderX is *1.6-1.9x* that of the Xeon server. Similarly, software isolation mechanisms require 45% to 80% more time when used on ThunderX.

II. Related Work

Key-value stores have been studied in depth in recent years. Atikoglu et al. [34] use traces from Facebook's Memcached deployment to analyze request composition in production systems. Leverich et al. [11] analyze the system challenges towards maintaining high throughput and low latency with Memcached. They show that queueing delay, scheduling delay, and load imbalance are three dominant factors for Memcached's latency. Li et al. [10] propose a hardware system design for achieving a billion Memcached requests per second. This prior work focuses on profiling Memcached on high-end servers traditionally used by cloud providers.

Over the past decade there has been a renewed interest in hosting cloud workloads on servers comprised of small, low-power cores. Davis et al. [16] explore the performance of multithreaded single- and super-scalar CMTs for commercial workloads running in large-scale systems. They find that single-scalar CMTs significantly outperform their superscalar counterparts given the same area budget. The Niagara microprocessor chip [17] makes the case for small cores to improve the design efficiency and throughput of memory and I/O-bounded workloads. Loghin et al. [18] study the performance of big data applications on mobile ARM nodes. Reddi et al. [5] present an in-depth evaluation of the impact of small cores on a production web search service that uses a compute-intensive machine learning engine, while Hölzle [19] states that brawny cores still outperform wimpy cores when tail latency is the metric of interest. Recent work has also explored special-purpose acceleration units for datacenter workloads, such as websearch indexing [20] and neural networks [21], [22]; although these designs improve both performance and power, they require significant design effort, and are only applicable for specific cloud services. In general, while these studies offer useful insights on the impact of small cores on cloud services, they are not directly applicable to workloads with microsecond level QoS targets, like Memcached. In addition, these studies are limited to dedicated resource instances, where a single application has exclusive access to the underlying platform. In Section V, we also study how resource partitioning and isolation impact tail latency on each of the examined platforms when two interactive services are co-scheduled.

III. Experimental Setup and Methodology

A. Experimental Platforms

Table I summarizes the two examined platforms.

Cavium's ThunderX differs from Intel Xeon in several ways [24]. The most fundamental difference is its ISA. ThunderX implements the ARMv8 64-bit ISA, while Xeon implements X86-64. ThunderX has 48 single-threaded ARM-based cores, which support 48 hardware threads, while Xeon E2699-v4 uses 22 dual-threaded cores to support 44 hardware threads [25]. The ThunderX cores are mostly in-order with some out-of-order execution of memory dependent operations, as opposed to the fully out-of-order Xeon cores. The ThunderX also operates at a lower frequency, and has a smaller last-level cache (LLC). However, its L1 instruction cache is more than double that of the Xeon system. In addition, it has a two-level cache hierarchy as opposed to a three-level for Xeon. Both have 2 levels of TLBs, although the TLB sizes are smaller for ThunderX. Smaller TLBs combined with a smaller LLC implies potentially more page walks and higher page walk latencies.

TABLE I
Processor Specification

	Intel Xeon	Cavium ThunderX
Model	E2699-v4	CN88XX_NT
Sockets	2	2
Cores/socket	22	48
Threads/core	2	1
Frequency	2.2GHz	1.8 GHz
Process	14nm	28nm
L1 Inst/Data Cache	32/32KB	78/32KB
L2	256K	None
Last-Level Cache	55M, 20 ways	16M, 16 ways
Cache line	64B	128B
uTLB	64 entries	32 entries
MTLB	1,536 entries	256 entries
Low Volume Pricing (Oct, 2016)	$4,115 [23]	$785 [24]

B. Application Deployment

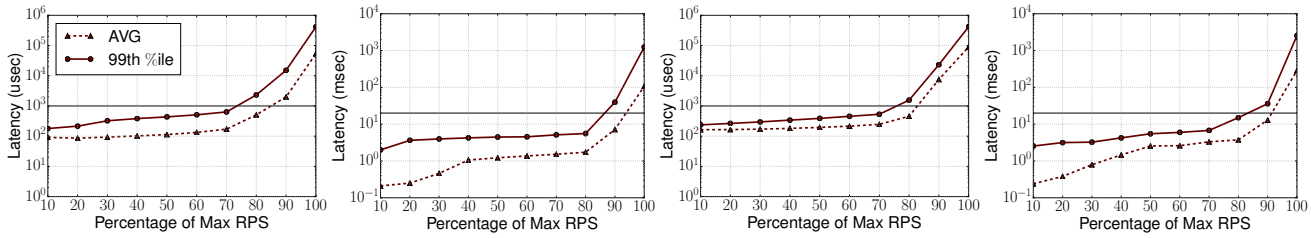
We use Memcached 1.4.36 and NGINX 1.12.0, compiled from their official sources on both platforms [7], [26]. We install Memcached and NGINX on both bare metal and containers. We use LXC (Linux containers) 2.0.7, and Ubuntu 16.04 with kernel 4.8.0 on both platforms.

ThunderX is a dual socket system, however, our test machine only has one socket. For fairness, we also use one socket on the Xeon server. Eight cores are exclusively allocated to network interrupts on both servers, one for each of their 8 network completion queues. Therefore, 14 and 40 cores are available for the cloud services on Xeon and ThunderX respectively.

Unless otherwise stated, the default service deployment is:

- **Memcached**: one 7-thread instance on LXC pinned to 7 different physical cores (hyper-threads on Xeon are not used); 6.4 million items, each with a 30B key and a 200B value; QoS target is set to 1ms for 99^{th} percentile latency.
- **NGINX**: one instance with 7 worker processes on LXC pinned to 7 physical cores; 100K static files, each file is 4KB; Open file cache [27] is enabled for faster file lookup. QoS is set to 20ms for 99^{th} percentile latency.

We disable hyper-threads (HT) on Xeon to allow for a more fair comparison as the ThunderX does not support HT; we study hyper-threading in Section V-B2. We statically configure



(a) Memcached on Xeon (b) NGINX on Xeon (c) Memcached on ThunderX (d) NGINX on ThunderX

Fig. 1. Latency with input load. *max RPS* is the maximum request injection rate as defined in Section III-B. At 7 physical cores, max RPS is 672K and 420K for Memcached and NGINX respectively on the Xeon, and 128K and 84K for Memcached and NGINX respectively on the ThunderX server. Horizontal lines show the target QoS for each application. The y-axis is logarithmic.

the number of threads at instantiation time, since NGINX and memcached do not support dynamic thread spawning. This also avoids overheads for synchronization, lock contention, and load balancing. We instantiate 7 threads per server, as some of the later experiments require running the two services on different physical cores, and only 14 cores are available in total on the Xeon platform.

The maximum achievable throughput (requests per second, RPS) differs between the two platforms, when using the same number of physical cores. We measure the maximum RPS that seven physical cores can sustain in each architecture, such that injecting higher rates will result in dropped requests. Xeon can sustain 672K and 420K RPS with seven cores for Memcached and NGINX, respectively; ThunderX can sustain 128K and 84K, respectively. The throughput per core on the ThunderX system is lower due to the lower frequency, smaller caches and TLBs (Section III-A), and because the system relies on parallelism, as opposed to high single-thread performance to provide good QoS at the socket level. (A back-of-envelope whole-socket extrapolation would put ThunderX at $128K \times 40/7 = 731K$ RPS for Memcached, and $84K \times 40/7 = 480K$ RPS for NGINX.) Socket-level comparisons for throughput can be found in [24], [25].

C. Testing Strategy

We use open-loop workload generators as clients for both Memcached and NGINX to ensure that latency measurements at high load are accurate [28], [29]. For Memcached, we use an in-house load generator, similar to mutilate [30]. For NGINX, we modified a popular open-source generator, wrk2 [31], from close- to open-loop. Clients run on one or several other Intel Xeon processors, with 10Gbps network links to the Xeon and ThunderX servers. We instantiate enough clients to avoid client-side delay from saturation. Therefore, latencies reported by clients are mostly due to server-side delays.

We use exponential distribution as requests' inter-arrival time distribution [11], to simulate a Poisson process, where requests are sent continuously and independently at a constant average rate. We use a Zipfian distribution for the request popularity in NGINX [32], [33]. We only generate GET requests as they statistically account for more than 95% of all requests in production systems [34]. For each experiment, we run the clients for 2 minutes (excluding time for warm-up and cool-down), and record achieved throughput, and average and 99^{th} percentile latency.

IV. Workload Characterization in Isolation

We explore how the average and tail (99^{th} percentile) latencies of Memcached and NGINX change with the following factors. In terms of application parameters, we study *input load*, *thread parallelism*, and *request size*. With respect to system parameters, we evaluate the overhead of *virtualization*. All studies are done on the two platforms described in Section III-A. Hereafter, we use tail and 99^{th} percentile latency interchangeably, unless otherwise noted.

A. Input Load

Cloud servers are usually overprovisioned and lightly utilized [1], [35]–[38]. Typical CPU utilization rarely exceeds 30%, with public clouds being even more underutilized than private systems [39]. There are several reasons for this underutilization, including the current reservation-based cluster management interfaces, provisioning for diurnal patterns and unpredictable load spikes, hardware heterogeneity, resource interference, and planning for future growth [36], [40], [41].

We first study how latency is impacted by increasing load, to determine whether overprovisioning is warranted in interactive services. We progressively increase request injection from 10% to 100% of the max RPS as determined in Section III-B, and plot the average and tail at each input load level. Injection rates per load level are kept stable, as discussed in Section III-C. To understand the source of increased latencies, we further use Systemtap [42], a system profiling tool that enables inserting probe points in the Linux kernel.

Figure 1 plots the relationship between latency and input load. We observe similar trends and thresholds on Xeon and ThunderX. The maximum input load for which the server still meets the target QoS is approximately 70% of the max RPS for Memcached and 80% for NGINX. These load points are significantly higher than current datacenter utilizations, signaling that, excluding unexpected load spikes, resource overprovisioning is not necessary to preserve tail latency QoS.

NGINX has a higher saturation point than Memcached, mostly because of its more relaxed SLO, 20ms in our case, which makes it possible to tolerate higher delays. Average

TABLE II
LATENCY BREAKDOWN OF MEMCACHED IN MICROSECONDS AT DIFFERENT PERCENTAGE OF MAX RPS.

	Xeon		ThunderX	
	10%	90%	10%	90%
Network	6	6	14	14
Epoll	3	782	4	1,290
Libevent	1	1,009	5	1,650
Read	1	3	9	20
Memcached	1	1	7	7
Send	5	5	24	24
Total	20	1,806	67	3,005

latencies are smaller on ThunderX because requests take longer to process on its smaller cores.

To understand the source of increased latencies, we further use Systemtap [42], a system profiling tool that inserts probe points at kernel level along the application's control flow. A Memcached request goes through the following stages on the server side [11]:

1) *Network:* A request first arrives at the server's NIC, raising a hardware interrupt. Linux acknowledges the interrupt, and further processes the packet in the *softIRQ* context. The request then goes through the network stack, including TCP/IP processing. The Memcached process is then invoked to further handle this packet.
2) *Epoll:* Memcached uses the *epoll_wait* syscall to queue and receive new requests. *epoll_wait* is called periodically at an interval defined by *timeout*. Requests coming in the same *timeout* interval are received at the same time, reducing the number of syscalls in the system. However, if previous requests have not been propagated to the later processing steps, new requests cannot be received, increasing the queueing time at *epoll_wait*.
3) *Libevent:* Received requests by *epoll_wait* are then forwarded to be parsed and processed. If requests are received at high rates, they will again be subject to long queueing delays before processing can commence.
4) *Read:* Memcached calls the *read* syscall to read the socket with the new incoming request, which involves unwrapping the network packet header and payload, applying any priorities denoted in the header, and forwarding the payload to the next stage for processing.
5) *Memcached:* Memcached requests are short and simple. The server process first parses the request, then looks up the key in its hash table, and retrieves a pointer to the requested value.
6) *Send:* Finally, the obtained key-value pair is processed by TCP/IP and sent to the NIC's TX queue.

Note that request processing in the *Memcached* stage is serialized. When request injection rates are high, this stage becomes a bottleneck, increasing the queueing delays in the *Read* phase, and especially in *Libevent* and *Epoll*. Depending on buffer sizes, queueing delays vary across stages. Table II shows the average latency breakdown at 10% and 90% of the max RPS for each platform.

TABLE III
LATENCY BREAKDOWN OF NGINX IN MILLISECONDS AT DIFFERENT PERCENTAGE OF MAX RPS.

	Xeon		ThunderX	
	10%	95%	10%	95%
Network	9	10	15	15
Epoll	13	10,230	38	12,364
Libevent	11	23,186	12	25,821
Read	1	3	8	19
Open	4	4	14	14
NGINX	12	12	46	46
Send	26	26	73	73
Total	76	33,471	206	38,353

When operating at the same percentage of max RPS, ThunderX experience a higher latency than Xeon. At 10% of max RPS, both servers are underutilized, and there is negligible queueing delay. The average latency of the Cavium server is 3.35x of the Intel platform because of the processor configuration parameters discussed in Section III-A. When servers are 90% loaded, batching in *Epoll* and *Libevent* result in significant queueing delays on both platforms, which dominate the total latency.

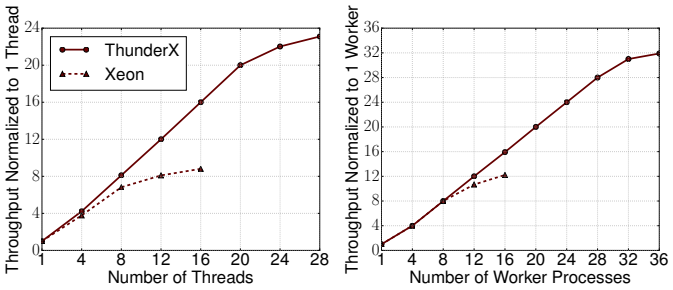
An NGINX request experiences the same stages of *Network*, *Epoll*, *Libevent*, and *Read* as Memcached. After *Read*, its control flow is the following:

1) *Request parsing:* The content of an NGINX request is obtained after a *Read*. Unlike Memcached, NGINX requests are more complex, and require additional time for HTTP header formatting and processing. Therefore, getting to the point of parsing and obtaining the request type and body takes longer than in Memcached.
2) *File lookup:* Once a request is parsed, and the requested file name is obtained, the server accesses the file system (FS). Instead of incurring the long latencies associated with FS accesses on every request, NGINX maintains an open file cache (in *ngx_http_core_module* [27]) that holds previously-opened file metadata in memory to accelerate file lookups. The file cache is maintained as a red black tree indexed by file name. If the name does not exist in the tree, it will be inserted upon first access to speed up later accesses to the same file.
3) *Open:* The *open* syscall is then invoked to open and read the file.
4) *Postprocessing:* Once the file content is obtained, there is a second processing phase. This step is heavily dependent on the specific NGINX configuration. For example, if *gzip_on* is enabled, the file content is compressed in this step. If *access_log* is enabled, NGINX records execution traces in a log file.
5) *Send:* Finally, a response packet is formed, processed by TCP/IP and sent to the NIC's TX queue.

As *Request parsing*, *File lookup* and *Postprocessing* are all user-space processing, and there is no batching in any of these steps, we combine and denote them as *NGINX*. Table III shows the average latency breakdown of NGINX at 10% and 95%

of the max RPS.

Under low load, *Epoll*, *NGINX* and *Send* contribute the most to latency. The long *Epoll* latency comes from the time spent waiting for new requests. NGINX involves more user-level processing than Memcached, leading to the large *NGINX processing* time. *Send* takes long because of the size of the requested file (4KB as discussed in Section III-B). Under high load on the other hand, most of the delay comes from queueing, with the rest of the parts remaining the same.



(a) Memcached (b) NGINX

Fig. 2. Scalability when increasing the number of threads. Each thread is pinned to a different physical core. Throughput is the maximum RPS under QoS (99^{th} percentile in 1ms for Memcached and 20ms for NGINX). Results are normalized to throughput under a single thread on the same machine. For Memcached, single thread throughput is 144K and 80K on Xeon and ThunderX respectively. For NGINX, it is 110K and 50K on Xeon and ThunderX respectively.

B. Scalability (Scale-Up versus Scale-Out)

Memcached and NGINX are both distributed applications whose datasets are sharded across a large number of machines [43]. However, given that modern servers are equipped with tens of cores, and interactive services can benefit from efficient data sharing across threads of a single machine, we want to first evaluate the scalability of the examined services as we increase the number of threads per server (*scale up*). Scaling up simplifies cross-thread data sharing, requires fewer hardware resources, such as memory, and alleviates the issue of sharding keys across instances for key-value stores like Memcached. On the other hand, if an application does not benefit from multithreading, launching multiple application instances with one thread (or a small number of threads) each can reduce lock contention, synchronization, and task stealing among threads. In this case, *scaling out* is preferred over scaling a single instance up.

Figure 2 shows the scalability of request throughput as we increase the number of threads. We pin each thread to a distinct physical core. We run this experiment on baremetal hardware to avoid overheads induced by containers or VMs, which are discussed in Section IV-D. For a given number of threads, we plot the maximum throughput for which the target QoS is met.

The initial observation from Figure 2 is that ThunderX is able to scale up further than Xeon. This is because the Xeon server sustains higher throughput than the ThunderX for the same number of threads. Therefore, Xeon suffers more from the synchronization and contention issues discussed below. The second observation is that both servers cannot fully utilize the entire socket. Xeon and ThunderX cannot scale further than 16 and 36 cores respectively because they are bottlenecked by the IRQ cores. The third observation is that NGINX shows better scalability than Memcached. Memcached is only able to scale linearly up to 20 threads on ThunderX, and up to 4 threads on Xeon, while NGINX stops scaling linearly after 28 workers on ThunderX and after 8 workers on Xeon. This is because NGINX has little shared state across threads, allowing each worker process to work independently.

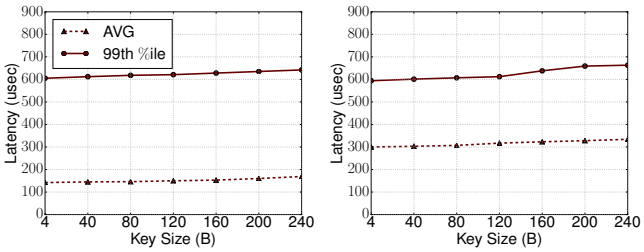
After the thread count exceeds the thresholds above, further scaling up does not produce higher throughput. Below we discuss in more detail the reasons that hinder throughput scalability when increasing the number of worker threads:

- **Interrupt handling:** Both servers support receive side scaling (RSS) [44], which distributes network receive processing across multiple hardware-based receive queues. By configuring the IRQ affinity, traffic for different queues can be processed on different physical cores. For example, the NICs of the two test machines both have 8 queues. We therefore exclusively allocate cores 0-7 for network processing. As load increases, we observe that the IRQ cores become saturated ahead of the cores servicing regular application worker threads. While the remaining unallocated cores can still become saturated by non-network intensive batch jobs, the compute resources needed for network interrupts limit the maximum throughput for the examined interactive services. This effect is more severe in Memcached than NGINX because Memcached requests are simpler, thus its throughput is higher which causes more burden in network processing.
- **Load imbalance:** When instantiating a multi-threaded service, we run the risk of work not being evenly distributed across the multiple threads. For instance, Memcached naively allocates a new client connection to threads in a round-robin way. Requests to different connections are not guaranteed to require equal amounts of processing. However, the behavior of clients is not known while setting up connections. This requires rewiring the way Memcached binds connections to worker threads.
- **Lock contention:** Memcached is a stateful service. All the threads share the entire dataset and hash table. Each item is associated with several counters, such as the number of accesses, the number of hits and misses, etc. Such counter accesses are guarded by locks. In NGINX, locks should be acquired ahead of accessing files. With more threads, there is a higher probability that several threads access the same key or file at the same time, contending for locks.

The issues above show that after the thread count exceeds an application- and system-dependent threshold, it is more beneficial to set up multiple instances with a smaller number of threads each, i.e., scale out.

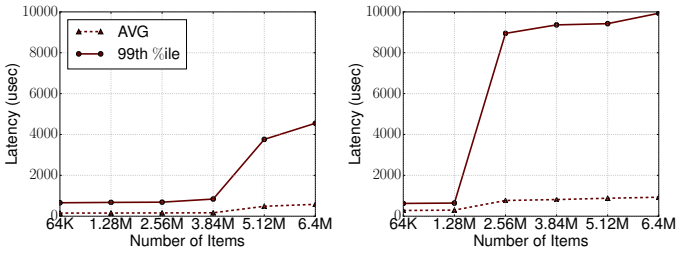
C. Request Size

Memcached and NGINX are widely deployed in both private and public clouds. Request size distributions vary across



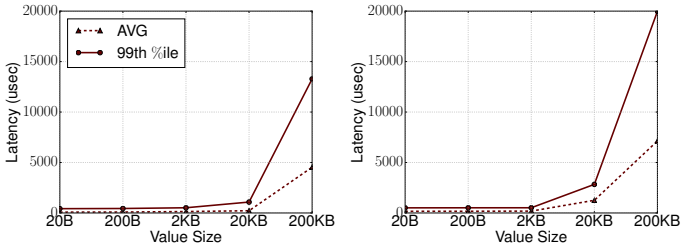
(a) Xeon (b) ThunderX

Fig. 3. Impact of Memcached key size. Both Xeon and ThunderX operate at 70% of their respective max RPS.



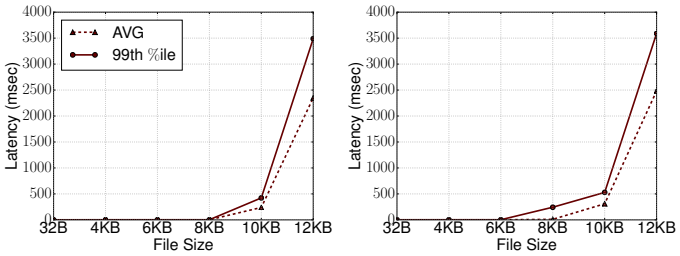
(a) Xeon (b) ThunderX

Fig. 5. Impact of the number of Memcached items. Both Xeon and ThunderX operate at 90% of their respective max RPS.



(a) Xeon (b) ThunderX

Fig. 4. Impact of Memcached value size. Both Xeon and ThunderX operate at 70% of their respective max RPS.



(a) Xeon (b) ThunderX

Fig. 6. Impact of NGINX file size. Both Xeon and ThunderX operate at 80% of their respective max RPS.

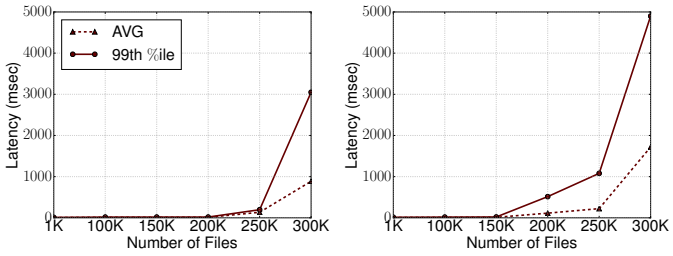
use cases, and can result in substantially different throughput and latency. Here we focus on the impact of the key and value size, and the number of items for Memcached, and on the impact of the file size, and number of files for NGINX.

1) Memcached:

- **Key size:** Figure 3 shows latencies when sweeping keys in $[4B, 240B]$. Latency is barely influenced. Memcached uses the highly optimized $strcmp()$ function to compare two keys. In addition, the maximum allowed key size is restricted to 250B, which is not large enough to cause a substantial difference in either throughput or latency. For key sizes of over 200B there is a small increase in tail latency due to the increased memory traffic.
- **Value size:** There is no limit for value sizes. Therefore, we sweep values from 2B to 200KB. As shown in Figure 4, latencies increase dramatically for larger value sizes. Memory and network play an important role in this case. Larger values result in more time spent in memory copy when forming the response packets. They also lead to larger network packets, which translates to higher per packet latency both from processing and queueing, and lower throughput when the link's bandwidth limit is reached. Using Systemtap, we find an increase in the latency of the *Send* stage. This increase further leads to higher queueing delays in *Epoll* and *Libevent*.
- **Number of items:** Figure 5 shows how latency changes as the number of items increases. When there are only 64K items, with 30B keys and 200B values, the total dataset size is around 15MB which is less than the LLC size of both servers. When there are 6.4M items, the dataset is over 1GB. ThunderX is more sensitive to the number of items; latencies increase with 2.56M items, while latency on the Xeon server remains low. Cache utilization is linked to this behavior. Each item is represented as an item object, whose pointers are frequently used in hash table lookups, LRU maintenance, item statistics update, etc. Suppose there are 2.56 million items; we can roughly estimate that approximately 20MB are needed to store 2.56M 8B pointers (in 64-bit systems). All the pointers fit in the 55MB LLC of Xeon, but do not fit in the LLC of ThunderX (16MB).

2) NGINX:

- **File Size:** Related work has shown that web file size distributions have a long tail [45]; about 50% of web files are less than 1KB, 95% are less than 64KB, and the 99^{th} percentile is 32GB. The distribution is also greatly influenced by the scope of the website. For example, a personal website typically consists of small text files and small image files. However, websites on photography include thousands of multi-MB high-quality pictures. Figure 7 shows the impact of file size on NGINX. Similarly to Memcached, larger files lead to higher memory and network latencies. In addition, large files can easily saturate the server's network bandwidth, lowering throughput. Therefore, for websites serving large static files, high network bandwidth is more important than compute resources.
- **Number of files:** When we only use 1K files, with 4KB each, the total dataset size is approximately 4MB, which is less than the LLC size of both servers. In comparison, when there are 300K items, the dataset is



(a) Xeon (b) ThunderX

Fig. 7. Impact of the number of NGINX files. Both Xeon and ThunderX operate at 90% of their respective max RPS.

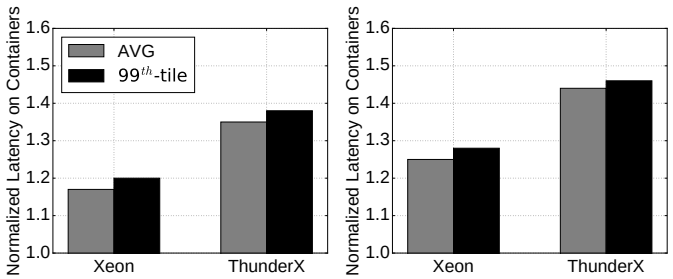
over 1GB. As shown in Figure 7, the number of files also has a great impact on NGINX latency because of increased cache thrashing. The *open_file_cache* in *ngx_http_core_module* [27] can substantially help reduce the end-to-end request latency for NGINX. The file cache maintains a red-black tree that stores all file metadata to speed up file lookups. When the number of files is small, the entire tree with all its metadata fits in the cache. However, if *open_file_cache* is disabled, the overhead of frequent accesses to the file system prevents NGINX from achieving high RPS while meeting QoS.

As with Memcached, latency increases earlier on ThunderX than Xeon because of the former's smaller last-level cache. Each tree node in the file cache contains hundreds of bytes, including flags, integers, and strings for the file path, file size, file descriptor ID, permission access for files, etc. It also contains a set of pointers for maintaining the tree structure. When 200K files are frequently accessed, a tree with 200K nodes requires tens of MB of space, which fits in the LLC of Xeon, but does not fit in the LLC of the ThunderX server.

D. Virtualization

Resources in both private and public clouds are almost-always virtualized [46]. Virtual machines (VM) [47] provide full virtualization of hardware, and improve security in the presence of multi-tenancy. Different VMs can also run different operating systems (OS) in isolation. If multiple OSes are not needed, a more lightweight option is OS-level virtualization, where the kernel is shared. Linux containers (LXC) are one of the most popular OS-level virtualization mechanisms. Containers provide easier software packaging, and support a variety of hardware as well as software isolation mechanisms inherited from the Linux kernel. In this paper, we use LXC for virtualization. To better observe the overhead of virtualization compared to queueing delays, the CPU utilization of each server is kept at 10%.

Figure 8 shows the average and 99^{th} percentile tail latency using containers, normalized to latency on baremetal. We find that containers introduce high overheads: 1) to tail latency more than average latency because virtualization creates more unpredictability in request processing, which affects tail latency more than average; 2) to NGINX more than Memcached because NGINX involves heavier processing than Memcached;



(a) Memcached (b) NGINX

Fig. 8. Latency when running on containers normalized to baremetal. Both Xeon and ThunderX operate at 10% of their respective max RPS.

3) to ThunderX more than Xeon because the additional level of indirection introduced by containers is more cumbersome for the ThunderX's weaker cores than for Xeon, as discussed in Section III-A.

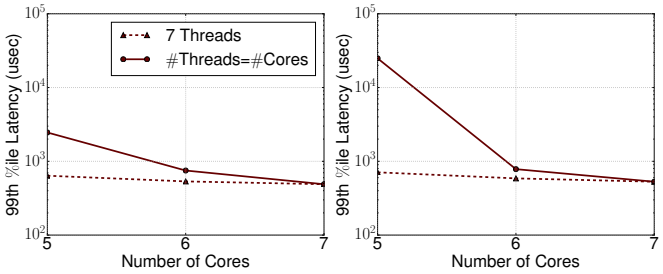
V. Workload Characterization Under Multi-Tenancy

Recent studies have proposed colocating latency-critical (LC) workloads with best-effort tasks in datacenters [1], [2], [48]. Though potentially improving server utilization, colocation can either cause LC workloads to suffer from resource interference, or the performance of best-effort jobs to be frequently sacrificed. This also prevents colocating two or more high priority and/or interactive workloads. To address this issue, it is critical to understand the resource needs of interactive services. In this section, we use several hardware and software isolation mechanisms to limit the amount of allocated resources per-application, and study the impact on latency. The examined isolation mechanisms are supported by both platforms, although some operate differently. Apart from the impact on latency, we also compare the overhead of isolation mechanisms between the two platforms.

A. Isolation Mechanisms

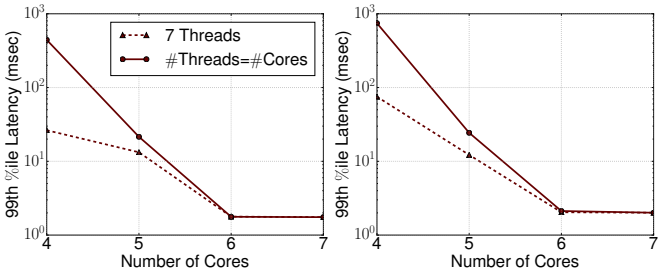
We use three software and one hardware isolation mechanisms [1]:

1) **Software isolation:** We use mechanisms provided by the Linux kernel to partition *cores*, *memory capacity*, and *network bandwidth*. We use **cpuset cgroups** to pin each workload to a set of CPUs. We show that not only the number, but also the location of CPUs matters (see Section V-B). **memory cgroup** is used to restrict memory capacity. **qdisc** [50] with hierarchical token bucket queueing discipline is used to limit the outgoing network bandwidth. These isolation mechanisms are the same on the Xeon and ThunderX servers.
2) **Hardware isolation:** Cache partitioning requires hardware support. Intel Cache Allocation Technology (CAT) [51] supports a number of service classes; 16 in our test machine. With 20 cache ways, each class corresponds to a 20-bit cache way mask. Each CPU is then bundled to one of the 16 classes. Because of hardware limitations, the mask can only have consecutive 1s. e.g.,



(a) Xeon (b) ThunderX

Fig. 9. The relationship between the number of threads and cores for Memcached. Both Xeon and ThunderX operate at 60% of their max RPS.



(a) Xeon (b) ThunderX

Fig. 10. The relationship between the number of threads and cores for NGINX. Both Xeon and ThunderX operate at 60% of their max RPS.

0x55555 is not allowed. CAT impacts both the cache allocation and replacement policy.

Unlike Intel CAT, there is no indirection of service classes on ThunderX. Each core is directly associated with a way mask, so there can be 48 different cache way allocations on a 48-core processor. There is no restriction for consecutive 1s as in Intel CAT. In addition, cache partitioning only influences cache replacement on the Cavium board. This helps reduce the wasted cache space. For example, if the working set of an application fits in the cache, i.e., there are only compulsory misses, it will not experience more cache misses with way partitioning than it would experience without [52].

Way masks on both platforms can be accessed or changed by reading or writing registers. There are several tools for accessing hardware registers on X86 platforms, but not for ARM processors. To address this, we designed a C++ program with inline assembly used to change hardware registers on both platforms, which is additionally 8x faster than existing register sampling tools available for the Xeon server.

We further investigate the overhead of these isolation mechanisms by applying each mechanism 10K times. We also study the overhead of frequent container instantiation and teardown. Table IV shows the average overhead of each isolation mechanism the two platforms.

TABLE IV
OVERHEAD OF ISOLATION MECHANISMS

	CPU	Cache	Memory	Network	Container Restart
Xeon	6.5ms	1.6ms	6.9ms	2.4ms	0.97s
ThunderX	11.7ms	2ms	11.9ms	3.5ms	1.55s

Except for container restart, all isolation mechanisms take a few milliseconds. For Memcached whose SLO is in a few milliseconds, any unnecessary or incorrect isolation decision contributes to higher tail latency. Frequently killing and instantiating containers, as in serverless settings, should also be avoided unless necessary as its overhead is in the order of seconds. In this study, we focus on static isolation before each run to avoid interference from dynamic partitioning.

TABLE V
HYPER-THREADING

Load	Memcached			Nginx		
	4–4	4–7	7–7	4–4	4–7	7–7
25%	904μs	253μs	242μs	3.6ms	3.1ms	3.0ms
50%	-	696μs	524μs	-	9.6ms	7.4ms
75%	-	-	992μs	-	703.9ms	8.9 ms

B. Core Isolation

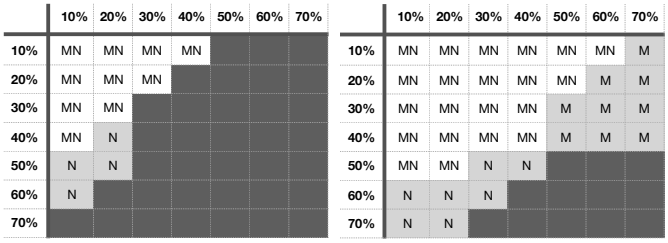
1) Number of threads VS Number of cores: We bring each application to 60% of their max load. With 7 threads, the aggregate CPU utilization is less than 500%. We sweep the number of allocated cores, and examine whether five cores - operating close to saturation - are sufficient.

Figures 9 and 10 show that fewer cores than the number of threads result in SLO violations. When multiple threads are mapped to the same physical core, only one thread can get executed, increasing latencies of other threads, due to context switching. We also plot the case of reducing the number of threads when using fewer cores, which eliminates the overhead of context switching. The comparison shows that context switching is one of the dominant factors for latency, and its overhead is more severe on ThunderX.

2) Benefits of Hyper-threading: The study above does not make use of the Intel Hyper-Threading Technology, which enables two threads to run on the same physical core [53]. Hyper-Threading is often disabled in datacenters to reduce interference between logical cores [54]. We experiment with the effect of hyper-threading by instantiating 7 threads on the 7 logical cores of 4 physical cores, and comparing against using 7 logical cores on 7 physical cores. For this experiment, we focus on Xeon, because hyper-threading is not available of the ThunderX platform.

Table V compares different uses of hyper-threading. $n-m$ represents the configuration with n physical cores and m logical cores. If the target throughput cannot be met, the corresponding entry is –.

The table shows that Hyper-threading works well if load is low. Compared to *4–4*, configuration *4–7* avoids the overhead of frequent context switching when sharing logical cores, and makes better use of compute resources. In fact, Hyper-threading is more useful when colocating different applications



(a) Same Logical Core (b) Different Logical Cores

Fig. 11. QoS when colocating Memcached and NGINX with and without Hyper-threading. In Figure 11a application are colocated on the same logical cores of 7 physical cores. In Figure 11b they are colocated on different logical cores of 7 physical cores. Each row is a load point of Memcached, and each column a load point of NGINX. White cells signify that both applications meet their target QoS. Light grey cells with M or N mean that only Memcached or NGINX meets its QoS, respectively. Dark grey cells represent that neither application can meet its target QoS.

TABLE VI
IMPACT OF INTERFERENCE WITH IRQ

Percentage of Max RPS	Memcached		Nginx	
	Xeon	ThunderX	Xeon	ThunderX
25%	358μs	397μs	3.4ms	3.9ms
50%	6463μs	30136μs	9.11ms	10.2ms
75%	-	-	19.1ms	6640ms

on different logical cores of the same physical cores. Figure 11 shows the benefit of colocating Memcached and NGINX on the same physical cores, but different logical cores. Though still sharing the limited compute resources, the overhead of context switching is eliminated.

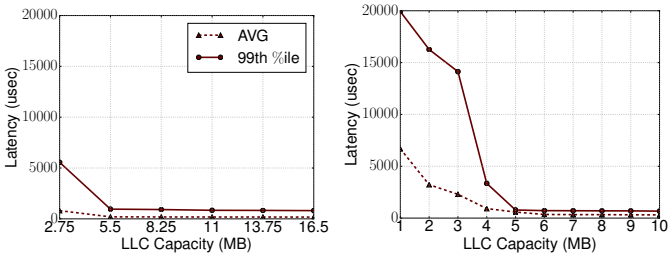
3) Interference with SoftIRQ cores: As discussed in Section IV-B, interrupt handling is an important part of interactive workloads. A set of cores is usually exclusively reserved for network interrupts. Using *cpuset cgroup*, we can further study the impact of interference caused by interrupts when workloads are colocated with them.

Table VI shows the tail latency after colocating the interactive services with interrupt cores. We find that (1) Memcached is more sensitive to interference with IRQ cores. This is because Memcached usually achieves higher throughput than NGINX because of its requests being simpler. IRQ cores then handle more interrupts when running Memcached, creating more interference. (2) ThunderX is more sensitive to IRQ interference because of the higher impact of context switching.

C. Cache Sensitivity

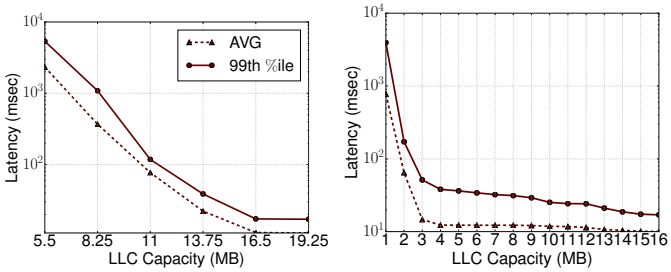
The Xeon has a 55MB LLC with 20 cache ways, so the minimum allocation granularity is 2.75MB. The ThunderX platform has 16MB LLC with 16 cache ways, therefore cache space can be controlled at 1MB increments.

Figure 12 shows how latency drops with larger cache capacity for Memcached. We can infer that about 5MB of data should be kept in the LLC. As discussed in Section IV-C, Memcached benefits from LLC if all the item pointers can be kept in cache. However, with 6.4 million items, all the pointers take hundreds of MB. Therefore, the LLC is only useful to keep data not related to Memcached items, including Memcached statistics, pointers to Memcached LRU lists, etc.



(a) Xeon (b) ThunderX

Fig. 12. Impact of the LLC capacity for Memcached. Both Xeon and ThunderX operate at 70% of their max RPS.



(a) Xeon (b) ThunderX

Fig. 13. Impact of the LLC capacity for NGINX. Both Xeon and ThunderX operate at 80% of their max RPS.

Figure 13 shows the impact of cache capacity for NGINX. Unlike Memcached, the turning points on Xeon and ThunderX are quite different. On Xeon, we can infer that the working set size of NGINX with 100K 4KB files is approximately 16.5MB, which is larger than the LLC of the ThunderX. Given that the working set size does not fit in the LLC on ThunderX to begin with and the access pattern is uniform and random, NGINX is not sensitive to LLC allocation. Latency is higher with 1-2 cache ways because of conflict misses, but beyond 2 cache ways, latency does not benefit further.

D. Memory Capacity and Network Bandwidth Isolation

Both Memcached and Nginx have a fixed requirement of memory capacity if their datasets remain unchanged. The memory needed by Memcached includes the space to store all keys, values, and hash table entries, plus an additional fudge factor caused by Memcached's slab-based memory management. The memory needed by NGINX includes the space to store all the files and the open file cache.

Network bandwidth requirements are also fixed, and slightly higher than $throughput * values$ or $filesize$, for given datasets and load.

Neither workload benefits from memory capacities beyond their dataset sizes, or higher network bandwidth than needed to transfer response packets. Both services suffer significantly if either memory capacity or network bandwidth are insufficient, resulting in 1000x less throughput.

VI. Conclusions

We presented a detailed study of two representative interactive services, Memcached and NGINX, on an Intel Xeon and a Cavium ThunderX server platform. We study the impact of a number of application and system parameters on the average and tail latency of the two applications, and use resource isolation mechanisms to study their sensitivity to different hardware resource availabilities.

Comparisons between the two platforms show that ThunderX achieves lower throughput per core, and shows higher latency and higher overhead to different isolation mechanisms, due to overheads from context switching, virtualization, and resource interference. However, ThunderX is still able to meet the target QoS in most cases, and it shows better thread scalability, in part because of its higher core count.

VII. Acknowledgements

This work was supported in part by a research contract with Cavium, and by equipment donations from Cavium and Intel.

References

[1] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ISCA-42*, 2015.

[2] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," in *ASPLOS*, 2014.

[3] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *MICRO*, 2015.

[4] X. Liang, M. Nguyen, and H. Che, "Wimpy or brawny cores: A throughput perspective," *J. Parallel Distrib. Comput.*, 2013.

[5] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores: Quantifying and mitigating the price of efficiency," in *ISCA-37*, 2010.

[6] K. Vaid, "Datacenter power efficiency: Separating fact from fiction," Workshop on Power Aware Computing and Systems, 2010.

[7] "Nginx official website," http://nginx.org.

[8] "Usage statistics and market share of nginx for websites," https://w3techs.com/technologies/details/ws-nginx/all/all.

[9] B. Fitzpatrick, "Distributed caching with memcached," in *Linux Journal*, 2004.

[10] S. Li, H. Lim, V. W. Lee *et al.*, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ISCA*, 2015.

[11] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *EuroSys*, 2014.

[12] J. Li, N. K. Sharma, and S. D. Ports, Dan RK Gribble, "Tales of the tail: Hardware, OS, and application-level sources of tail latency," in *SoCC*, 2014.

[13] M.-C. Lee, F.-Y. Leu, and Y.-P. Chen, "Cache replacement algorithms for youtube," in *AINA-28*, 2014.

[14] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *ACM SIGARCH Computer Architecture News*, 2008.

[15] M. Coppola, B. Falsafi, J. Goodacre, and G. Kornaros, "From embedded multi-core socs to scale-out processors," in *DATE*, 2013.

[16] J. D. Davis, J. Laudon, and K. Olukotun, "Maximizing cmp throughput with mediocre cores," in *PACT-14*, 2005.

[17] L. Geppert, "Suns big splash: Niagara multiprocessor chip," *IEEE Spectrum*, 2005.

[18] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo, "A performance study of big data on small nodes," *VLDB*, 2015.

[19] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE Micro*, 2010.

[20] A. Putnam, A. M. Caulfield, E. S. Chung *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA-41*, 2014.

[21] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS-19*, 2014.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA-44*, 2017.

[23] "Specifications of Intel Xeon E5-2699 v4," http://ark.intel.com/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz.

[24] Cavium Inc., "High performance memory caching using thunderx," *Tirias Research*, 2016.

[25] ——, "High performance nginx content delievery using thunderx," *Tirias Research*, 2016.

[26] "Memcached official website," http://memcached.org.

[27] "Nginx documentation of http core module," http://nginx.org/en/docs/http/ngx_http_core_module.html.

[28] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale." in *NSDI*, 2006.

[29] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *ISCA-43*, 2016.

[30] "Memcached load generator," https://github.com/leverich/mutilate.

[31] "Wrk2: A constant throughput, correct latency recording variant of wrk," https://github.com/giltene/wrk2.

[32] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet." *Glottometrics*, 2002.

[33] L. Ramaswamy, L. Liu, and A. Iyengar, "Cache clouds: Cooperative caching of dynamic documents in edge networks," in *ICDCS-25*, 2005.

[34] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.

[35] L. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2009.

[36] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *ASPLOS-19*, 2014.

[37] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: precise online qos management for increased utilization in warehouse scale computers," in *ISCA-40*, 2013.

[38] J. Mars and L. Tang, "Whare-map: heterogeneity in "homogeneous" warehouse-scale computers," in *ISCA-40*, 2013.

[39] "Host server cpu utilization in amazon ec2 cloud," http://goo.gl/2LTx4T.

[40] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," ser. EuroSys, 2015.

[41] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys*. 2013.

[42] D. Domingo and W. Cohen, "Systemtap 2.9 systemtap beginners guide," 2013.

[43] M. Ferdman, A. Adileh, O. Kocberber *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS*, 2012.

[44] I. S. Adapters, "Receive side scaling on intel network adapters."

[45] A. S. Tanenbaum, J. N. Herder, and H. Bos, "File size distribution on unix systems: then and now," *SIGOSR*, 2006.

[46] R. W. Schmidt and S. Grarup, "Vapp: A standards-based container for cloud providers," *SIGOSR*, 2010.

[47] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*, 2005.

[48] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *ASPLOS-21*, 2016.

[49] "Cgroups," http://man7.org/linux/man-pages/man7/cgroups.7.html.

[50] M. A. Brown, "Traffic control howto," http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[51] "Intel ®64 and IA-32 Architecture Software Developer's Manual, vol3B: System Programming Guide, Part 2, September 2014."

[52] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *ISCA-38*, 2011.

[53] D. Marr, F. Binns, D. Hill *et al.*, "Hyper-threading technology in the netburst® microarchitecture," *HotChips*, 2002.

[54] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE Micro*, 2010.