Reining in Server Application Tail Latency with Open Data Plane

Geoffrey Blake Luis E. Peña Pavel Shamis Curtis Dunham Eric Van Hensbergen

ARM Research

{Geoffrey.Blake,Luis.Pena,Pavel.Shamis,Curtis.Dunham,Eric.VanHensbergen}@arm.com

Abstract

Online, Data-Intensive (OLDI) services are a class of scale-out workloads that require all requests to be serviced as deterministically as possible. Occasional outliers in request latencies on one machine can impact the performance of the scale-out system as a whole and data-centers need to put strict Service Level Agreements (SLA) on these outliers to maintain acceptable performance across the entire system. However, maintaining strict SLAs across a scale-out workload is a difficult challenge. As a service scales, the SLAs become increasingly strict and, to meet these SLAs, servers are routinely run below full utilization. This untenable trend requires research into new programming tools for creating scalable applications with both high per-node utilization and predictable performance.

Scale-out OLDI workloads look increasingly like networking applications from a performance standpoint as well as from an execution standpoint because they are event driven by nature from IO. Therefore, leveraging techniques used in networking applications to manage tail latency and performance is an interesting line of investigation. In this work we present Open Data Plane (ODP), an API originally for high performance networking, that we have extended to support more general server applications, as the API for scale-out server applications. ODP presents an event oriented framework with asynchronous IO interfaces and built-in event scheduling for optimized load balancing. Using the extended ODP API we evaluate a ported version of Memcached, resulting in a 30% improvement, and a Proxy-Ceph application written using ODP, showing a potential 8x performance improvement over the base application.

1. Introduction

Modern OLDI services function by distributing work and data across many servers. A single user's request typically waits for the return and aggregation of answers from multiple servers before the final response can be sent. This dependence on responses from multiple servers makes the average response latency and total throughput of individual servers unrepresentative of the overall performance of an OLDI service. Instead, a server's 99th percentile or higher response latency (also known as tail latency) is a more accurate indicator of performance of a modern OLDI application. Tail latency can frequently be an order of magnitude or more higher than the observed average latency. The importance of tail latency is illustrated in Figure 1 from Dean and Barroso [10]. If an OLDI service is built using servers that respond within the desired Service Level Agreement (SLA) 99% of the time, across the aggregate of a few hundred machines a response will likely fail to meet the SLA for more than 50% of the incoming requests. In scale-out applications, the uncommon case for a single node is the common case for the entire service. For OLDI service providers, significant monetary costs are associated with responses that miss the SLA [11], so providers typically under-utilize and over-provision machines to maintain SLAs.

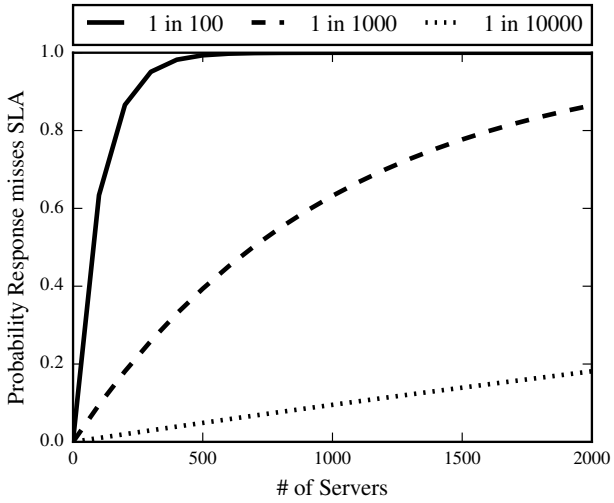


Figure 1: Probability a request to a scale-out service will finish slower than the SLA when the service is scaled to a given number of servers and composed of servers with different probabilities a single server may delay a response that misses its SLA: 1 in 100 requests to a single server takes longer than SLA, 1 in 1000 requests and 1 in 10000 requests.

Depending on the application, servers may only be utilized at 10% - 50% of their total capacity to allow meeting of SLAs [24], and these under-utilized resources cannot easily be recovered. Lo *et al.* [24] propose a solution that co-schedules batch applications onto machines running OLDI applications, and shows careful tuning of resource partitions allows the OLDI application to meet its SLA. It is much harder to co-schedule multiple OLDI applications onto the same machine because they are extremely sensitive to perturbations in the system, causing spikes in tail latency. To cope with this under-utilization, capacity is added by scaling the application out to more machines. Probability theory shows that as more machines are added, it only increases the likelihood that some machine misses its SLA. Therefore it is also necessary for the community to find new ways of increasing the capacity of single-nodes to scale up for OLDI applications.

Poor understanding of the causes of tail latency is one of the main reasons resources are overcommitted to an OLDI application. Tail latency contributors are difficult to find because, by definition, they happen rarely. Traditional profiling tools however, have been built for optimizing the common cases and are ill-suited to identifying tail latency sources. This has led to recent research into identifying tail latency sources for different applications [9, 21, 30]. New workloads have been proposed by Kasture *et al.* [20] to study the causes of tail latency by concentrating on latency critical OLDI type applications instead of long-running batch workloads.

These recent studies have concluded that tail latencies are not usually in the assumed critical path of the application. While optimizing the hot paths will deliver better average performance, it may leave 99th percentile latency unchanged. Tail latency examples include system preemption such as interrupts, queuing due to resource contention, rarely encountered code-paths or hitting corner cases in data-structures as described by Majkowski [27]. A study by Blake *et al.* [9] shows that Memcached's static assignment of connections to threads leads to a load imbalance with adverse effects on tail latency. Studies done on production data-centers have found tail latency hiding in the interactions of complex software making locally optimal decisions but globally sub-optimal decisions as shown in a presentation from Sites [29]. These multiple causes of tail latency will require systems researchers to look for comprehensive solutions to solve the problems with tail latency.

To provide part of this comprehensive solution, looking to the networking field will be valuable for finding ideas that can be moved over into the more general purpose environment of OLDI services found in the data-center OLDI server application computation is driven almost entirely by IO. Being IO driven naturally fits an event driven programming model that many high-performance networking applications use to attain their throughput and latency goals. In general, many server applications are still written in a traditional style with many threads and synchronous IO as writing asynchronous event-driven code is difficult with present tools. To continue scaling OLDI application performance while maintaining SLAs on the tail-latency, developers must look to different tools and techniques to optimize their applications.

This paper investigates the use of networking focused APIs in place of currently established APIs for server development and if they can increase machine utilization while still maintaining strict SLAs for tail latencies at the 99th percentile. For this investigation we choose the Open Data Plane [2] (ODP) API, with the aim to address these challenges. The main contribution of this work is to demonstrate ODP is a viable platform for OLDI server application development after adding the proper extensions to make it more generally applicable outside of networking. We present these modifications and the justifications for their inclusion into ODP later in this paper. In order to demonstrate the benefits of ODP we modify Memcached [13] and create a proxy application that mimics Ceph [33] using the ODP API as a replacement for the POSIX APIs. We show that we can get up to multiple factors of improvement in supported queries per second at a chosen 99th percentile latency SLA by leveraging the unique capabilities of ODP.

The rest of the paper is organized as follows: In Section 2 we describe the ODP API, why it was selected for study, and the modifications made to support general purpose server applications. In Section 3 we describe the modifications required to the applications studied. In Section 4 we evaluate the performance of ODP on a representative server platform. In Section 5 we discuss our experiences with the ODP API, in Section 6 we discuss related work and finally offer our conclusions in Section 8.

2. The Open Data Plane API

The ODP API is an open-source project from Linaro targeting cross-platform deployment of applications for software defined network data-planes. Specifically the library targets networking equipment such as cellular base-stations, networking middle-ware boxes like Virtual Private Network (VPN) concentrators and other functions that require high throughput and low latency packet processing. ODP applications are meant to run on anything from a general purpose server to specialized platforms that integrate numerous accelerators. ODP also promotes a more robust event-driven programming model which we believe is essential for building better tail-latency optimized OLDI applications. This makes ODP different from other software-defined data-planes like the Data Plane Development Kit [17] (DPDK) that focus on optimizing to a specific platform type and the common case.

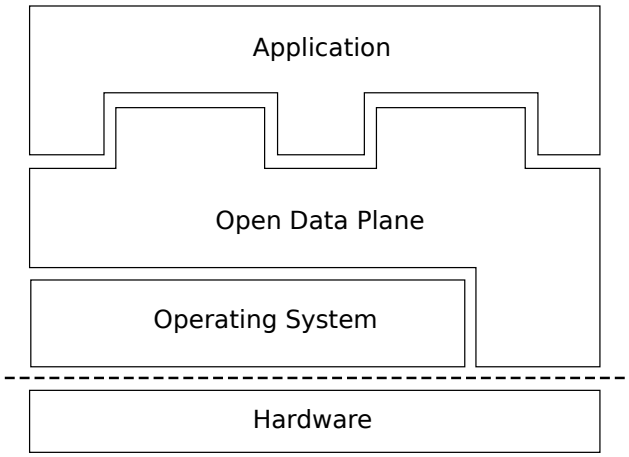


Figure 2: ODP is not a traditional API layer as it extends up into the application as well as offering OS-bypass modes if the kernel and hardware support such a mode.

2.1 API Design

Conceptually ODP does not fit into the traditional view of an API as an additional layer in the software stack for applications as seen in Figure 2. Instead ODP is an API layer with some tendrils that extend up into the application as they need to be tightly integrated with ODP to take full advantage of the API. This tight coupling is a product of ODP defining and promoting an event-based programming model to enable asynchronous IO and the eventual use of hardware offload. ODP also replaces parts of the Operating System (OS) kernel by offering some OS-bypass capabilities for accessing network IO.

ODP primarily supports an event-driven programming model as it is structured to provide IO and services in a completely asynchronous manner as well as allowing for parts of the API may be implemented by hardware offload. Figure 3 shows the flow of data through an ODP application and all the components of the ODP library. ODP offers a comprehensive suite of networking focused APIs such as direct packet IO from the hardware, buffer management, packet classification, queues, synchronization primitives, cryptography, egress traffic management, timers and event scheduling. ODP can be configured into a polling mode where the scheduler is not used and threads pull events directly off the queues they are managing, but as systems get bigger and more complex, we see the event based model being used more commonly to maintain the desired throughput and latency requirements by providing a mechanism for efficient load balancing.

2.2 Server Extensions API

While ODP provides efficient abstractions for event-based asynchronous programming of networking applications, we need additional API enhancements to improve ODP's suitability for server applications. These proposed enhancements are covered in detail in the following sections. These enhancements were inspired by our choice of applications to study, Memcached and Ceph. The two applications are different in the tasks they need to support, but similar in that they are both IO driven and appear to be a natural fit to leveraging the ODP programming model and library.

2.2.1 Layer 4 Network IO

One of ODP's major drawbacks is its lack of an integrated network stack for terminating network connections. To address this, we modified ODP to work with traditional TCP/IP sockets and the Linux kernel networking stack to provide network termination services to application programmers. The interface presented aligns well with the rest of ODP's asynchronous event-based model: the programmer sets a socket connection, attaches the socket to a pair of ODP queues, one for ingress and one for egress, and passes those queues to the scheduler. For the rest of the program the ODP API manages socket ingress and then passes events (in the form of received buffers) from the socket to the application via callbacks. For message egress, the application simply puts data into the ODP queue attached to the socket, and the ODP library sends the packet out. Partial message sends and reads are handled by the ODP runtime. Because an ODP application deals with callbacks that are passed an event, the ODP API has functions to gather data about where an event came from to support an application's decisions on where the response should be routed to.

Using the kernel's networking stack may be seen as a low performance option compared with an OS-bypass enabled stack, but we are primarily interested in reducing tail latency, not getting maximum throughput. While the kernel networking stack has been maligned in the past for being bloated and slow, it has been shown to not be a culprit in the tail latencies seen in modern OLDI applications [9]. Therefore using the kernel networking stack is acceptable to use in this library, even though processing of incoming and outgoing packets on the network interface card (NIC) hardware is out of the control of ODP. We leave it as future work to integrate a user-space networking stack that is under the full control of the application and evaluate if there is potential tail latency optimizations to be made with full control of the system from ingress to egress of events.

2.2.2 Layer 7 Packet De-multiplexing

Our second key addition to ODP is the inclusion of a de-multiplexer for incoming network messages. Applications using TCP/IP are provided a byte-stream abstraction, but server applications actually communicate in discrete messages, leveraging TCP/IP only for delivery. For Memcached, the application we modified for this study, a considerable part of its code is spent de-multiplexing messages from sockets. We enhanced ODP to support an application level message de-multiplexer API inserted between the Layer 4

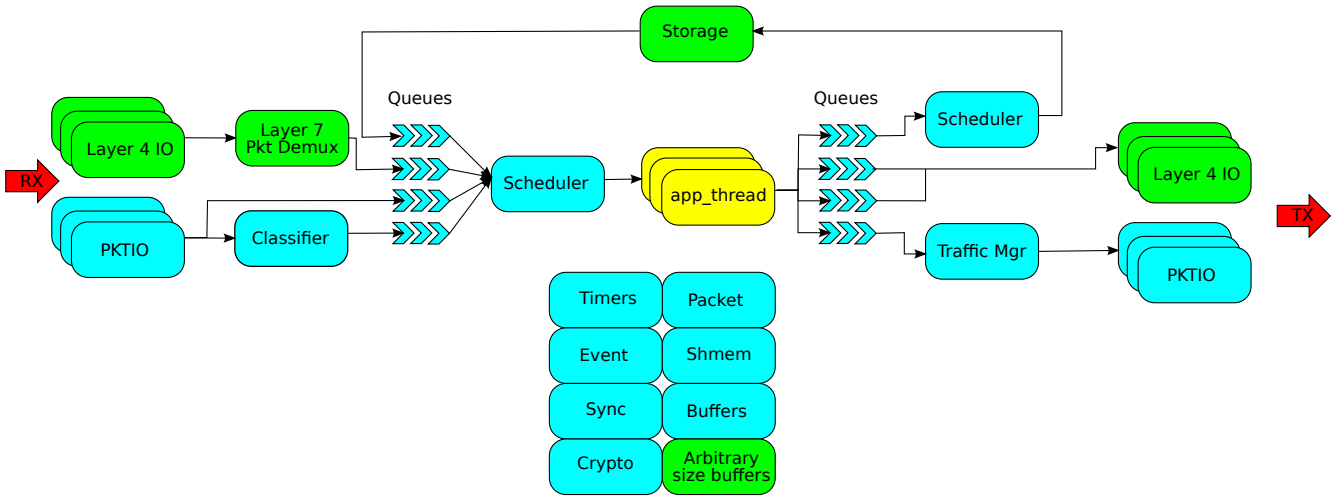


Figure 3: ODP's event flow. All input is routed through the scheduler. Application threads append into queues that feed output, or in the case of storage, back into the scheduler. ODP standard modules are highlighted in blue, our additions are highlighted in green.

input and the ODP queues attached to the Layer 4 input. As stream input is processed, it is passed through an attached packet de-multiplexer, and it processes the incoming stream-bytes into discrete sized messages. Figure 3 illustrates how the packet de-multiplexing is integrated into the ODP data-flow. The application has the option to define the packet de-multiplexing function in software, or use built-in de-multiplexers by setting up a parameters structure and passing it to the API.

This interface draws inspiration from ODP's classification API. For simple protocols, the de-multiplexer can be configured with a definition of byte-fields and offsets for determining the header and payload sizes. This is similar to the classification API already present in ODP that sorts incoming packets into different queues using their headers. Like the classification API, the packet de-multiplexing API is designed such that it could be implemented in hardware. For more complex cases like variable length headers or text processing, a software callback mechanism is also available for running code on the application processors.

2.2.3 Arbitrary Sized Buffer Allocation

When one first starts working with ODP, it quickly becomes apparent the library was designed with embedded systems and hardware accelerators in mind. All buffer pools are allocated at application start and tuned to the application and hardware requirements with a specific total buffer pool size, and each buffer to be allocated is a fixed size. This is a valid design point for an embedded networking data-plane as the programmer can determine *a priori* the buffer sizes needed throughout the program. This is further simplified by ODP's design point being Layer 3 networking and below, where packet sizes have well defined limits.

In contrast, defining fixed buffer sizes is not always possible for generic server applications such as the Ceph object-store or Memcached, which can access and receive arbitrarily-sized objects. One approach would be to determine the maximum sized buffer the application could possibly see, but this would lead to severely under-utilized memory resources if the common case is small and only occasionally a large buffer is needed. To fix this shortcoming in the ODP library, we implemented a memory allocator that could provide arbitrarily sized buffers by combining multiple fixed-sized buffer segments from the allocated buffer pools and modifying the API to take a size option for buffer allocation. We chose this design instead of building our own generic memory allocator like tcmalloc [15] or jemalloc [12] to provide an opportunity for memory allocation to be hardware accelerated. Memory allocation in the networking data-plane is sometimes done by hardware engines to help with throughput and latency. Hardware can be designed to deal with fixed-sized buffer pools and build scatter-gather lists as we built in software for ODP, but less so for general memory allocation that uses more advanced data-structures and interacts with the OS.

2.2.4 Storage API

Lack of access to storage was the final shortcoming in ODP that we rectified to make it suitable for our server applications. The current storage implementation in our ODP server extensions works on standard files, but again, we designed the interface with hardware offload in mind. To that end, the storage API is completely asynchronous. The application submits IO command descriptors into ODP queues and waits for completions to be delivered by the scheduler much like the Linux kernel asynchronous IO system call facilities. The ODP scheduler also decides when file IO requests are delivered to be processed in addition to delivering completions. As with the other asynchronous APIs defined here, an application registers a callback with the storage API to take delivery of IO completions. If storage were kept separate, we would face the unsolved problem of composing different event runtimes, so we chose to add storage extensions to what is fundamentally a networking API to maintain the invariant that all event sources must pass through a unified scheduler, which is key to successfully managing tail latency.

At present, our storage implementation serially and synchronously services IO operations from a queue; multiple IO queues allow parallel IO. Much like modern NICs hash packet headers to consistently route packet flows, we hash filenames to split IOs across queues. To prevent starvation in the case of heavy bursts of storage requests, which would cause tail latency to spike, the storage API limits bandwidth with a token bucket algorithm. Again, Figure 3 illustrates how the storage API is integrated into the ODP library and accessed by the application.

3. ODP Benchmarks

To evaluate the benefits of the ODP event programming model and asynchronous IO facilities for general purpose server applications, we created two benchmarks to evaluate the performance and scaling of ODP on current hardware. We ported the Memcached [13] key-value store to ODP instead of using libevent [26] and POSIX sockets. We also looked at the emerging area of software-defined storage by evaluating the Ceph [33] object-store and its tail latency in a similar manner to the work done for Memcached in a previous work [9] and wrote an ODP application that mimics the Ceph code base.

3.1 Memcached

Memcached is a commonly used key-value store in production data-centers to primarily cache web content in main memory. In such deployments, objects are sharded across many Memcached servers, so front-end servers will simultaneously query multiple servers to gather the content to be sent back to the user. Thus, tail latency should be optimized in Memcached installations, and responses typically must be received in short single-digit to sub-millisecond time frames.

As shown in previous profiles of tail latency for Memcached [9], the primary cause of tail latency is load imbalance between connections leading to standing queues on a small fraction of cores while others remain idle. Efficiently distributing load is one of ODP's strengths, so it follows that using ODP should benefit Memcached's tail latency performance.

Modifying Memcached to use ODP was relatively straightforward task because Memcached has already been developed as an event driven program using libevent. To make the modifications we eliminated most of the code in the `drive_machine()` function and instead let the ODP scheduler and framework select tasks, parsing network streams with the previously defined layer 7 stream de-multiplexer and sending responses through ODP queues. We also modified Memcached to use ODP allocated buffers where possible.

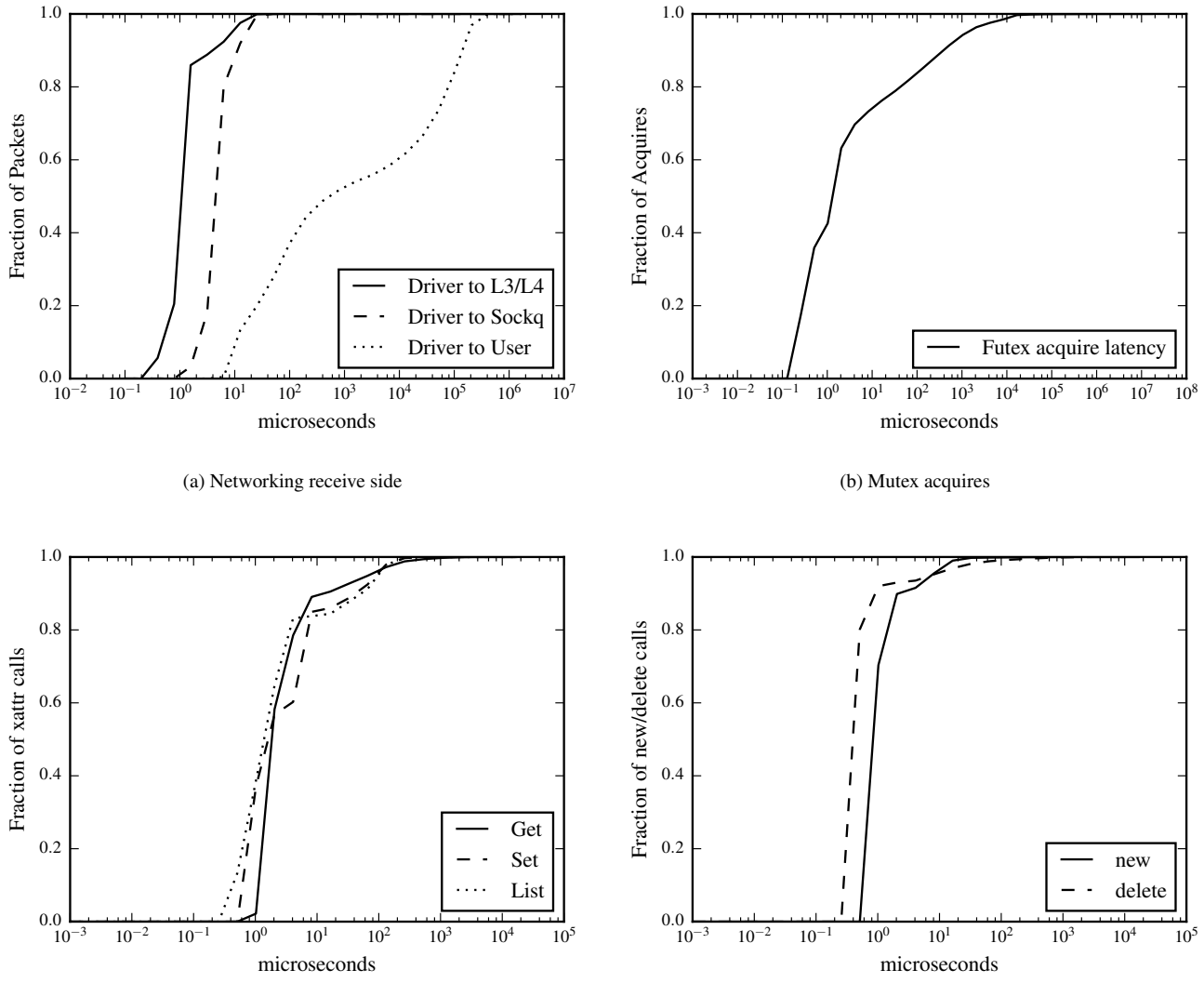
3.2 Object Store

We also looked into software-defined storage, as this is becoming a critical new data-center technology for building scale-out storage solutions from commodity machines as a replacement for traditional Storage Area Network (SAN) setups. Scale-out software-defined storage with commodity machines allows more flexibility when it comes to designing capacity, resiliency and performance. We chose the popular Ceph object-store as an additional platform to investigate how to reduce tail latency with the ODP library. Because one of the use cases for Ceph is to act as fast block storage for virtual machines (VM), and data is striped to multiple machines in a data-center, response latencies need to be low to provide responsive virtual machines, so tail latency is an important metric to optimize.

3.2.1 Tail Latency Study

We first studied Ceph to isolate its tail latency limitations. Similar to [9], we traced the latency of network messages with Systemtap [18], visualizing stack traces with Flame Graphs [16], and utilizing the "Linux Tracing Toolkit next generation" [1] tool to view the execution of a standard Ceph installation with 3 object store daemons (OSD) using out-of-the-box default configuration settings running on a many-core server using RAM Disks for storage. We chose to use RAM Disks in this study to concentrate on the tail latency induced by the system software stack and not the variable latency of a storage device. This is a valid methodology as future server systems will soon be using storage with access latencies in single digit micro-seconds or less with technologies such as storage class memory [14]. We study the application under heavy load to dilate tail latencies and make them easier to identify.

We found that Ceph sees tail latency coming from the application itself, just as was the case in Memcached, but for Ceph the load imbalance does not appear as the main contributing factor. In the case of Ceph (we experimented



(a) Networking receive side

(b) Mutex acquires

(c) Extended attribute ops

(d) Memory allocation

Figure 4: CDFs of the latencies experienced by Ceph on a many-core server platform for the main identified tail latency contributors: application receipt of network input, mutex acquires, extended attribute manipulation and memory allocation.

with v0.10.2), the design of the application is a main factor in its poor tail latency performance. Ceph uses the traditional threaded model of server application design, spawning hundreds of threads to service requests coming into the system, uses coarse grained synchronization, heavily loads the memory allocator with millions of small (64 bytes on average) allocations per second from all the threads, and leans heavily on OS services that were not built for frequent and concurrent accesses such as the extended attribute facility for filesystems. In our tail-latency study we found that 50% of the time of Ceph can be spent just in extended attribute manipulation of the file system, memory allocation and manipulating mutexes. So the common case is also impacted by these design decisions.

Figure 4 shows the effects on tail latency of these code paths as cumulative distribution functions (CDF) of latencies. Figure 4a shows that the main networking stack is particularly well behaved with a sharp curve to its cumulative distribution function with only a small amount of extra latency induced by the ”General Receive Offload” (GRO) facility in the kernel affecting the stack when passing packets to the IP layer. Most of the tail latency is due to the application as indicated by the third curve in Figure 4a, which shows the latencies of how long a packet waits in the socket queue before being copied into the application. Going further, a main reason for the application’s high tail latency spikes is seen in the distribution of mutex acquire times shown in Figure 4b. The Ceph code base spends many code lines imple-

menting its own event-based model constructed using mutexes and condition variables. Many of these mutexes are coarse grained and can suffer from contention and long acquire times as seen in the CDF plot. Other tail latency inducing code paths can be seen in the distribution of manipulating file system extended attributes in Figure 4c and the behavior of memory allocation operations in Figure 4d. All of these factors contribute to Ceph's poor tail latency behavior when configured to use fast storage.

All these design decisions lead to a code base that does not perform as well as expected. Until recently this code complexity would not have been an issue due to Ceph usually accessing rotating disk media with access latencies in the tens of milliseconds. Future faster storage like NVMe already starts exposing these code inefficiencies and technology like SCM will require Ceph to make significant code improvements.

3.2.2 Proxy-Ceph Microbenchmark

Because of the sheer size and complexity of the Ceph object-store code-base, we developed a Ceph like application using ODP we call Proxy-Ceph. The Proxy-Ceph application mimics the behavior of Ceph running in replication mode using ODP as investigated above. This allows us to study the performance of ODP running in an environment that is handling multiple sources of IO instead of trying to rewrite a large code-base with a fraction of the engineers and understanding of the Ceph developers.

Proxy-Ceph is written from the outset to be event driven, and therefore only uses as many threads as cores. This eliminates overheads of managing multiple in-flight threads using condition variables, mutexes and blocking IO calls as done in Ceph. Proxy-Ceph ties into all the ODP extensions developed for this paper and uses much of the existing ODP library functionality. By writing our application this way, it is completely event driven, all IO is asynchronous and scheduled using the ODP scheduler, therefore not relying on the OS scheduling policies to decide which threads to run when as is the case of Ceph's current software architecture. This allowed us to design the application using shared nothing parallel state machines to represent in flight transactions and eliminate much of the synchronization overhead we see in Ceph. Proxy-Ceph is a much leaner core piece of software, with 1,800 lines of code, compared to the almost 1 million lines of code in Ceph's source directory.

4. Evaluation

In this section we evaluate the performance of ODP by comparing the tail latency performance of each converted ODP application to the original application. For each application we measure the throughput for a defined target SLA latency at the 99th percentile. For all our experiments we use a custom version of ODP based on the linux-generic distribution of the API and do not leverage any customized platform optimizations beyond what the compiler generates for us at this time.

4.1 Experimental Setup

For our tests we have set up a testing rack of machines connected to 10Gbps networking to evaluate our baseline applications and ODP modified applications. For our evaluations we use a single Cavium ThunderX based platform with 48 ARMv8.1a cores running at 2GHz with integrated networking as our system-under-test (SUT). We use three drive systems generating load traffic to make sure we are measuring the SUT by keeping the drive systems lightly loaded. In this evaluation we show results for the SUT using all 48 cores.

To drive traffic into the SUT and measure tail latency we leverage the mutilate [7] utility which has been additionally modified to collect statistics and communicate with our Proxy-Ceph and Ceph applications, which use a different messaging format. For our testing we configured mutilate with the workload distributions described in Table 1. Our tests are structured to test small packet performance and primarily read performance of the benchmarked applications.

For Memcached we fit a general extreme value distribution to the microblog workload defined by Lim et al. [23] and chose a 1ms SLA for our target 99th percentile latency. This workload stresses small packet performance of the system instead of testing the bandwidth of the network infrastructure and will highlight the properties of ODP.

For Ceph and Proxy-Ceph we fit an exponential distribution to the SPECsfs 2008 [4] documented file size distribution to represent a potential workload for a Ceph object store that also is heavily slanted towards 90% reads. We pick a 5ms SLA that appears to be a reasonable target for remote storage to hit. For example, the CERN laboratory is building its Ceph clusters to hit this SLA [31]. As stated in Section 3.2.1, we test both Ceph and our Proxy-Ceph using RAM Disks formatted with the ext4 filesystem to simulate performance of software-defined storage using future storage technology. We configured Ceph using the out-of-the-box defaults. We configure three OSDs on a single machine and specify three copies of each object to be made. Proxy-Ceph is configured the same in terms of number of daemons created and the replication policy.

4.2 Results

In this section we will present the performance results of the stock applications and the ODP version of the applications for the chosen SLAs as defined in the previous section.

Memcached: The results for stock Memcached and Memcached modified to use ODP (Memcached+ODP) can be seen in Figure 5. Memcached using ODP is able to deliver 30% more performance at SLA than stock Memcached on our SUT with 64 clients connected through TCP. This performance improvement is being made primarily by ODP's ability to spread load dynamically to the more lightly loaded

Benchmark	Parameters	Description	Target SLA
Memcached	`-K uniform:250 -V gev:880,200,0.25 -u 0.1`	Microblog [23] key-value distribution in the 500B-2000B size emphasizing smaller packets with a 10% write fraction.	1ms gets
Proxy-Ceph	`-K uniform:250 -V exponential:0.00003333,1024 -u 0.1`	SPECsfs 2008 [4] like workload, files ranging from 1kB to MBs in size with a 10% write fraction.	5ms reads

Table 1: Mutilate [7] workload parameters

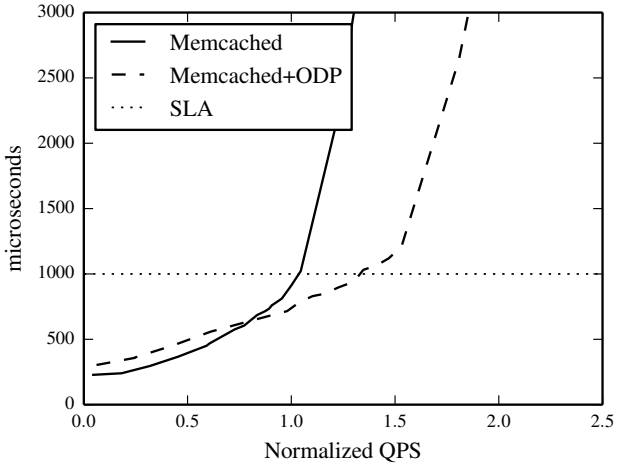


Figure 5: Measured queries per second on Memcached and Memcached+ODP with a target 99th percentile latency of 1ms using 48 cores and 64 total connections.

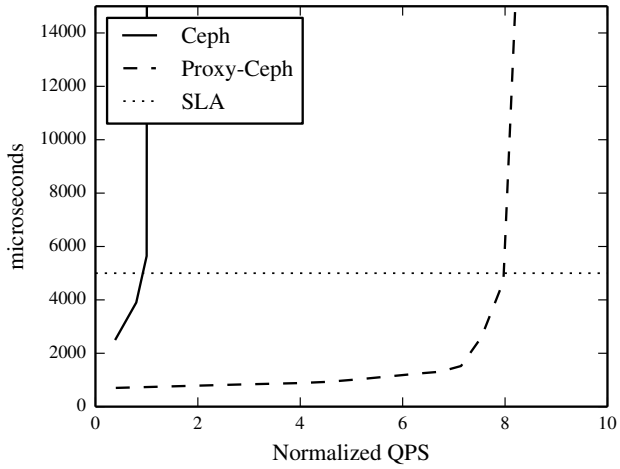


Figure 6: Measured queries per second on Ceph and Proxy-Ceph with a target 99th percentile latency of 5ms using 48 cores and 64 total connections.

cores in contrast to stock Memcached statically partitioning its load and suffering from imbalances in request lengths. This conclusion is being made from two observations about stock Memcached as shown by Blake *et al.* [9]: Memcached statically partitions its sockets to different threads, tail latency for Memcached is primarily caused by some sockets' queues being more full than others and the assigned thread not being able to process the back-log under the SLA. ODP in contrast, does not statically partition the incoming sockets, instead putting them into a global pool of connections to get data from for all threads. Each time a thread is ready to process data it calls the ODP runtime to find a batch of data for it to process. The scheduler makes sure to keep work distributed evenly among the threads according to how quickly they are processing data, meaning a fast thread will process more.

Additionally, Figure 5 also shows that under light load when both Memcached and Memcached+ODP are well under SLA, that Memcached is in fact faster than Memcached+ODP per request. This indicates that ODP with our extensions in its current form is adding overhead and hurting the common case because it requires Memcached to call into the global scheduler and go through some extra software layers. As the load on the server grows though, this extra overhead becomes less important as ODP's load balancing performs better at managing tail latency and allows extra utilization to be attained. This shows that for optimizing an application for tail latency, it may require going slower on average to go faster overall.

This is a significant result as it shows that using networking infrastructure programming models can benefit general server applications that require similar throughput and latencies as networking functions, but also match the execution model of networking applications by being primarily IO (event) driven. It should also be noted that this is still an unoptimized version of ODP as the code in the linux-generic branch has yet to be optimized in any way and the simple act of load balancing outweighs the addition of ODP overhead into the application.

Ceph and Proxy-Ceph: For the Proxy-Ceph benchmark we see large performance gains over a stock Ceph installation. Even when accounting for the 50% bottlenecks described in Section 3.2.1 our Proxy-Ceph application still outperforms Ceph by a substantial margin. The results in Fig-

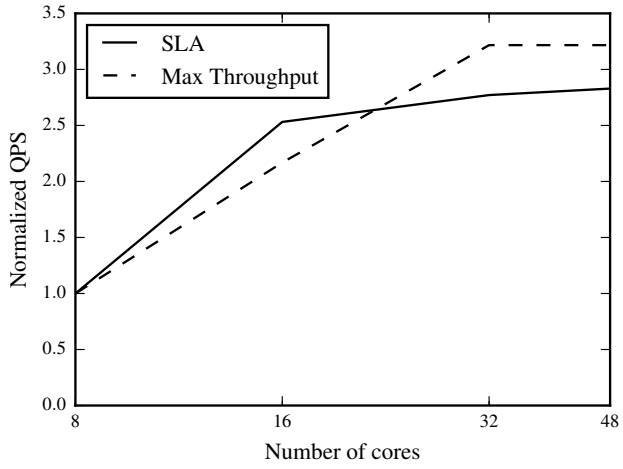


Figure 7: Scaling of Memcached+ODP when loaded with the Microblog workload, 0% write fraction, and 64 total connections for full bandwidth load and load that meets 1ms SLA.

ure 6 show a 8x improvement over Ceph at our 5ms 99th percentile latency SLA. [1]

This large performance difference is mainly due to all the efficiencies the event based programming model can offer to application developers. Instead of relying on the kernel to multiplex many threads to track outstanding transactions and performing intra-thread communication through pthread mutexes and condition variables, we use the event-based programming model from ODP to drive many parallel state machines that represented our outstanding transactions. From our inspection of the Ceph code base, a transaction's state and state transitions are instead embedded in the IO processing code instead of being self-contained, leading to code which is harder to understand and possibly less scalable. Writing in an event-based asynchronous framework forces a programmer to think differently on how to craft their application. During the exercise of creating Proxy-Ceph we found that we naturally designed a shared nothing coding style with minimal, fine-grained synchronization. This ultimately led to the more scalable program in the form of Proxy-Ceph over the stock Ceph application. This result shows that programming towards the model that ODP presents can result in much faster code, even when it is doing operations that are fairly heavy-weight, such as file-system operations and kernel networking.

ODP Scaling: Finally we look at the scaling performance of ODP itself on the SUT. ODP has synchronization points within itself that may limit scalability as server systems continue to add cores. For example, one can configure our SUT with up to 96 cores spread across 2 sockets. Systems from other vendors support hundreds of general purpose threads per socket, meaning future server applications have to be written with scalability in mind to get scale-up performance when needed. We study ODP's scalability using Memcached with the same packet distribution as defined in Table 1 but with no writes to the system, thereby eliminating as much synchronization contention in Memcached as possible. We then measure the performance of Memcached with ODP as more cores are added at full load and at a reduced load that meets our 1ms SLA. Figure 7 shows results of these tests. ODP sees linear scaling up to 32 cores in the overloaded case. At 48 cores we see no scaling, but further investigation of our results revealed that the system saturated the bandwidth of the 10Gbps network link. For the results at 1ms SLA, we see sub-optimal scaling past 16 cores. This implies that the ODP API and runtime itself can have a large impact on tail latency as more cores are added, likely due to the internal synchronization of ODP. The version of ODP tested was the linux-generic version from Linaro which is an all-software, unoptimized version of ODP. We believe there is large scope for improvement as optimized versions are developed by hardware vendors for their platforms and a software optimized version is created by Linaro.

ODP can offer significant advantages in tail latency by increasing utilization and increasing ultimate performance over traditionally programmed server applications as seen in our evaluation. By taking lessons learned from the networking infrastructure community, we can design solutions to the problem of tail latency in the data-center and extract more utilization from a single-node, thereby reducing the number of machines required in a scale-out application's provisioning. These lessons include using events as the main unit of work instead of managing threads, leveraging asynchronous processing whenever possible, and using efficient, automatic load-balancing. Using programming techniques developed by network application community also offers a path to potentially enabling hardware offload to be used more widely.

5. Experiences in Writing Applications with ODP

We have shown that writing applications towards the Open Data Plane library can attain performance benefits while optimizing for tail latency. ODP offers the promise of fully asynchronous IO, OS-bypass and transparent access to certain useful hardware accelerators, which should enable further performance gains in applications which must meet strict SLA requirements. Programming with ODP in general was not difficult, the API and programming model are well defined and documented. Creating both the Memcached port and the custom object store took only a few months of effort each for a single engineer. For Memcached most of the effort was spent finding all the places ODP API calls had to be inserted. For the Proxy-Ceph application, programming for ODP from the outset made it easier, and most of the ef-

[1] If accepted we will included numbers from an additional Proxy-Ceph application that is written with the same programming style as the Ceph source code for a fairer comparison.

fort was spent in standard software engineering tasks such as testing and debugging.

Yet there are still issues with the ODP API that will make it hard to see widespread adoption outside of some niche server applications. In this section we will offer some observations of programming with the API outside of the embedded networking domain that ODP was originally designed for.

The main short-coming of the ODP API is that it is targeted towards an embedded programming audience. It assumes the programmer will want to manage all their resources by hand and that the application is being written as the primary application for the system that is under complete control of the programmer. An example of this is buffer management in ODP, the developer must specify at start time how big they want their buffer chunks to be—which we presented a solution to for allocating arbitrary buffer sizes in section 2.2.3—as well as the maximum size of the buffer pool. This is hard to estimate for a sufficiently large application. We found when developing with ODP we had to over-provision memory resources to avoid running out of buffers. General purpose allocators grow and shrink their pools on demand yet this behavior can be shown to induce tail latency as well. The buffer allocation problem is made tricky in ODP in that it may have a hardware offload-engine supporting memory allocation behind the API. Additionally many tunable parameters inside of ODP are defined at compile time instead of dynamically. Again this stems from being designed originally to an embedded application domain where defining fixed resources and tunable options at compile time is acceptable practice.

Lastly the API is written with expert C programmers in mind, and while this allows fine-grained control of performance and code behavior and can be an advantage, it means ODP lacks some of the productivity syntactic sugar that newer languages provide to ease the development burden of event-based programs. ODP gives a developer the basic primitives to construct a high performance event-based program with the potential to leverage hardware accelerator engines transparently, but they are required to build all the machinery to support event based programs. ODP events offer a single opaque pointer for developers to store meta-data about how to process the event when it gets scheduled onto a core. This means a developer has to program features like continuations, lambda functions and futures that are available in more modern languages by hand to handle all the asynchronous operations ODP provides for all IO and processing. For expert programmers, this is not a major drawback since the potential performance gains may be worth the investment.

Overall ODP offers a new way for developers to think about and program high performance server applications. Because ODP targets the networking infrastructure segment, and server applications are increasing requiring performance that is close to networking requirements, it overall can increase productivity as it provides the high performance IO infrastructure and load balancing implementation built in.

6. Related Work

Tail latency has become an important research topic in the networking and server community recently because of its potential for impacting many current and future scale-out services once large scale has been achieved. Dean and Barroso's article *The Tail at Scale* [10] is an excellent introduction to the area and provides insight into what those in the industry are doing to combat the problems tail latency exposes. Tail latency research can be loosely categorized into four main avenues: network infrastructure, data-center scale, measurement studies and node-scale work.

Network infrastructure: Tail latency in delivering packets from source to destination is an important area as data-center networks have grown complex to offer the bandwidth and latency required in today's large scale deployments as is described by Singh *et al.* [28] detailing Google's data-center network. Work by Vulimiri *et al.*'s [32] internet focused work, or Alizadeh *et al.*'s [5] data-center focused work show that by sacrificing some bandwidth to either make redundant requests or actively keep network links under-utilized can improve the tail-latency characteristics of the underlying network-fabric serving the machines in the data-center. Other networking focused work looks specifically into redefining network protocols to alleviate tail latency. Examples include Zats *et al.*'s *DeTail* that hastens "flow completion" by improving congestion notifications between network layers [35] and Alizadeh *et al.*'s *pFabric* transport scheme where flow scheduling and rate control are decoupled [6]. These works concentrate on solving the problem of tail latency by improving the network fabric which is important, but also just a part of the solution for tail latency in the data-center. Our work is complementary to this work on reducing tail latency in the network.

Data-center scale: Other work describes how to solve tail latency issues at the level of the data-center itself. Dean and Barroso [10] while introducing tail latency as a problem for scale-out web-services also describe some solutions. Some of the key solutions involve dropping slow responses if the workload can tolerate partial responses, or sending multiple requests to redundant instances and selecting the fastest responder. The work by Lo *et al.* [24] shows how to co-schedule applications on the same server and preserve tail latency behavior by using hardware cache allocation facilities and to only allow batch jobs to fill in resources that a latency sensitive OLDI application is not using. Again our work is complementary to this line of research as increasing the load a single node can handle makes these techniques more effective.

Measurement studies: There has also been previous work in the area of identifying where tail latency occurs. Work from Blake *et al.* [9] characterizes Memcached and identifies where tail latency is coming from for that specific application. Zhang *et al.* [36] propose a new measurement methodology for tail latency and study micro-architectural factors that affect tail latency. Li *et al.* [21] study more applications for tail latency and also develop an analytical model to determine if tail latency can be modeled using queuing theory or if there are effects beyond classical theory to explain tail latency. Other work by Kasture *et al.* [20] proposes a new set of applications to benchmark techniques for reducing tail latency as well as doing initial investigations into the causes of tail latency. Xu *et al.* show that multi-tenant cloud servers exacerbate tail latency when latency-sensitive and compute-intensive workloads share resources [34]. Our work leverages these measurement studies and their methodology for conducting tail latency measurements of our own and developing solutions, but our work is mainly orthogonal to these measurement studies.

Single-node scale: Works that focus on optimizing a single node, as we do, are the closest related to ours. Considerable work has been done in the various node layers, from the application itself, the underlying runtime, and the OS. Work by Kapoor *et al.* [19] is the most similar to our work, looking at tail latency and attempting to optimize it for their target applications. They also develop a task scheduler but mis-identify the cause of the observed tail latency, attributing it to the OS, thus devoting most of the paper devoted to describing an OS-bypass networking layer. In contrast, our work targets a different cause of tail latency and attempt to solve it through changing the programming model and tools instead. Other work concentrates on particular layers of the software stack. Peter *et al.* [25] and Belay *et al.* [8] look at optimizing the OS to take better advantage of the hardware that is used today by allowing applications to manage self-virtualized hardware directly and moving the OS to only doing set-up functions. This work in the OS field is complementary to our work as it would eliminate sources of tail latency due to the OS. Other work looks at exhaustively optimizing an application to get average response latency improvements as well as tail latency improvements due to overall software enhancements. Work by Li *et al.* [22] show that by optimizing the entire software stack for Memcached, that a single node can scale up to service a large number of requests per second. This work is different from ours in that they concentrated on showing the ultimate improvement one could achieve for a single application by changing the software top-to-bottom completely. Lim *et al.* [23] look at optimizing Memcached using FPGAs to get large average performance improvement. In contrast this work studies the affects of changing the programming model and using tools from the networking community to construct better tail latency optimized applications, and not doing an exhaustive optimization of a chosen application.

7. Future Work

There are many avenues we left unexplored in this work. One area of future work is to optimize the ODP implementation to better match the platform we tested on instead of using the standard linux-generic implementation of ODP. We believe there are significant performance gains to be made in the ODP library itself that would increase the performance of our target applications and allow more utilization to be extracted at SLA.

Another area of future is to integrate OS-bypass into our server extensions. Currently we are layering ODP on top of standard Linux sockets and files for performing IO. For applications like Memcached and Ceph, they could take advantage of OS-bypass in the networking stack and in new storage devices like NVMe that promotes using virtualization to get easier access to the raw device for performance. We are looking as a first step in future work is to integrate the Open Fast Path [3] (OFP) user-space TCP/IP stack into our ODP implementation. An additional benefit to doing this OS-bypass work is that ODP will be able to manage the entire application in an event based manner and make better scheduling decisions for the specific application being run and optimize the schedule for tail latency. Currently the kernel pre-empts the application to do network IO, and this may be a poor choice if a request is being processed and is close to missing its SLA.

ODP also offers the promise of transparent integration with hardware acceleration. The API is designed the way it is to enable this use case for future hardware platforms to offer accelerators that applications can use without modification if they are written with ODP. Although there are no platforms currently available that offer such accelerator integration, we plan to look at hardware accelerators for the server domain more closely either with real hardware when it becomes available, in simulation, or emulated using some partitioned off processor cores as the accelerator engines. We plan to study the performance benefits and investigate solutions to the limitations hardware accelerators would place on applications designed with a write-once, deploy anywhere philosophy in terms.

Other future work that can be done is looking deeper into event based systems as a basic primitive for OLDI applications instead of time-sharing as the basic primitive. We believe there are definite advantages to integrating events more tightly in server systems from our work with ODP, and from the observation that networking applications themselves have to be event driven to get the required performance. A significant portion of the work would be in how to make events easier to work with, as we found ODP's very basic framework for constructing an application from events to be challenging.

8. Conclusions

Online Data Intensive applications have many characteristics in common with networking infrastructure applications. Execution is driven primarily by IO events and both are concerned with maximizing throughput while maintaining a strict SLA that is sensitive to tail latency, but the two application types are developed using fundamentally different methods. We believe that OLDI applications should borrow from the networking community the programming tools they use to boost performance and better manage tail latency.

In this work we show that programming OLDI applications with strict tail latency requirements using our modified Open Data Plane API can provide significant performance increases. We were able to maintain the desired SLA and control tail-latency better under more load than the same applications coded with traditional APIs and methods. We see an ODP version of Memcached and a custom Proxy-Ceph application written with ODP improve by 30% and 8x, respectively, compared with stock Memcached and Ceph on a state-of-the-art ARMv8 server platform. These improvements are shown by our analysis to be caused by ODP's asynchronous event-driven nature and the ability to dynamically and efficiently load balance incoming requests to the proper core. Additionally these performance results were shown with the linux-generic version of ODP which has no platform optimizations applied. Finding improvements by changing the programming model is necessary as server applications continue to require stricter performance guarantees to be scaled out to more machines.

References

[1] Linux tracing toolkit: next generation. URL http://lttng.org/.

[2] OpenDataPlane. URL http://opendataplane.org/.

[3] OpenFastPath. URL http://openfastpath.org/.

[4] SPECsfs2008. URL https://www.spec.org/sfs2008/.

[5] 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*, pages 253–266, 2012.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 435–446, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. .

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1097-0. .

[8] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4.

[9] G. Blake and A. G. Saidi. Where does the time go? characterizing tail latency in memcached. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 21–31. IEEE, 2015.

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

[11] K. Eaton. How one second could cost amazon $1.6 billion in sales. URL http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[12] J. Evans. Scalable memory allocation using jemalloc. *Notes by Facebook Engineering*, 2011.

[13] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[14] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4/5):439, 2008.

[15] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc, 2009.

[16] B. Gregg. The flame graph. *Queue*, 14(2):10:91–10:110, Mar. 2016. ISSN 1542-7730. . URL http://doi.acm.org/10.1145/2927299.2927301.

[17] Intel. Data Plane Development Kit (DPDK). URL http://www.dpdk.org/.

[18] B. Jacob, P. Larson, B. Leitao, and S. da Silva. Systemtap: instrumenting the linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008.

[19] 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.

[20] H. Kasture and D. Sanchez. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, September 2016.

[21] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3252-1. .

[22] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3402-0. . URL http://doi.acm.org/10.1145/2749469.2750416.

[23] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 36–47. ACM, 2013.

[24] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3402-0. . URL http://doi.acm.org/10.1145/2749469.2749475.

[25] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4.

[26] N. Provos and N. Mathewson. libevent—an event notification library, 2003.

[27] T. revenge of the listening sockets. Majkowski, marek, 2016. URL https://blog.cloudflare.com/revenge-listening-sockets/?p=0.

[28] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. . URL http://doi.acm.org/10.1145/2785956.2787508.

[29] D. Sites. Datacenter computers: modern challenges in cpu design, 2015. URL http://www.pdl.cmu.edu/SDI/2015/slides/DatacenterComputers.pdf.

[30] A. Sriraman, S. Liu, S. Gunbay, S. Su, and T. F. Wenisch. Deconstructing the tail at scale effect across network protocols. *Workshop on Duplicating, Deconstructing and Debunking*, 2016.

[31] D. van der Ster. Ceph at cern: A year in the life of a petabyte-scale block storage service. URL https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/ceph-at-cern-a-year-in-the-life-of-a-petabyte-scale-block-storage-service.

[32] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 13–18. ACM, 2012.

[33] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[34] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 329–341, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3.

[35] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: Reducing the flow completion time tail in datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 42(4):139–150, Aug. 2012. ISSN 0146-4833. .

[36] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, ISCA-43, New York, NY, USA, 2016. ACM.