Reducing Datacenter Application Latency with Endhost NIC Support

Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, Amin Vahdat†
UC San Diego †UC San Diego and Google, Inc.
{rkapoor,gmporter,malveeka,voelker,vahdat}@cs.ucsd.edu

ABSTRACT

In datacenter applications, predictability in service time and controlled latency, especially tail latency, is essential for building performant applications. This is especially true for applications or services build by accessing data across thousands of servers to generate a user response. Current practice has been to run such services at low utilization to rein in latency outliers, which decreases efficiency and limits the number of service invocations developers can issue while still meeting tight latency budgets.

In this paper, we analyze the three datacenter applications, Memcached, OpenFlow, and web search, to measure the effect on tail latency of 1) kernel socket handling, NIC interaction, and the network stack, 2) application locks contested in the kernel, and 3) application-layer queuing due to requests being stalled behind straggler threads. We propose a novel approach of reducing the above sources of latency by relying on support from the NIC hardware, and we find that the resulting improvements indeed dramatically reduce end-to-end application latency.

1. INTRODUCTION

Modern Web applications often rely on composing the results of a large number of subservice invocations. For example, an end-user response may be built incrementally from dependent, sequential requests to networked services such as caches or key-value stores. Or, a set of requests can be issued in parallel to a large number of servers (e.g., web search indices) to locate and retrieve individual data items spread across thousands of machines. Hence, the 99th percentile of latency typically defines service level objectives (SLOs); when hundreds or thousands of individual remote operations are involved, the tail of the performance distribution, rather than the average, determines service performance. Being driven by the tail increases development complexity and reduces application quality [41].

Within the datacenter, end-to-end application latency is the sum of a number of components, including interconnect fabric latency, the endhost kernel and network stack, and the application itself. Datacenter networks today can deliver both high bandwidth and low latency to better support scale-out applications [2, 1]. As a result, these interconnect fabrics themselves are not likely to be a significant source of latency unless they are heavily congested, and several efforts aim to minimize congestion, and thus latency [3, 4, 18]. On the other hand, the interwoven trends of increased cores per server, increased DRAM capacity, and the availability of low-latency, flash-based SSDs has the potential to reduce the latency of the application itself.

A key remaining component is kernel latency, which includes socket handling, NIC interaction, the network stack, and lock contention. As we will show, despite recent improvements in kernel performance [12], kernel overheads can be an order of magnitude larger than the datacenter network fabric and application latency combined. Thus, for a growing class of applications, reducing this kernel overhead is critical to reducing the effective latency of the end-to-end network path connecting clients to server applications.

We consider latency overheads for a number of applications, including Memcached [30], web search, and also an OpenFlow [29] controller. For these applications, the kernel latency overhead can account for over 90% of end-to-end application latency. This overhead also accounts for a significant source of latency variation, especially at high request loads. We apply an old idea to reducing this latency component–user-level/kernel-bypass networking [14, 49]. User-level networking has been available for decades, but it is now available with commodity NIC/OS support. While user-level networking removes kernel socket overhead, two substantial sources of overhead remain: lock contention and load balancing hotspots.

First, we partition requests based on fields in the application request as early as possible, in the NIC itself. This reduces lock contention, since multiple application threads can concurrently process requests that access shared state. Our goal is to partition requests among application threads so that each thread maintains exclusive read/write access to a partition of the state. While several systems implement this approach within the application itself [25, 32, 48], our approach is novel in that we leverage the NIC's capability to perform deep packet inspection to partition requests based on packet headers. This separates requests before they even arrive to the application itself, reducing latency and eliminating application queuing and contention for locks protecting

application work queues.

Second, we extend the in-NIC request partition logic with an extensible module to rebalance load away from overloaded CPU cores. Hotspots and skew in incoming requests can lead to application-layer queuing, and thus an increase in end-to-end latency. Even with an in-network load balancer forwarding flows across a large number of servers, balancing requests within a single server remains important since we wish to avoid the case where server performance degrades to the performance of the most loaded core.

Reducing the tail-latency of datacenter applications has the potential to improve the efficiency of distributed applications since more clients can be served from a limited set of resources. At the same time, developers will be able to issue more operations within a strict time budget, giving them freedom to develop complex applications. Based on a complete implementation and evaluation, we find that eliminating kernel, locking, and load balancing overheads can substantially improve data center application throughput and responsiveness. For example, we show how Memcached can support 200,000 requests per second with a mean operation latency of 10 μs with a 99th percentile latency of only 30 μs, a factor of 20 lower than unmodified Memcached. We find similar benefits for web search and the OpenFlow controller.

Substantial work remains before we are capable of delivering this level of performance in production data center deployments. For instance, we assume no multi-tenancy, uniform server deployments (with uniform performance profiles), and predictable network fabric performance. Given a large number of parallel efforts, e.g., low latency network fabrics and congestion control protocols [3, 4, 18] and virtualization containers [45] for predictable performance, our work addresses one of the key remaining bottlenecks for delivering end-to-end predictable low latency operations across the data center.

2. RELATED WORK

We now consider several related research efforts aiming to reducing end-to-end network latency and improve application performance.

Optimized Network/OS interfaces: A key bottleneck that our work addresses is kernel and network stack overhead. We share this goal with several industrial efforts. Myrinet [10] is a message-passing network that supports flow control and error control, low-latency forwarding with cut-through switches, and an optimized network/OS interface designed for low latency. It specifically supports direct I/O from user-space application directly to the network, which has been subsequently adopted into Ethernet. Infiniband is a low-latency, high-bandwidth switched interconnect fabric created by the Infiniband Trade Association [24] that is often used in high-performance computing clusters, with a particular focus on efficient and low-latency interfaces, even supporting remote DMA operations [38]. While these efforts address a key bottleneck, our approach focuses on commodity Ethernet switching and the entire end-to-end application latency path, including application lock contention and hotspots.

User-level Networking: User-level networking was developed to support application which emit packets at a high rate, and to remove latencies in the kernel [49]. This need was especially pressing to support the high packet rates needed by ATM networks, where operating system overhead was a major performance bottleneck. The feasibility of applying user-level networking to both ATM and Ethernet was further studied in [50], and an analysis of latency in the end-host network stack was carried out by Larsen et al. [27]. This interface was extended by Virtual Interface Architecture [14], in which processes use a virtual interface to arrange for data transfers directly to remote virtual memory using a form of zero-copy RDMA, reducing the programming burden. Alpine [20] took a different tact by making user-level development easier by enables pulling some in-kernel functions, such as the TCP state machine, into user-space, and interfacing to that code via the user-level APIs. While user-level networking APIs are key to the early partitioning aspect of our design, we also focus on per-CPU core load balancing and removing application lock contention through deep-packet inspection using these APIs to reduce application tail latency.

Operating System Improvements: In this paper we highlight several performance bottlenecks within the kernel. Several projects are focusing on improving the scalability and performance of the Linux kernel. Corey [11] identifies numerous instances of in-kernel data structure sharing which reduces potential parallelism across threads, and proposes address ranges, kernel cores, and shares to address this. Boyd-Wickizer et al. [12] study the scalability of seven applications, including Memcached, across a 48-core computer. They found that by modifying the kernel and applications themselves, it is possible to remove many performance bottlenecks present in these components. However, their study focuses on throughput, and not latency. Zhuravlev, Blagodurov, and Fedorova [52] show that it is possible to mitigate in-kernel shared contention through intelligent scheduling, and Ruan and Pai [40] investigated the latency characteristics of Flash and the Apache Web server, finding that locking and blocking system calls were significant causes of application performance degradation. In our work, we find that even for single-threaded processing the kernel introduces significant additional latency, even after accounting for these recent improvements.

Lock Contention: Lock contention has long been recognized as a key impediment to performance of shared memory, multi-threaded applications [46]. Rather than relying on mutual exclusion, some lighter-weight locking strategies are feasible for some applications, such as read/write locks. However, replacing mutex locks with read/write locks has the potential of increasing state requirements [15]. Triplett et al. [47] propose a dynamic concurrent hash table with re-

sizing using a *Read-Copy Update* (RCU) mechanism. This mechanism works well in situations where the number of reads are significantly greater than writes. CPHASH [32] is a concurrent hash table partitioned across CPU L1/L2 caches, in which clients perform operations on partition by sending the message through shared memory to the right partition. CPHASH achieves 5% better throughput as compared to memcached, but latency is not the primary focus of their approach. Similarly, VoltDB [48] and H-Store [25] partition application state in memory across the CPU cores to achieve scalability. Here, incoming requests are partitioned at the application layer after arriving to the process. Our approach is different in that we rely on deep-packet capabilities of the NIC hardware to partition requests before they arrive to the OS or application.

Datacenter Networks: Unlike wide area networks, the datacenter network fabric can potentially support very low latency communication between the servers since these networks are confined to a single geographic location, resulting in a very small propagation delay. Low-latency, cut-through switches support sub-microsecond packet forwarding [6, 16]. New network topologies [1, 2, 22] can help reduce the distance packets have to travel from one server to another by reducing the number of switch hops, as compared to an enterprise or ISP backbone topology. In addition, new transport and networking protocols like DCTCP [4] and QCN [3] further reduce in-network queuing and congestion, and thus reducing the network latency. Ongoing work such as Detail [18] also focuses on reducing latency by performing in-network traffic management.

Datacenter Applications: Our approach has the potential to benefit a variety of datacenter applications. One class, key-value (KV) stores, serve as a basic building block for building loosely-coupled distributed systems. A widely deployed KV store is Memcached. Memcached stores small chunks of unstructured data, and exports a simple API consisting of operations that get, set, delete, and manipulate KV pairs. KV pairs are stored entirely in-memory, with no persistence. Memcached deployments split functionality between one or more servers and a number of clients. KV-pairs are partitioned across the set of servers using a hash function shared between the clients and the servers. In Memcached, clients are independent, and issue requests directly to a single server responsible for a particular key based on the shared hash. This simplifies the design of the distributed server tier, since servers do not need to communicate with one another. Clients are responsible for inserting KV-pairs into the cache, as well as Thus, scaling the server tier is trivial since Memcached does not need to ensure cache consistency or invalidate data. deleting or invalidating them according to the semantics of the application. For high-throughput environments, requests are typically issued using UDP [42] to reduce latency and to require fewer kernel resources than would be needed to support the same number of TCP connections.

Two thousand Memcached instances have been deployed at Facebook [41]. There have been numerous efforts to improve its throughput [8, 42], though none specifically looking at predictable tail-latency. Other KV-stores have been deployed with different storage semantics. Dynamo [19] is an eventually-consistent KV-store supporting a high insertion and query rate while surviving failures of individual components. Other persistent KV-stores include BerkeleyDB [9], LevelDB [28], and Redis [39]. The applicability of our approach is based only on the relative latency of the application compared to the kernel overhead, and so the logic of these applications can be treated largely as a black box.

Another application we consider is web search, which is a well-studied problem, with numerous scalable implementations. Fox et al.[21] describe the principles underlying the HotBot commercial web search engine. In [13], Brin and Page describe the Google web search engine. We choose web search given that it is a good example of a horizontally-scaled datacenter application. Finally, the third application we consider is an OpenFlow [29] controller. This application is different from Memcached and the web search application in that it is typically not horizontally scaled. However, given that the OpenFlow controller can be on the critical path for new flows to be introduced into the network, its performance is critical, even if the entire application is only deployed on a single server. Several controller designs have been proposed with low-latency in mind [23, 26, 51]. We feel that this set of case studies is significantly diverse to represent a range of realistic and widely-deployed datacenter applications.

3. BACKGROUND AND MOTIVATION

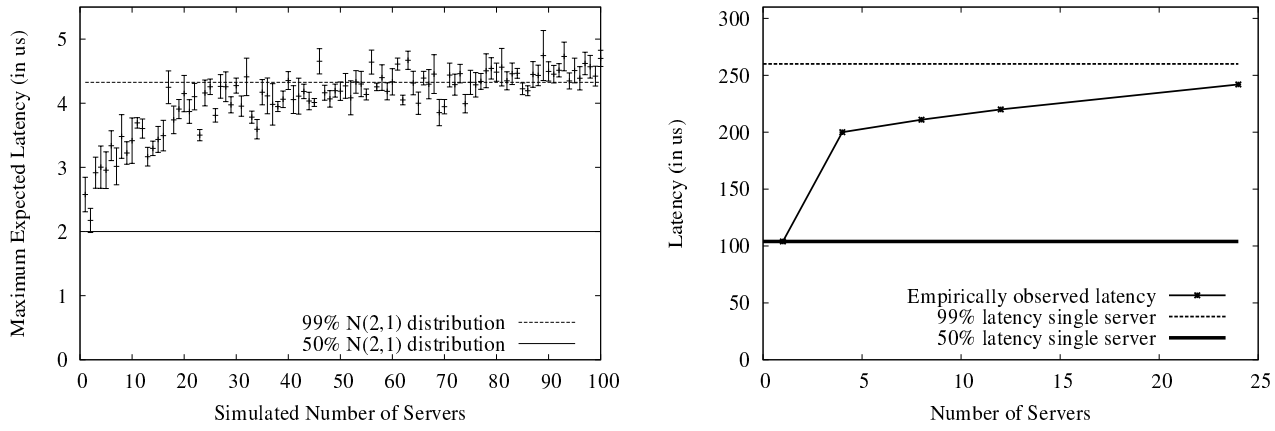
In this section we discuss the effect of latency and high latency variation on datacenter communication patterns and how that impacts the end-to-end performance and operation of data center applications. We then describe the different sources contributing to latency in the data center context.

3.1 Effect of Latency on Datacenter Communication Patterns

We begin by describing two general, yet pervasive, communication patterns present in datacenter networks, and the consequences of variable latency on their performance.

3.1.1 The Partition/Aggregate Pattern

Network communication patterns in which state must be retrieved from a large number of servers in parallel before a response is returned to the requesting service falls into the Partition/Aggregate communication pattern. An example of this pattern is a horizontally scaled web search query that must access state from hundreds to thousands of inverted indices to generate the final response. It is well-known that the achievable service-level objective of an application relying on this pattern is limited by the slowest response generated, since all requests must complete before a response can be



(a) Predicted by probabilistic analysis.

(b) Empirically-observed.

Figure 1: **As the scale of the Partition/Aggregate communication pattern increases, latency increases due to stragglers.**

sent back to the user. In practice this means that the latency seen by the end user approaches the tail latency of the underlying services, and in this section we show this straggler behavior experimentally.

Analysis: Consider a client issuing a single request to each of S service instances in parallel. For simplicity, we assume the service time is an independent and identically (i.i.d.) distributed random variable with a normal distribution. Consider an S-length vector of the form:

$$\vec{v} =< N(\mu,\sigma), N(\mu,\sigma), ..., N(\mu,\sigma) >$$

where $N()$ is the normal distribution, and $\mu = 2\mu s$ and $\sigma = 1\mu s$ (these values are based on our observations of Memcached's latency, described in Section 3.2). We estimate this system by computing values of the random variables for each set of i variables where i ranges from 1 and 100. For each set we compute the maximum over the values of the variables in the set, repeating each measurement five times to determine the latency and variance. The result is shown in Figure 1(a).

As the number of servers increases, the maximum observed value in $\vec{v}$ increases as well. We also plot the 50th and 99th percentiles of the underlying $N(2,1)$ distribution. In this simulation, when the number of servers is small, the maximum expected latency is close to the mean of 2 μs (the 50th percentile of the random variable). However, as S grows the maximum observed value is approximately $4.25\mu s$, which approaches the 99th percentile of latency of the underlying distribution. Thus, the end-to-end latency of the Partition/Aggregate communication pattern is driven by the tail-latency of nodes at scale.

Experimental validation: To validate the above probabilistic analysis, we perform the following experiment. We set up six Memcached clients, each on different machines, and measured the latency seen by one of these clients. The results of this experiment are shown in Figure 1(b). Each client issues a set of S parallel *get* requests to a set of S server instances (where S ranges from 1 to 24). Each server instance runs on its own machine. We used the *memslap* load generator included with Memcached to generate requests uniformly distributed across the key-space at a low request rate, so as not to induce significant load on the servers. Also shown is the observed single-server median latency (approximately $100\mu s$) and the 99th percentile of latency (approximately $255\mu s$).

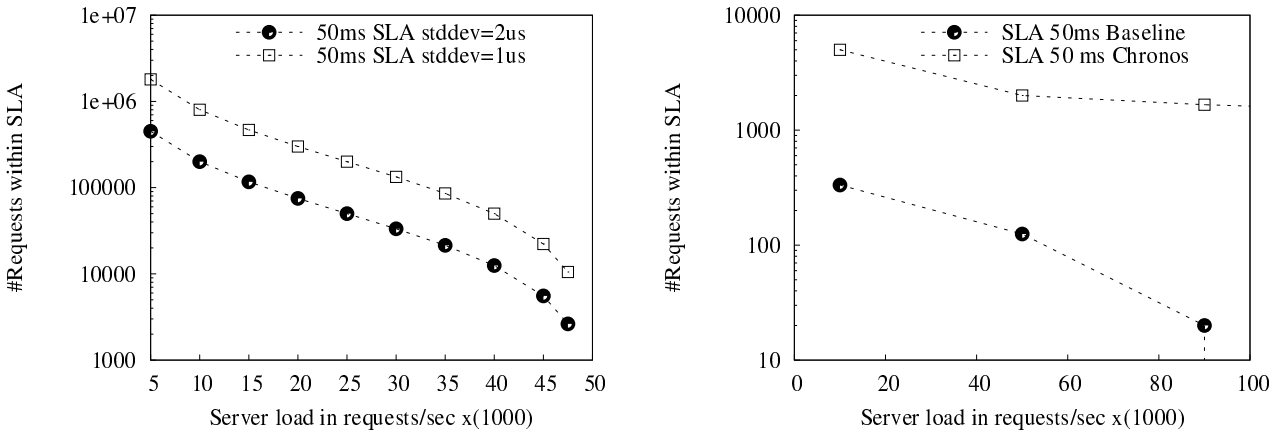
As predicted, when issuing a single request to a single server the observed latency is nearly the 50th percentile of service time. However, as S increases, the observed latency of the set of requests quickly approaches the long tail of latency, in this case just below the 99th percentile. This effect can be seen even when we issue a low request rate, since when the application latency is as small as one microsecond, variable latency in other components of the end-to-end path have a pronounced effect on the flow. As server utilization increases, the tail-latency becomes even higher.

3.1.2 The Dependent/Sequential Pattern

A second network communication pattern in datacenters is the dependent/sequential workflow pattern, where applications issue requests one after another such that a future request is dependent on the results of previous requests. Dependent/sequential patterns force Facebook to limit the number of requests that can be issued to build a user's page to between 100 and 150 [41]. The reason for this limit is to control latency, since a large number of sequential requests can add up to a large aggregate latency. Another example of this pattern are search queries that are iteratively refined based on previous results.

In both cases, increasing the load on the subservices results in increased service time, lowering the number of operations allowed during a particular time budget. This observation is widely known, and in this section we show how it can be validated both through a queuing analysis as well as a simple microbenchmark.

Consider a simple model of a single-threaded server where



(a) Predicted by queuing analysis.

(b) Empirically-observed.

Figure 2: **For the Dependent/Sequential communication pattern, the number of subservice invocations permitted by the developer to meet end-to-end latency SLAs depends on the variance of subservice latency.**

clients send requests to the server according to a Poisson process at a rate λ. The server processes requests one at a time with an average service time of μ. Since the service time is variable, we model the system as an M/G/1 queue. Using the Pollaczek-Khinchine transformation [5], we compute the expected wait time as a function of the variance of the service time using

$$W = \frac{\rho + \lambda\mu Var(S)}{2(\mu - \lambda)}$$

where $\rho = \lambda/\mu$. This estimate differs from the M/M/1 case (which is used when there is no variation of service time) in that it has a variance term in the numerator. Based on this model, we can predict the service latency as a function of service load, mean latency, and the standard deviation of variance. To observe the effect of latency variation, we evaluated the model against $\sigma = 1$ (based on our observations of Memcached), and $\sigma = 2$ (representing a higher variance service). For each σ value, we use the model to compute the latency, and from that, we compute the number of of service invocations that a developer can issue while fitting into a specified end-to-end latency budget, and plot the results in Figure 2(a). As expected, that budget is significantly reduced in the presence of increased latency variance.

To validate this model, we compare the prediction of the number of permitted service invocations to the actual number as measured within a deployment of Memcached within our testbed, shown in Figure 2(b). The experimental setup and experiments are described in detail in Section 5.1. Here, we measure the 99th percentile of latency for both baseline Memcached, as well as Memcached implemented with Chronos (CH). Each point represents the number of service invocations permitted with the specified SLA, as a function of the server load, in requests per second.

These simple studies confirm the intuition that delivering predictable, low latency response requires not just a low latency mean, but also a small variation from the mean.

3.2 Sources of End-to-End Application Latency

To better understand the latency bottlenecks in data center applications, we profile a representative example of an in-memory key-value store, Memcached. We describe each of the components that contribute to the end-to-end latency within Memcached, as shown in Table 1.

Datacenter Fabric: The datacenter fabric latency is the amount of time it takes to traverse the network between the client and server. This can be further decomposed into the propagation delay and in-switch delay. Within a datacenter, speed of light propagation delay is approximately 1us. Within each switch, the delay is approximately 1-4 us. Low-Latency, cut-through switches further reduce this packet forwarding latency to below one microsecond. A packet from client to server typically traverse 5-6 switches [2].

A packet can also suffer queuing delay based on prevailing network congestion. We calculate the queuing delay by measuring additional time packet waits in switch buffers. Typical commodity silicon might have between 1-10MB buffers today for 10Gb/s switches. However, this memory is shared among all ports. So for a 32-port switch with relatively even load across ports and with 2MB of combined buffering, approximately 64KB would be allocated to each port. During periods of congestion, this equates to an incoming packet having to wait for up to 50 μs (42 1500-byte packets) before it can leave the switch. If all buffers along the six hops between the source and destination are fully congested, then this delay can become significant. Several efforts described in section 2 aim to minimize congestion and thus latency. We expect that in common case, the networks paths will be largely uncongested. While potential end-to-end bottlenecks such as delay in the datacenter fabric are outside the scope of this effort, the value of Chronos is that it addresses a key remaining bottleneck to delivering low-latency services.

Component	Description	Mean latency (μs)	99 %ile latency (μs)	Overall share
DC Fabric	Propagation delay	< 1	-	-
	Single Switch	1-4	40-60	1%
	Network Path†	6	150	7 %
Endhost	Net. serialization	1.3	1.3	1.4 %
	DMA	2.6	2.6	3 %
	Kernel (incl. lock contention)	**76**	**1200-2500**	**86-95** %
Application	Application*	2	3	2 %
	Total latency	88	1356-2656	100 %

Table 1: **Factors that contribute to latency in datacenter communication. †The network fabric latency assumes six switch hops per path and at most 2-3 switches congested along the path. Switch latency is calculated assuming 32 port switch with a 2MB shared buffer (i.e., 64KB may be allocated to each port). Application latency is based on Memcached latency.**

End-host: Endhost latency includes the time required to receive and send packets to and from the NIC, as well as delivering them to the application. This time includes the latency incurred due to network serialization, issuing a DMA for the packet from the NIC buffer to an OS buffer, and traversing the OS network stack to move the packet to its destination userspace process.

To understand the constituent sources of endhost latency, we profile a typical Memcached request, determining the underlying components contributing to the observed latency under load. We issued 20,000 requests per second to the server, which is approximately 2% network utilization in our testbed. We instrumented Memcached 1.6 beta and collected timestamps during request processing. To measure the elapsed server response time, we installed a packet mirroring rule into our switch to copy packets to and from our server to a second measurement server running Myricom's Sniffer10G stack, delivering precise timestamps for a 10 Gbps packet capture (at approx. 20ns resolution). Section 5 presents full details on the testbed setup. We then measured the Memcached application latency by wrapping timer calls around the application. We record The start time of this measurement immediately after the socket *recv()* call is issued; the end time is measured just before the application issues the socket *send()* call.

A median request took 82 μs to complete at low utilization, with that time divided across the categories shown in Table 1. *Network Serialization Latency* is based on a 100B request packet and a 1500B response at 10 Gbps. *DMA latency* is the transfer time of a 1600B request and response calculated assuming a DMA engine running at 5 GHz.

Application: This is the time required to process a message or request, perform the application logic, and generate a response. In the case of Memcached, this includes the time to parse the request, look up a key in the hash table, determine the location of value in memory and generate a response for the client. The application latency in Memcached is 2 μs. In section 3.3 we discuss other factors that contribute to application latency, including lock contention.

The remainder of the time between the observed request latency and the above components includes the kernel networking stack, context switch overhead, lock contention, kernel scheduling overhead, and other in-kernel, non-application endhost activity. The contribution of kernel alone accounts for more than 90% of the end-host latency and approximately 85% of end-to-end latency. In the next section, and in rest of the paper, we focus our efforts on understanding the effect of kernel latency on the end-host application performance, aiming to reduce this important and significant component of latency.

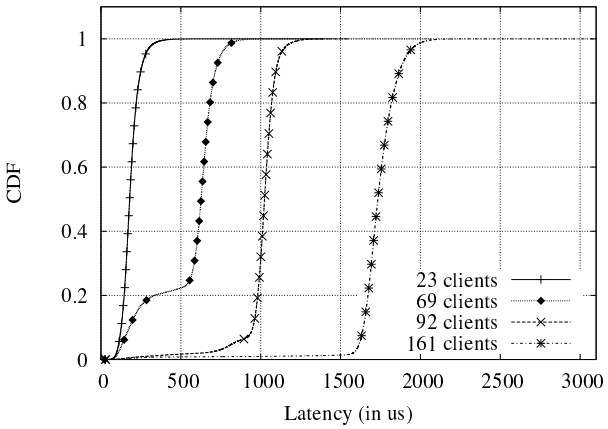


Figure 3: **The latency of a Memcached server running at 10% of network capacity. A fixed number of closed-loop clients were used to generate traffic. Increasing the concurrency leads to increased lock contention inside the OS, resulting in higher application latency.**

3.3 End-to-End Latency in Memcached

In this section we see how the latency overheads shown in Table 1 impact the end-to-end latency of Memcached. We first measure the maximum load that a server can handle and then observe its performance under different load characteristics, and with varying level of concurrency. Recall that the level of concurrency affects the in-kernel overhead.

To measure Memcached performance, we use a config-

urable number of *Memslap* clients, which are closed-loop load generators included with the Memcached distribution. Each client is deployed on its own core to lower measurement variability. We observe that Memcached can support up to 120,000 requests per sec. We next subject it to a fixed request load, and observe the distribution of latency. We evaluated the server at a low request load of 40,000 requests per second, which is approximately 40% of the server's maximum throughput, and also at a high load of 90,000 requests per second, or about 90% of its maximum throughput. On each of the 23 client machines, we reserve one CPU core for Linux, leaving seven for client instances, which means we can support up to 161 clients.

At low (40%) utilization, increasing the number of clients had little effect on distribution of latency, which remained small (not shown). Most responses completed in under 200 μs, with the tail continuing up to approximately 400 μs. This corresponds to lower levels of load at which developers run their services to ensure low tail-latency. However, at high (90%) utilization, which is still only 10% of network utilization, increasing the number of clients had a pronounced effect on observed latency. In fact, it caused it to increase dramatically as the number of clients increases, reaching a maximum at about 2000 μs, shown in Figure 3. Two sources of this queuing are variance in the application response time, and an increase in lock contention within the application due to increase in concurrent requests. We used the mutrace [34] tool during runtime to validate this last point. We discuss and evaluate this aspect in more detail in section 5.1.

These measurements aid our understanding of current practices of running services at low levels of utilization. To operate these services at higher utilization requires necessitates reining in the latency outliers. In this case, increased queuing in the application plays a significant role.

4. DESIGN

In this section we describe Chronos, an architecture for low-latency application deployment which can support high levels of request concurrency.

4.1 Goals

Our goal is to build an architecture with these features:

1. **Low mean and tail latency:** A key to increasing the efficiency of endhost servers is reducing unnecessary latency, which Chronos accomplishes by reducing the overhead of handling sockets and communication in the kernel, as well as removing lock contention and per-thread application hotspots.

2. **Support high levels of concurrency with reduced or no lock contention:** Reduce or eliminate application lock contention by partitioning requests across application threads within a single endhost, such that multiple instances/threads minimize accesses to shared state.

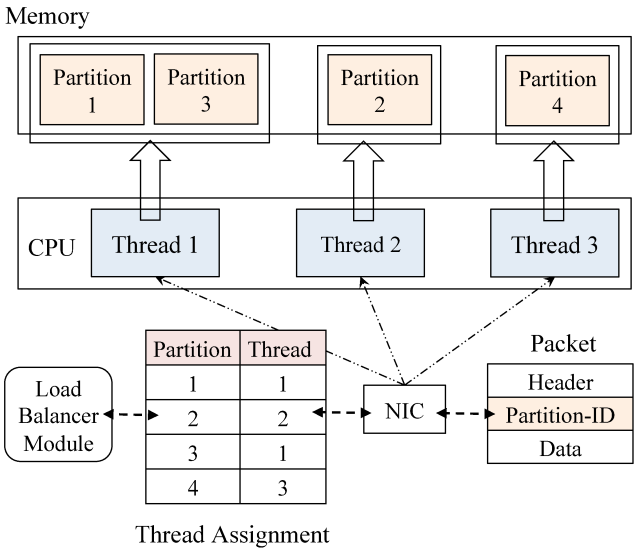


Figure 4: **Chronos system overview. Each incoming request is examined for a partition ID field in the application header, and dispatched to an application thread based on consulting a lookup table, which is populated by the load balancing module. Request is delivered directly to an application thread using a kernel-bypass mechanism. The load balancing module maintains a mapping of partition IDs to hardware threads, and periodically updates this mapping based on traffic conditions.**

3. **Early assignment and introspection:** We seek to partition incoming client requests as early as possible to avoid application-level queue build-up due to request skew and application lock contention. We leverage hardware support in the NIC to determine application semantics through deep packet inspection.

4. **Self tuning:** Rather than assuming a priori knowledge of the incoming traffic pattern and application behavior, Chronos load balances requests within a single node across internal resources to avoid hotspots and application-level queuing.

4.2 Design and Implementation

We now describe the design of Chronos. Its request servicing pipeline is carried out in three stages: request handling, request partitioning and load balancing.

Request handling: As described in Section 3.2, a major source of latency in end host applications is the operating system stack. We eliminate this overhead by moving request handling out of the kernel and entirely into user-space by employing user-level, kernel-bypass, zero-copy network APIs, which are available from several vendors today [35, 37, 43, 44]. At load time, the NIC driver allocates and maintains a region of main memory dedicated to storing incoming packets. This memory is mapped as circular send

and receive ring buffers, called NIC queues, in the application's address space. The receive-side scaling (RSS) feature of these NICs permits the administrator to divide a given physical interface into rings, each providing exclusive access to its own send and receive queue. Outgoing packets are enqueued into a selected ring, which are sent on the wire based on by a scheduler implemented by the NIC. Incoming packets are by default forwarded to a queue based on an inbuilt hash function, however this can be customized by the user. The use of a custom hash function provides flexibility in hashing on arbitrary packet offsets to determine the destination queue. The execution of this hash function is done in userspace on any one of the CPU cores, and for simple hash functions the execution time is in nanoseconds, which is less than the packet inter-arrival times for 10 Gbps links. When a packet arrives from the wire, it is copied to the queue chosen by the classification function. The application is not notified when packets arrive with an interrupt, but must instead poll for new packets using a *receive()* API call. A dedicated thread monitors the NIC queues and registers packet reception events with the application.

Request partitioning: Bypassing the kernel results in significant reduction in latency, now that a request can be delivered from the NIC to the application in as low as 1-4 μs. However, this reduction in packet transfer latency exposes new application bottlenecks such as lock contention and instantaneous core overload, or processing hot spots, due to skew in the requests. Reducing application lock contention relies on separating requests that manipulate disjoint application state as early in the endhost as possible. A classic approach to minimizing lock contention is minimizing shared state by statically dividing the state into disjoint partitions that can be processed concurrently. Chronos makes use of a similar architecture to partition application data into multiple partitions, each assigned to a hardware thread. Its custom user space classification function performs deep packet inspection to examine the application header in the packet and separate out requests as shown in Figure 4.

While it would be possible to add a new field to the application header, we choose instead to overload an existing field. In the case of Memcached, we rely on the *virtual bucket*, or *vBucket* field, which denotes a partition of keyspace. For the search application we use the search term itself, and for the OpenFlow controller we use the switch ID. We did not implement H-Store or the message broker in this work but they could be partitioned on the basis of partitionID and client queueID.

Discussion: Partitionable data assumption fits well for class of applications like key-value stores, search, Hstore, message broker and OpenFlow. Requests for data from multiple partitions is an active area of research, and one we hope to study in future work [25].

Concurrent access to the partitioned data is still protected by a mutex to ensure program correctness, however the in-NIC partitioning function ensures that there is a serialized

Algorithm 1 The in-NIC packet classification function, which distributes requests to application threads based on the partition ID request header field.

```
1: procedure INITIALIZER:
2:     epochMap ← randomPartitionToThread
3:     lastUpTime ← 0
4:     threadLoadMap ← 0
5:     threadEpochLoad ← 0
6: end procedure
7: procedure HANDLENEWPACKET(pkt)
8:     partitionID ← packet.getPartitionID()
9:     epochMap.updateRate(partitionID)
10:    if packet.arrivalTime − lastUpTime ≥ epoch
   then
11:        exec loadbalancer()
12:        lastUpTime ← pkt.stamp(); rate ← 0
13:    end if
14:    thread ← epochMap.Thread(partitionID)
15:    threadLoadMap(thread) + +
16:    totalEpochLoad + +
17:    return thread
18: end procedure
```

set of operations to a given partition. The only time that two application threads might try to access the same partition is during the small windows where the load balancing algorithm updates its mapping. This remapping can cause some requests to follow the new mapping, while other requests are still being processed under the previous mapping. We will show in the evaluation that this is a relatively rare event, and for reasonable update rates of the load balancer, would not affect the 99th percentile of latency.

Extensible in-NIC load balancing: We expect that our endhost should be able to handle large spikes of load, with multiple concurrent requests, while running the underlying the system at high levels of utilization.

Chronos uses a classification function based on the partition ID field, and uses a soft-state table to map the partition ID field to an application thread. To reduce lock contention, the mapping should ensure an exclusive partitioning of the field. The load balancing module periodically updates the table based on the offered load and key popularity.

We chose to implement a simple load balancing algorithm in Chronos. It divides time into epochs, where each epoch is of maximum configurable duration T. It maintains a mapping of each partition to an assigned NIC queue (and thus application thread), along with per-partition load information. The first field in the table, *mapPartitionToQueue*, indicates the partition ID to application thread mapping. The second maintains the number of accesses in the current epoch, a simple indicator of load and popularity. The load balancer also maintains a separate counter for each thread, which indicates the number of requests served by the application thread in the current epoch.

Algorithm 2 $Chronos$ Load Balancer updates partition field-thread mapping based on load offered in last epoch.

```
1: IdealLoad ← totalEpochLoad()/totalThreads
2: for all v ∈ partitionID do
3:     prevEpochThread ← epochMap.getThread(v)
4:     if ThreadLoad[prevEpochThread] ≤ idealLoad
   then
5:         currentEpochMap.assign(v,lastT)
6:         threadLoadMap[prevEpochThread] =
   partitionLoad
7:     else
8:         for all thread ∉ {{totalThreads} −
   prevEpochThread} do
9:             if threadLoadMap[k] ≤ IdealLoad then
10:                currentEpochMap.assign(v,k)
11:                threadLoadMap[lastT] ←
   partitionLoad
12:                break
13:            end if
14:        end for
15:    end if
16: end for
17: epochMap ← currentEpochMap
```

At application load time, the load balancer initializes the table with a random mapping of partition field to threads. Algorithms 1 and 2 show psuedocode for the $Chronos$ classifier and load-balancer modules, respectively. A new epoch is triggered when either the duration of the current epoch has elapsed, or the number of requests to the server to a particular thread exceeds a configurable threshold. At the start of a new epoch, the load balancer computes the new mapping based on previous epoch's load as described in algorithm 2.

First, it computes the total load in the last epoch and divides that by the number of threads to obtain Ideal Load (IL) each thread should serve in the next epoch, under the assumption that load distribution will remain the same. Second, the algorithm processes each partition, assigning it to a thread to which it was assigned in the last epoch. This assignment succeeds if the new thread load is less than the computed IL.

For the Chronos load balancer to work effectively, the number of partitions should be at least the number of cores available across all of the application instances. Chronos load balancing does not add to cache pollution that might happen due to sharing of partitions among threads. In fact, baseline application will have lower cache locality given that all of its threads access a centralized hash table.

Discussion:How does User space hash function work? In regards to processing application headers our particular NIC supports only 5 tuple processing in the hardware. We expect future generations of NICs to support more flexible header offset processing (and are already seeing examples of such). An option, userspace custom hash function enables deep packet inspection and arbitrary processing over the packet contents. This function works by registering a C function with the NIC API, and then when a new packet arrives, the NIC will call the function, passing it a pointer to the packet and the packet length. The execution of that C function will occur on one of the CPU cores. The function you provide can do arbitrary processing over any or all of the packet contents. The deeper the processing that you do on the packet, the more of the packet contents have to be copied to DRAM, and so there is an overhead here. In our case, the performance penalty due to user space processing was outweighed by the latency incurred in the kernel.

4.3 Application Implementations

Memcached: Rather than building a new key-value store, we base Chronos on the original Memcached codebase. Chronos is a drop-in implementation of Memcached that modifies only 48 lines of the original Memcached code base, and adds 350 lines. We replace the centralized hash table and slab class table of Memcached which were protected by mutex locks with N such tables and lists, one per partition. These modifications include support for user-level network APIs, for the in-NIC load balancer, and for adding support for finer-grained application locks.

Web Search: Web search query evaluation is composed of two components. The first looks up the query term in an inverted-index server to retrieve the list of documents matching that term. The second retrieves the documents from the document server. The inverted index is partitioned across thousands of servers based on either document identifier or word/term. For Chronos, we choose to implement term based partitioning so that we could evaluate the load balancer. We wrote our own implementation of web search based on Memcached.

It is important that web search index tables are kept updated, and so modifications to them are periodically necessary. One approach is to create a completely new copy of the in-memory index and to then atomically flip to the new version. This would impose a factor of two memory overhead. Another option is to update portions of the index in place, which requires sufficient locking to protect the datastructures. We implemented an index server using read/write locks and UNet API's. The index server maintains the index-table as search term and associated documents IDs, as well as word frequency and other related information. We also implemented a version of index server with RCU mechanism from an open-source codebase provided by the RCU authors [47]. We modified the codebase to work with UNet API's. Chronos further divides the index server table into several partition based on terms.

OpenFlow Controller: We also choose to implement Chronos with an OpenFlow controller application provided by [36]. This application is different from the Memcached and web search applications, since it is typically not horizontally scaled in the same way as these other applications. However, given

that the OpenFlow controller can be on the critical path for new flows to be admitted into the network, its performance is critical, even if the entire application is only deployed on a single server.

5. EVALUATION

In this section we evaluate Chronos using both micro and macro-benchmarks. We begin by describing Chronos's implementation, the workloads we use, and performance metrics we measure. We first establish the kernel overhead of Memcached. We make comparisons against an optimized MOSBENCH pk kernel [33] which removes lock contention and other scalability bottlenecks from the kernel. Next we describe latency profile for both baseline Memcached and Chronos under uniform demand and uniform key access patterns. Then, we evaluate against more realistic workloads by introducing skew into the inter-request arrival times, as well as skew in the key access patterns. We then evaluate Chronos's load balancing functionality to measure how quickly it can respond to hotspots in client requests. Lastly, we evaluate web-search application and OpenFlow Controller. For web-search application we evaluated against a user level RCU implementation of Memcached open-source by authors of [47].

Implementation: We deployed Chronos on 50 HP DL-380G6 servers, each with two Intel E5520 four-core CPUs (2.26GHz) running Debian Linux with kernel version 2.6.28. Each machine has 24 GB of DRAM (1066 MHz) divided into two banks of 12 GB each. All of our servers are plugged into a single Cisco Nexus 5596UP 96-port 10 Gbps switch running NX-OS 5.0(3)N1(1a). This switch configuration approximates the ideal condition of nonblocking bandwidth on a single switch. We do not focus on network sources of latency variability in this evaluation. Each server is equipped with a Myricom 10 Gbps 10G-PCIE2-8B2-2S+E dual-port NIC connected to a PCI-Express Gen 2 bus. Each NIC is connected to the switch with a 10 Gbps copper direct-attach cable.

When testing against kernel sockets, we use the myri10ge network driver version 1.4.3-1.378 with interrupt coalescing turned off. For user-level, kernel-bypass experiments we use the Sniffer10G driver and firmware version 2.0 beta. We run Memcached version 1.6 beta, configured to support the binary protocol and virtual buckets.

Metrics and Workloads: Like any complex system, the performance observed from Memcached and Chronos is heavily dependent on the workload, which we define using the following metrics: 1) request rate, 2) request concurrency, 3) key distribution, and 4) number of clients. The metrics of performance we study for both systems are 1) number of requests per second served, 2) mean latency distribution, and 3) 99th percentile latency distributions.

To evaluate baseline Memcached and Chronos under realistic conditions, we use two load generators. The first, Memslap [31], is a closed loop benchmark tool distributed with

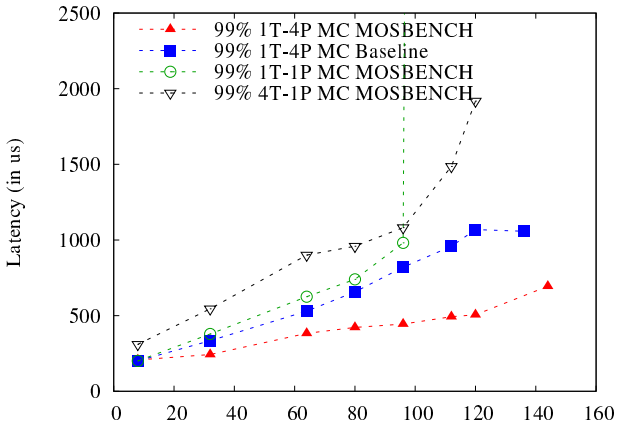


Figure 5: **Legend: NT-MP stands for N thread M processes of Memcached(MC). Shown is the tail latency for one and four threads (1T and 4T) running in either one process or four processes (1P or 4P). Even with the in-kernel improvements of Mosbench, latency still grows.**

Memcached that uses the standard Linux network stack. It generates a series of *get()* and *put()* operations using randomly generated data. We configure it to issue 90% get and 10% put operations for 64-byte keys and 1024-byte values. For the results that follow, we found that varying the key size had a minimal effect on the relative performance between Chronos and baseline Memcached. The second load generator is an open-loop load program we built in-house using low-latency, user-level network APIs to reduce measurement variability. Each instance of this second load generator issues requests at a configurable rate, up to 10 Gbps per instance, with either uniform or exponential inter-arrival times. The KV-pair distribution used by the tool is patterned on YCSB [17]. Note that the latency numbers reported in figures generated by closed loop generator are off by 50-70 μs since they must traverse they kernel and network stack.

5.1 Latency

We now examine the latency of a single threaded instance of Memcached using default kernel installed on system as well as an optimized MOSBENCH pk kernel. We instantiated 4 instances of single threaded Memcached, each on its own core. To measure its performance we use a configurable number of *Memslap* clients, each deployed on its own core to lower the measurement variability. A client would open socket connection to one of the 4 Memcached process. While running in single threaded mode and thus free of intra-thread resource contention, we expect single threaded, multiple process Memcached latency and variance to be lower than multi-threaded instance on MOSBENCH. Figure 5 shows our results. Even running Memcached with MOSBENCH kernel 99% tile latency is still high 810 μs with 140 clients divided across 4 processes , indicating that kernel contribution is significant even when with optimized

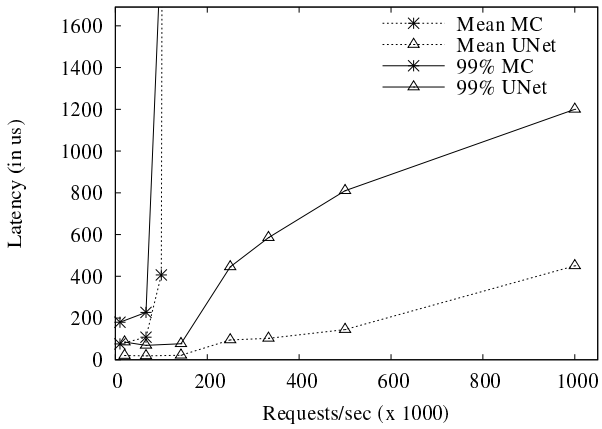


Figure 6: **Latency of baseline Memcached (MC), Memcached with user-level network APIs (UNet locks), and Chronos (CH).**

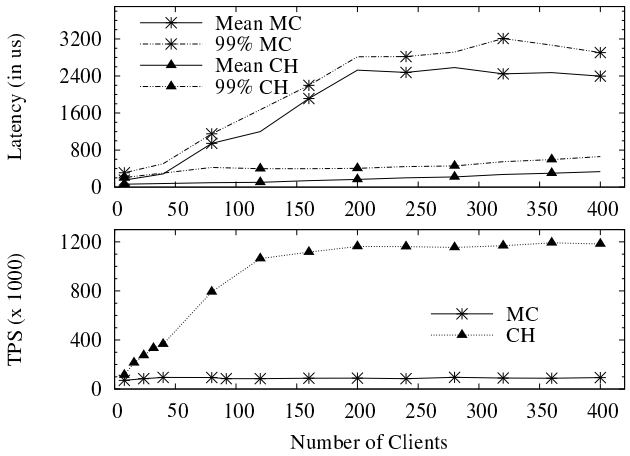


Figure 7: **Latency as a function of the number of clients with the Memslap benchmark (closed loop).**

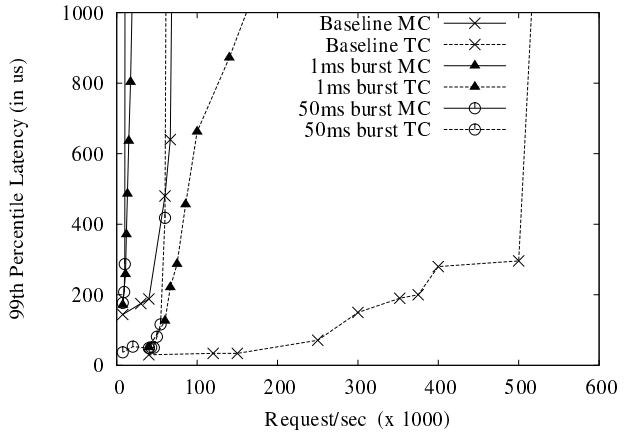


Figure 8: **The effect of skewed request inter-arrival times on tail latency.**

version of kernel and no application lock contention.

Next, we examine the latency distribution of baseline Memcached and Chronos under ideal conditions of a uniform key access pattern and uniform inter-request arrival time. To evaluate the effect that the user-level network APIs played on observed latency, we created a version of Memcached with user-level APIs (and no other Chronos features). All instances were setup with 4 application threads each. We instantiated 10 client machines running our custom open-loop load generator utilizing user-level network APIs. Each client issues requests at a configurable rate, measuring the response time as perceived by the client as well as any lost responses. The server is pre-installed with 4GB of random data, and clients issue requests from this set of keys using a uniform distribution with uniform inter-request times. We use 1K bytes values and 64 byte keys in a 9:1 ratio of gets to sets. Each client terminates when the observed request drop rate exceeds 1%.

The results are shown in Figure 6. For comparison, baseline Memcached supports up to approximately 120,000 requests per second before dropping a significant number of requests. Chronos supports a mean latency of about 25 μs up through 500,000 requests per second and rises just above $50\mu s$ at 1M requests per second. The version of Memcached with just the socket API replaced with the user-level kernel API not only has higher mean latency, but the variation of latency is significantly higher, as shown by the 99th percentile, indicating that reducing variability in the network stack, operating system, and application are all important to reduce tail latency.

We compare the single-threaded latency of both baseline Memcached and Chronos, as shown in Figure 8. Here we see that a single-threaded instance of Chronos can support up to 500,000 request/sec, with tail-latency contained up through 200,000 requests per second.

We next evaluate the performance of Chronos with a larger number of clients. Shown in Figure 7, we instantiated eight client Memslap processes on each physical client machine, and scaled up to 50 client machines. We see that Chronos supports over 1 million transactions per second (TPS), limited only by the NIC's throughput limit of 10 Gbps. At this point, the request throughput levels out, causing a small amount of additional client latency as requests wait to be transmitted. In contrast, baseline Memcached suffers from low throughput and high latency at these rates.

5.2 Skew In Request Inter-Arrival Times

In this section, we study how skew in the inter-arrival time of requests affects both baseline Memcached as well as Chronos. In the next section, we will study how skew in the key access pattern affects latency.

The presence of skewed request inter-arrival times means that although the average request load might be manageable, at very fine-grained timescales there are short periods of overload. Depending on how skewed the request pattern is, there might be several back-to-back requests followed by a gap in requests. From the point of view of the server, that

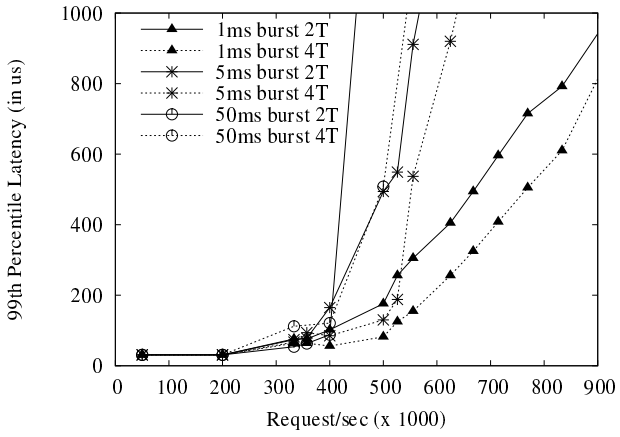


Figure 9: **The latency of two thread (2T) and four thread (4T) implementations of Chronos under skewed request arrivals**

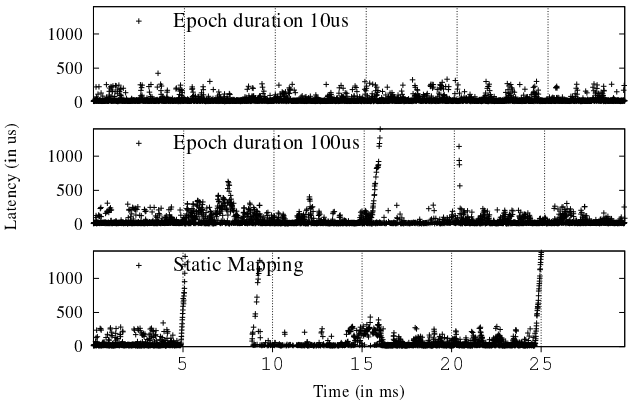


Figure 10: **An evaluation of the responsiveness of the Chronos load balancer module across two time epochs (100 μs on top and 10 μs in the middle) and the static mapping strategy (on the bottom).**

workload induces a momentary state of overload, which results in application-layer queuing.

To study this behavior, we use the methodology described by Banga and Druschel [7], originally presented in the context of web server evaluation. Here, multiple clients generate traffic at a fixed rate, punctuated with short bursty periods. These periods are characterized by two parameters: 1) the ratio of the maximum request rate and the average request rate, and 2) the duration of bursts. We fix the maximum-to-average request ratio to be 10, and limit the burst duration to be 10% of the number of requests sent. Lastly, we ensure that the number of requests in a burst are fixed across the experiments.

Figure 8 shows the 99th percentile of latency for both baseline Memcached as well as Chronos across a range of burst periods. We see that in the baseline even short burst duration's of 1 millisecond impose significant levels of application queuing at 10,000 requests per second, driving latency up to over a millisecond. Note that without request inter-arrival time skew, baseline Memcached supports up to a factor of 10 larger request rate.

For Chronos under a uniform request inter-arrival rate, latency stays largely flat up through 325,000 requests per second. However, just as in the baseline case, inducing request bursts drives up latency significantly while reducing the throughput of the system. For 1 millisecond bursts, the request rate with controlled latency is reduced to 100,000 requests per second, with an observed latency of up to 1 millisecond at over 250,000 requests per second. For longer burst duration's, this effect is more pronounced.

Figure 9 considers request loads up to 1M requests per second forwarded to $Chronos$ instances with either two or four application threads, each running on its own CPU core. As in the single-thread case, bursts in request rates arriving faster than the effective service time of the application induce application queuing, and thus increases in delay. This effect is more pronounced at higher loads, given that there is less time between arriving requests. Adding additional cores mitigates the effect of bursts, but for sufficient burst lengths queuing will build up with any fixed number of CPU cores.

5.3 Load balancing

The application queuing behavior described in the previous section occurs in any system that handles requests at a fixed rate, assuming that enough skew is present to induce short bursts of overload. To mitigate this queuing effect, Chronos employs a load balancing module that periodically reapportions requests across application threads within a single server. The time elapsed between the re-mapping of keys denotes an *epoch*. This provides an upper bound for how much skew will be present within a single Chronos node. As described in Section 4.2, the load balancer works in concert with the NIC-level hash function to ensure that requests are sent to application threads in such a way to minimize or eliminate lock contention. Thus, we require that the load balancer assign keys across application threads such that each thread sees a strict partition of vBuckets.

To evaluate the responsiveness of the Chronos to key access skew, we created an experiment as follows. First, we setup a Chronos instance with four threads. We then configured the load balancing module with an epoch time of 10 μs and 100 μs. A single open-loop client sends requests at a rate of 1 million requests/sec. Keys are chosen at random at the start of each epoch such that three keys receive 99% of the requests. This pattern is motivated by the desire to have three of the four cores handling the hot/popular keys, and the remaining core to receive all of the cold/unpopular keys. We know by construction that without an adaptive load balancing module, each time the epoch changes overload would occur since two or more popular keys would be handled by a single application thread, and the rate of requests is sufficiently high to induce overload in that case. We repeat the

Component	Switches	Mean latency (μs)	99 %ile latency (μs)
Kernel Based	1	65	140
Kernel Based	16	120	250
UNet API Based	1	8	50
UNet API Based	16	10	51

Table 2: **Latency of the OpenFlow Controller. Because of its single-threaded nature, we are not able to evaluate in-NIC partitioning or load balancing. However, removing the kernel overhead reduces the tail latency significantly.**

same experiment for a Chronos instance with static mapping of keys to threads.

Figure 10 shows the latency distribution for Chronos at $10\mu s$ (top), $100\mu s$ (middle), and for the static mapping (bottom). At the start of each epoch, we see occasional long spikes in the 100 μs case before it is able to adapt to shifts in workload. The static mapping approach fails when two or more popular keys are served from the same application since these types of co-located request hotspots cannot be migrated to other cores.

Discussion: Due to our reliance on partition to spread load across cores, there are certain cases that will cause Chronos's load balancing element to fail or perform poorly. When a single key in a partition, or the partition itself becomes hugely popular, the rate of requests to that partition can overwhelm a single application thread. This happens when the request load approaches 500,000 requests/sec (which is greater than 5 Gbps of traffic). When a single key becomes that popular, we are limited in our response, and would suggest that the application itself be re-architected, since such a high get/set load on a single key would not be practical at scale. However, it is more likely that several keys in the same partition might together induce such a high load. We can alleviate this condition by moving those common keys to separate vBuckets, or by modifying the request handling logic in Chronos to allow the server to split and join buckets based on load demands. We have not yet evaluated these possible features.

5.4 Web Search

As described in Section 4.3, the web search application maintains a hash table to store the term and associated document, protected by read/write locks. In Chronos, we further divide this index into twelve partitions based on the term, and store them in separate tables protected by a mutex. We evaluate Chronos in comparison to an RCU lock-based implementation of the hash table that was provided by Triplett et al [47]. Additionally, we modified this implementation to work with the same user-level networking API used in Chronos to provide a direct comparison. For search we used 10 Byte key and 1400 Byte value as inverted index list. The results of this evaluation are shown in Figure 11. Here, we see that even with an implementation based on read/write locks and RCU, we see higher latency.

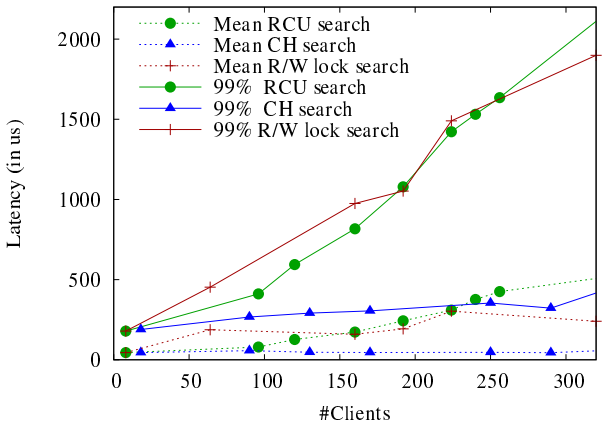


Figure 11: **Web search latency of single Index server.**

5.5 OpenFlow Controller

We now evaluate Chronos as implemented with a TCP-based OpenFlow controller. Ensuring low latency in OpenFlow controllers is critical since they can be on the critical path to admitting new flows into the network.

In our experimental setup, we replaced the stock kernel TCP socket network implementation with a user-level TCP implementation provided by our NIC vendor. The controller software itself is single-threaded. We then compare this user-level TCP implementation with the baseline controller. To generate load, we used the *cbench* benchmark included with OpenFlow. Cbench emulates switches that send *packet-in* messages to the controller, and wait for flow modification rules in return. The controller implements a learning switch application, which generates appropriate rule events in response to packet-in events. In our experiment, we simulated 16 switches supporting 1M MAC entries. Each emulated switch connects to the controller and generate packet-in events. To measure just the controller latency, we installed packet mirroring rule described in section 3.2. The results of this experiment are shown in Table 2. We see that removing the kernel has the predictable effect of removing average latency. However, the effect on the 99th percentile of latency is that the difference between one emulated switch and sixteen emulated switches is only a single microsecond, as compared to 110 microseconds in the baseline case.

The TCP implementation provided by our vendor is not compatible with multiple NIC queues, and so we cannot evaluate it in the context of a multi-threaded deployment of OpenFlow controller, or with the request load balancing module. However, this micro-benchmark indicates that at least under these workloads, the performance of a user-level TCP implementation was similar to that of user-level UDP.

6. CONCLUSIONS

The scale of modern datacenters enables developers to deploy applications across thousands of servers. However that same scale imposes high monetary, energy, and management costs, placing increased importance on efficiency. To meet strict SLA demands, developers typically run services at low utilization to rein in latency outliers, which decreases efficiency. In this work, we present Chronos, an architecture to reduce data center application latency especially at the tail. Chronos removes significant sources of application latency by removing the kernel and network stack from the critical path of communication, by partitioning requests based on application-level packet header fields in the NIC itself, and by load balancing requests across application instances via an in-NIC load balancing module. Through an evaluation of Chronos as implemented in Memcached, OpenFlow, and a web search application, we show that we can reduce latency by up to a factor of twenty, while significantly reining in latency outliers. The result is a system that can enable more throughput by increasing predictability, a key contribution to improving datacenter efficiency.

Acknowledgements

This work was supported by the National Science Foundation (CNS-1116079). Thanks to Abhijeet Bharkar and Mohammad Naghshuar.

7. REFERENCES

[1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *Proc of High Performance Computing, Networking, Storage and Analysis*, 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of SIGCOMM 2008*, August 2008.

[3] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *Proc. of Communication, Control, and Computing*, 2008.

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

[5] A. Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, 1978.

[6] Low-Latency Lossless Fabric. `http://www.aristanetworks.com/en/products/7500series/technology#losslessfabric`.

[7] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proc. of 1st USENIX USITS*, 1997.

[8] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-Core Key-Value Store. In *Proc. of 2nd IGCC*, 2011.

[9] BerkeleyDB. `http://www.oracle.com/technology/products/berkeley-db/index.html/`.

[10] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 1995.

[11] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proc. of 8th USENIX OSDI*, 2008.

[12] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proc. of 9th USENIX OSDI*, 2010.

[13] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proc. of 7th WWW Conference*, 1998 April.

[14] P. Buonadonna, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proc. of 11th IEEE/ACM SC*, 1998.

[15] B. Cantrill and J. Bonwick. Real-world concurrency. *Commun. ACM*, 2008.

[16] Cut-Through and Store-and-Forward Ethernet Switching for Low-Latency Environments. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-465436.html`.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of 1st ACM SOCC*, 2010.

[18] Z. David, D. Tathagata, P. Mohan, and R. H. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. Number UCB/EECS-2011-113, 2011.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of 21st ACM SOSP*, 2007.

[20] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proc. of 3rd USENIX USITS*, 2001.

[21] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. of SOSP*, 1997.

[22] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of SIGCOMM*, 2009.

[23] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *Proc. of SIGCOMM*, 2008.

[24] Infiniband. http://www.infinibandta.org/.

[25] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proc. of SIGMOD*, 2010.

[26] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proc. of OSDI*, 2010.

[27] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni. Architectural Breakdown of End-to-End Latency in a TCP/IP Network. *International Journal of Parallel Programming*, 2009.

[28] LevelDB: A Fast and Lightweight Key/Value Database Library. http://code.google.com/p/leveldb/.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *Proc of SIGCOMM*, 2008.

[30] Memcached. http://memcached.org/.

[31] Memslap. http://docs.libmemcached.org/memslap.html.

[32] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHash: A Cache-Partitioned Hash Table. Number TR2011051, 2011.

[33] MOSBENCH Source Code. http://pdos.csail.mit.edu/mosbench/.

[34] Mutrace. http://git.0pointer.de/?p=mutrace.git.

[35] Myricom Sniffer. http://www.myricom.com/sniffer.html.

[36] Openflowcontroller source code. http://www.openflow.org/wp/downloads/.

[37] PF_RING Direct NIC Access. http://www.ntop.org/products/pf_ring/dna/.

[38] Rdma:remote direct memory access. http://www.rdmaconsortium.org/.

[39] Redis. http://redis.io/.

[40] Y. Ruan and V. S. Pai. The Origins of Network Server Latency & the Myth of Connection Scheduling. In *Proc. of Joint International Conference on Measurement and Modeling of Computer Systems*, 2004.

[41] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proc. of HotOS*, 2011.

[42] P. Saab. Scaling Memcached at Facebook. http://facebook.com/note.php?note_id=39391378919, 2008.

[43] SMC SMC10GPCIe-10BT Network Adapter. http://www.smc.com/files/AY/DS_SMC10GPCIe-10BT.pdf.

[44] SolarFlare Solarstorm Network Adapters. http://www.solarflare.com/Enterprise-10GbE-Adapters.

[45] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proc. of the EuroSys*, 2007.

[46] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proc. of 15 ACM PPoPP'10*, 2010.

[47] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proc. of USENIX ATC*, 2011.

[48] VoltDB. http://voltdb.com/.

[49] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of 15th ACM SOSP*, 1995.

[50] M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *Proc. of 3rd IEEE HPCA*, 1997.

[51] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *Proc. of SIGCOMM*, 2010.

[52] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors Via Scheduling. In *Proc. of 15th ACM ASPLOS*, March 2010.