Chronos: Predictable Low Latency for Data Center Applications

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

ABSTRACT

In data center applications, predictability in service time and controlled latency, especially tail latency, are essential for building performant applications. This is especially true for applications or services built 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 three data center applications, Memcached, OpenFlow, and Web search, to measure the effect of 1) kernel socket handling, NIC interaction, and the network stack, 2) application locks contested in the kernel, and 3) application-layer queueing due to requests being stalled behind straggler threads on tail latency. We propose Chronos, a framework to deliver predictable, low latency in data center applications. Chronos uses a combination of existing and new techniques to achieve this end, for example by supporting Memcached at 200,000 requests per second per server at mean latency of 10 μs with a 99^{th} percentile latency of only 30 μs, a factor of 20 lower than baseline Memcached.

Categories and Subject Descriptors

D.4.4 [**Communications Management**]: Network communication

General Terms

Algorithms, Design, Performance

Keywords

Cloud Computing, Predictable Latency, User-level Networking, Load Balancing

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 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 99^{th} 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 [32].

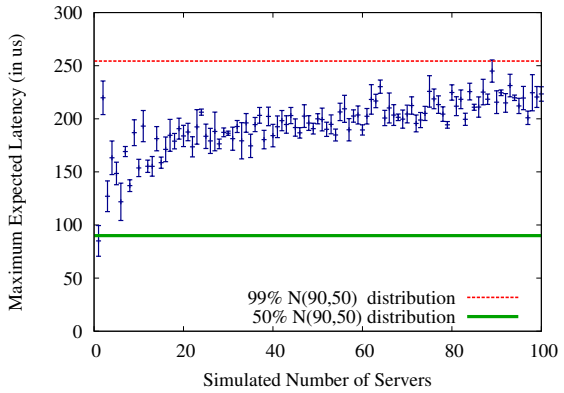
Within the data center, end-to-end application latency is the sum of a number of components, including interconnect fabric latency, the endhost kernel and network stack, and the application itself. The interconnect fabric is not likely to be a significant source of latency unless it is heavily congested, since these networks are designed to deliver both high bandwidth and low latency to better support scale-out applications [2, 3], and ongoing efforts aim to minimize congestion, and thus latency [4, 5, 40]. On the other hand, the latency of applications is decreasing as well, due to the interwoven trends of increased cores per server, increased DRAM capacity, and the availability of low-latency, flash-based SSDs.

What remains in large part is kernel latency, which includes interrupt handling, buffer management, the network stack, data copying and protection domain crossing. Despite recent improvements in kernel performance [14], kernel overheads can be an order of magnitude larger than the data center network fabric and application latency combined. In this paper, we examine the latency overheads of several common data center applications—Memcached [26], Web search, and an OpenFlow [25] controller—and find that 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.

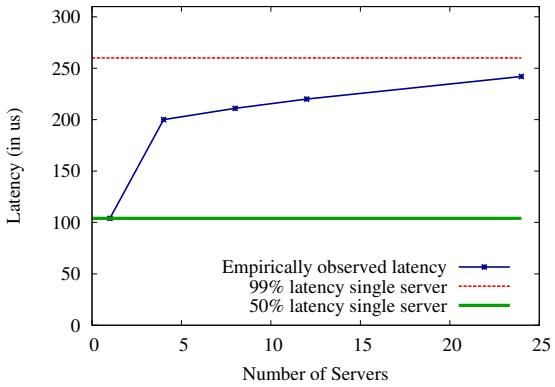
To eliminate this kernel and network stack overhead, we leverage user-level, kernel bypass, zero-copy network functionality. These APIs are known to minimize latency by eliminating the kernel from the critical message path, and thus avoiding overheads due to multiple copies and protection domain crossings [11, 16, 30, 39]. An interesting aspect of data centers that we focus on in this work is its very high link speeds, often 10 Gbps, and the need to support a dozen or more cores per machine. One key barrier to the adoption of user-level networking APIs has been supporting legacy applications. However, an advantage in data centers is that this barrier is much lower than before, since operators control the entire stack, from hardware to the operating system to the application. The result is that we can eliminate a major source of latency in the end-to-end path with minimal, and in some cases no, change to the application.

While user-level networking removes kernel overhead, it is not enough to fully realize low-latency applications. Removing the

SOCC'12, October 14-17, 2012, San Jose, CA USA



(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.

kernel from the network path leaves the application responsible for handling hotspots in load, and if application threads contend for locks, that contention still occurs in the kernel. To address these challenges, we propose Chronos, a communication framework that leverages both kernel bypass and NIC-level request dispatch to deliver predictable low latency for data center applications. Chronos directs incoming requests to concurrent application threads in a way that drastically reduces, and in some cases eliminates, application lock contention. Chronos also provides an extensible load balancer that spreads incoming requests across available processing cores to handle skewed request patterns while still delivering low-latency response time.

Our evaluation shows that Chronos substantially improves data center application throughput and latency. We show that Memcached implemented on Chronos, can support 200,000 requests per second with a mean operation latency of $10\mu s$ with a 99^{th} 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.

Contributions: In summary, Chronos makes the following contributions: We (1) analyze impact of tail latency on data center traffic patterns; (2) analyze sources of latency and latency variation, exposing application bottlenecks with user-level networking APIs; (3) design a framework using user-level networking APIs that leverages NIC-support to reduce lock contention and perform efficient load balancing to reduce the tail latency in data center networks; and (4) evaluate the resulting performance of three representative applications on a testbed with 50 servers.

2. BACKGROUND AND MOTIVATION

In this section we discuss the effect of latency and high latency variation on two data center workload patterns — (1) Partition/Aggregate, (2) Dependent/Sequential traffic pattern and how high latency variation impacts the end-to-end performance and operation of data center applications. We use Memcached as an example of each of these communication patterns. Memcached is a popular, in-memory Key-Value (KV) store, deployed at Facebook, Zynga, and Twitter [10, 26]. Its simple API consists of operations that get, set, delete, and manipulate KV pairs. For high throughput, Memchached requests are typically issued using UDP [33].

In this section, we seek to show that the end-to-end latency for the Partition/Aggregate communication pattern is driven by the tail-latency at scale. In the case of the Dependent/Sequential pattern, the tail latency determines the number of service invocations allowed within the SLO. Thus, it is important to reduce the variance in service latency in addition to bringing down overall latency.

2.1 The Partition/Aggregate Pattern

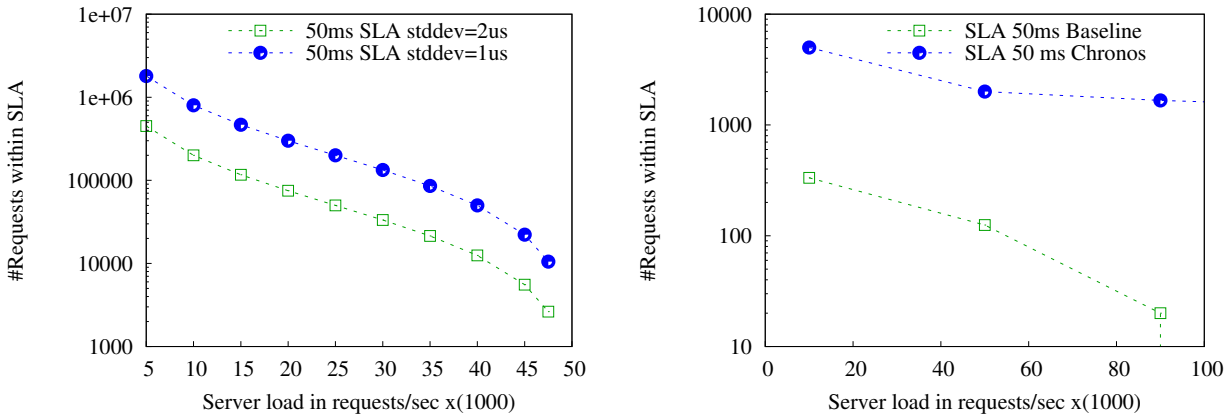
In the Partition/Aggregate communication pattern, data is retrieved from a large number of servers in parallel prior to being combined into a response for the requesting service. An example of this pattern is a horizontally scaled Web search query that must access state from hundreds to thousands of inverted indices to generate the final response. 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 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. Here, the key insight is that increasing the number of servers increases the probability of hitting the tail latency more often, and hence increases the overall latency seen by the end user. We now show this straggler behavior both theoretically and experimentally.

Analysis: We first 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 distributed (i.i.d.) 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 $\mu = 90\mu s$ and $\sigma = 50\mu s$ (these values are based on our observations of Memcached's latency, described in Section 3). We estimate service time by computing values of sets of i random variables, where i ranges from 1 to 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. Figure 1(a) shows the result.

As the number of servers increases, the maximum observed value in $\vec{v}$ increases as well. We also plot the 50^{th} and 99^{th} percentiles of the underlying $N(90, 50)$ distribution. In this simulation, when the number of servers is small, the maximum expected latency is close to the mean of 100 μs (the 50^{th} percentile of the random variable). However, as S grows the maximum observed value approaches the 99^{th} percentile value of $254.25\mu s$. In this way, the end-to-end latency of the Partition/Aggregate communication pattern is driven by the tail-latency of nodes at scale.



(a) Predicted by queueing 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.

Experimental validation: To validate the above probabilistic analysis, we perform the following experiment on our testbed. We set up six Memcached clients, each on different machines, and measured the latency seen by one of these clients. Each client issues S parallel *get* requests to a set of S server instances (where S ranges from 1 to 24). Clients waits for response from all the servers before generating next set of requests. Each server instance runs on its own machine. In addition, 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.

Figure 1(b) shows the results of experiments and observed single-server median latency (approximately $100\mu s$) and the 99^{th} percentile of latency (approximately $255\mu s$). As expected, when issuing a single request to a single server the observed latency is nearly the 50^{th} 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 99^{th} percentile.

2.2 The Dependent/Sequential Pattern

A second network communication pattern in data centers is the dependent/sequential workflow pattern, where applications issue requests one after another such that a subsequent request is dependent on the results of previous requests. Dependent/sequential patterns, for example, force Facebook to limit the number of requests that can be issued to build a user's page to between 100 and 150 [32]. The reason for this limit is to control latency, since a large number of sequential requests can add up to a large aggregate latency. With a large number of sequential requests the number of requests hitting the tail latency will also increase, thus lowering the number of otherwise possible sequential invocations. Another example of this pattern is 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 subsection we show how it can be validated both through a queueing analysis as well as a simple microbenchmark.

Consider a simple model of a single-threaded server where 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 [7], 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 $\rho = \lambda/\mu$.

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 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 predicted number of permitted service invocations to the actual number as measured with Memcached deployment in our testbed, shown in Figure 2(b). The experimental setup and experiments are described in detail in Section 5.2. Here, we measure the 99^{th} percentile of latency for both baseline Memcached as well as Memcached implemented on Chronos (CH) with uniform inter-arrival time and access pattern for requests. Each point in figure represents the number of service invocations permitted with the specified SLA, as a function of the server load, in requests per second.

The overall trends in these simple studies confirm the intuition that delivering predictable, low latency response requires not just a low mean latency , but also a small variation from the mean.

3. LATENCY CHARACTERIZATION

In this section, we give a detailed analysis of the main components contributing to the end-to-end latency in the data center applications. We summarize the results in Table 1 and report the contribution of each component in the end-to-end latency. This includes one-way network latency for a request to reach from the client to the server, the latency at endhost server to deliver the request to the application and application latency for processing

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: Latency sources in data center applications. The underlying operating system is Linux 2.6.28. †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.

the request and sending the out the response from the server. As a concrete example, we further analyze the impact of server load and lock contentions due to concurrent requests on the Memcached server latency.

3.1 Sources of End-to-End Application Latency

Data center Fabric: The data center fabric latency is the amount of time it takes a packet to traverse the network between the client and the server. This latency can be further decomposed into propagation delay and in-switch delay. Within a data center, speed of light propagation delay is approximately 1 μs. Within each switch, the switching delay is approximately 1–4 μs. Low-latency, cut-through switches further reduce this packet forwarding latency to below one microsecond. A packet from client to server typically traverses 5–6 switches [3]. A packet can also suffer queueing delay based on prevailing network congestion. We calculate the queueing delay by measuring the additional time a packet waits in switch buffers. Typical commodity silicon might have between 1–10MB buffers today for 10Gbps 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 7 aim to minimize congestion and thus latency. We expect that, in the common case, the networks paths will be largely uncongested. While in network bottlenecks such as delay in data center fabric are outside the scope of this effort, the value of Chronos is that it addresses the key latency bottlenecks in the endhost to deliver low-latency services.

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

To understand the constituent sources of endhost latency under load, we profile a typical Memcached request. We issued 20,000 requests/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 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 10Gbps packet capture (at approx. 20ns resolution). Section 5 presents full details on the testbed setup.

A median request took 82μs 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 10Gbps. *DMA latency* is the transfer time of a 1600B (request and response) calculated assuming a DMA engine running at 5GHz.

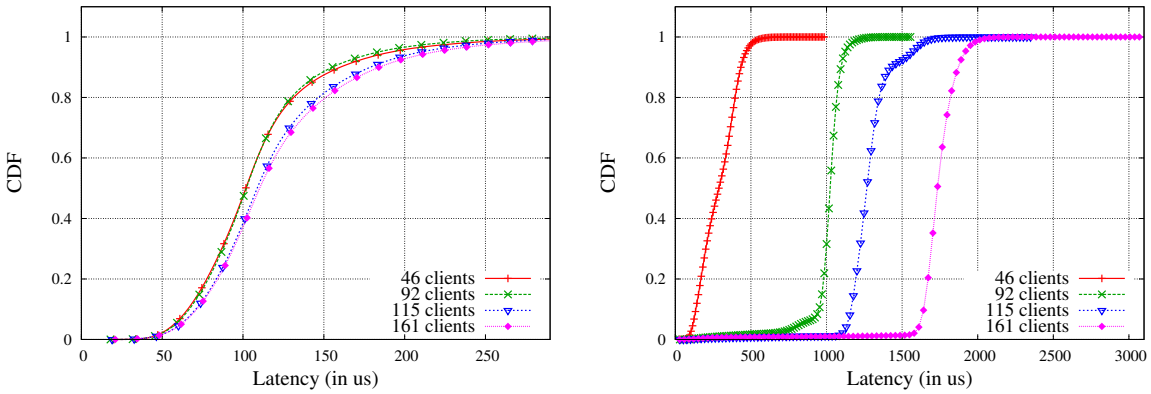
Application: This is the time required to process a message or request, perform the application logic, and generate a response. In the case of Memcached, this includes the time to parse the request, look up a key in the hash table, determine the location of value in memory and generate a response for the client. We 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. The application latency in Memcached is 2μs. In Section 3.2 we discuss other factors that contribute to application latency, including application thread 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 overhead 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 endhost application performance, aiming to reduce this important and significant component of latency.

3.2 End-to-End Latency in Memcached

In this section we further analyze Memcached latency. We show how increasing the load at the server results in queueing of pending requests in the kernel which significantly increases the tail latency. We further show that lock contention for processing concurrent requests also results in significant latency variation.

Effect of server load: To measure Memcached performance, we use a configurable number of Memslap clients [1], which are closed-loop (i.e., each client sends a new request only after receiving the response from the previous request) load generators included with the Memcached distribution to send requests to a Memcached server with four threads. Each client is deployed on its own core to lower measurement variability. We observe that Memcached can support up to 120,000 requests/second with sub millisecond tail latency. We next subject the Memcached server to a fixed request load, and observe the distribution of latency.



(a) Memcached latency distribution at 30% (low) utilization, (b) Memcached latency distribution at 70% (high) utilization

Figure 3: Latency of a Memcached server at a fixed load with varying numbers of closed-loop clients.

We evaluated the server at a low request load of 40,000 requests per second, which is approximately 30% of the server's maximum throughput, and also at a high load of 90,000 requests per second, or about 70% 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 server utilization (30%), increasing the number of clients had little effect on distribution of latency as shown in Figure 3(a). By increasing the number of clients we increase the number of concurrent requests at the server, even though load offered by each client drops. Most responses completed in under 150 μs, with the tail continuing up to approximately 300 μs, shown in Figure 3(a). This corresponds to lower levels of load at which developers run their services to ensure low tail-latency. However, at high server utilization (70%), increasing the number of clients had a pronounced effect on observed latency. High load resulted in a significant latency increase as the number of clients increased, reaching a maximum at about 2,000μs, shown in Figure 3(b). These measurements aid our understanding of current practices of running services at low levels of utilization. Operating these services at higher utilization necessitates reining in the latency outliers.

Request queueing in the application plays a significant role in the latency increase. Two sources of this queueing are variance in kernel service time and an increase in lock contention within the application due to an increase in concurrent requests. Profiling CPU cycles spent during the experiment shows that the bulk of the time is spent copying data and context switching.

Lock contention: We used the mutrace [27] tool during runtime to validate this last point and saw a significant amount of lock contention. We evaluated a Memcached instance with concurrent requests from 20 memslap clients and found that more than 50% of lock requests were contested, with that contested time accounting for about a third of the overall experiment duration. We found that the source of this lock contention in Memcached was a shared hash table protected by a pthread lock. This lock must be acquired both for update as well as lookup operations on the table. With pthread locks (used by Memcached), contention not only induces serialization, but must also be resolved in the kernel, adding further delay and variance to observed latency.

To quantify lock overhead, we modified a Memchached based Web search application to use two different synchronization primitives, (1) read/write locks and (2) an RCU (read copy update) mechanism [37] in place of the conventional pthread system locks

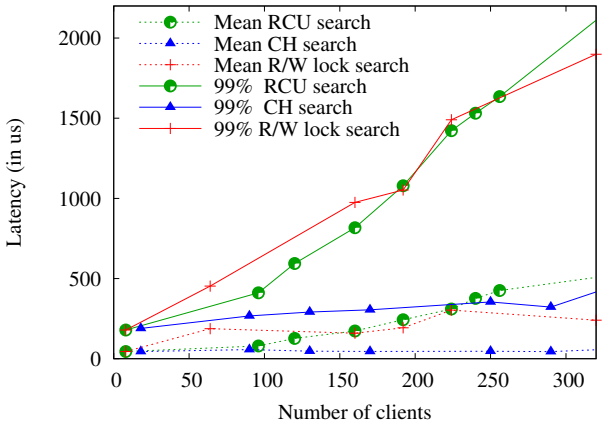


Figure 4: Web search latency of single Index server.

in Memcached. These synchronization primitives are more efficient for the read-dominant workloads that are common in applications like key-value stores (where the number of get requests is much larger than set requests) and search (where index-update is less frequent than index-lookup).

In addition to using the new locking primitives, we also modify the applications to use user-level networking APIs to bypass the kernel and eliminate kernel overheads in latency. We describe the user-level APIs in more detail in Section 4, but for illustration we can assume that use of these APIs removes kernel overhead completely. Bypassing the kernel with user-level APIs allows us to quantify the overhead caused due to application lock contention alone. For evaluation, we vary the number of memslap clients that send requests to the modified Memcached instances. We used 10-byte keys and 1400-byte values with a get/set requests ratio equal to 9:1 as suggested in [19]. Figure 4 shows the results of this experiment (the 99% and Mean CH search curves correspond to the Chronos results and can be ignored for now). Here, we see that even with an implementation based on read/write locks and RCU, latency remains high. Read/write locks do not scale because the state associated with them (number of readers) still needs to be updated atomically. For a small number of concurrent clients RCU performs well but as load increases there is significant variation

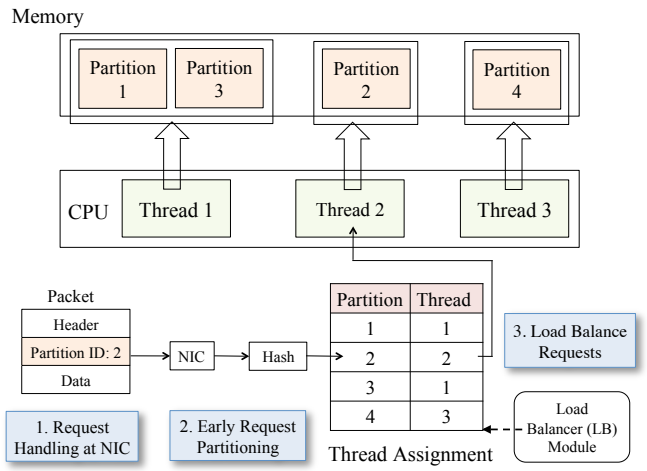


Figure 5: Chronos system overview.

in latency. Note that the performance of these synchronization primitives would be reduced if the workload pattern shifted towards a more write-heavy demand.

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:** Achieve low predictable latency by reducing the overhead of handling sockets and communication in the kernel. Reducing the application tail latency improves the latency predictability and the application performance.

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.

3. **Early request assignment and introspection:** Partition incoming client requests as early as possible to avoid application-level queue build-up due to request skew and application lock contention.

4. **Self tuning:** Dynamically load balance requests within a single node across internal resources to avoid hotspots and application-level queueing, without assuming apriori knowledge of the incoming request pattern and application behavior.

4.2 Design and Implementation

We now describe the design of Chronos (shown in Figure 5). Chronos partitions application data to allow concurrent access by the application threads (described in Section 4.2.2). It maintains a dynamic mapping between application threads and data partitions in a lookup table, and when a packet arrives at the server, Chronos examines the partition ID in the application header and consults the lookup table to steer the request to the proper application thread. The Chronos load balancer periodically updates the mapping between partitions and application threads to balance the load on each thread such that the request response time is minimized. In Chronos, requests are demultiplexed between application threads early, in the NIC, to avoid lock contention and multiple copies. At a high level, the Chronos request servicing pipeline is carried out in three stages: (1) request handling, (2) request partitioning and (3) load balancing.

4.2.1 Request Handling

As described in Section 3, a major source of latency in end host applications is the operating system stack. Chronos eliminates the latency overhead introduced due to kernel processing by moving request handling out of the kernel to user-space by using zero-copy, kernel-bypass network APIs. These APIs are available from several vendors and can be used with commodity server NICs [28, 34, 35].

We now explain one possible way of implementing user-level networking. When the NIC driver is loaded, it allocates a region of main memory dedicated for storing incoming packets. This memory is managed as send and receive ring buffers, called NIC queues. To bypass the kernel, an application can request an exclusive handle to one or more receive ring buffers for its different threads. The receive ring buffers are mapped to addresses in the application address space. Outgoing packets from the application are enqueued into a selected ring, and are sent on the wire by the NIC. The incoming packets at the NIC are classified to a receive ring based on the output of a hash function. This classifying function can be implemented in the hardware or in software. Though in-NIC request classification will be most efficient, it is less flexible than a software classifier. Chronos is not tied to any specific NIC implementation for user-level networking as long as it can correctly classify the incoming packets and assign them to the right application thread. For hardware classification, one could extend the receive-side scaling (RSS) feature in the NICs such that it hashes packets into rings based on a custom offset instead of hashing on the fixed 5-tuple in the packet header.

For prototyping Chronos, we use a custom hash function implemented in user space for request classification. The custom hash function enables deep packet inspection and arbitrary processing over the packet contents by executing the hash function on any one of the CPU cores. 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. This function returns the receive ring buffer id which the packet should be classified to. Note that there is no additional copying involved. However, software hashing has performance cost as it may cause cache misses. This is because the custom hash function would read the packet header first and then assign it logically to a ring buffer. The packet may then be processed by an application thread on a different CPU core, which may not share an L2 cache with the classifying core. For our implementation, the performance penalty due to user space processing was outweighed by the latency incurred in the kernel. For the simple custom hash functions we implemented the execution overhead is in nanoseconds, less than the packet inter-arrival times for 10 Gbps links.

Finally, note that the application is not interrupted as the packets arrive at the server. Instead, it must poll the receive ring buffer for new packets using *receive()*. For Chronos, we have a dedicated thread monitoring the NIC queues that registers packet reception events with the applications.

4.2.2 Request Partitioning

Bypassing the kernel significantly reduces the latency, since a request can now 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 namely lock contention, core overloading or processing hot-spots due to skewed requests. These bottlenecks are responsible for significant variation in latency causing unpredictability. A classic approach to reducing lock contention is to separate requests that manipulate disjoint application state as early as possible. Chronos uses this approach and minimizes shared state with static division of the state into disjoint partitions that can be processed concurrently. For instance, in case of Memcached, we replace a single centralized hash table with the entire keyspace and associated slab class lists with N hash tables and slab class lists with smaller regions of the keyspace. Each of these N hash tables represents a partition and can now be assigned to a hardware thread for concurrent processing. A single thread can handle multiple data partitions.

With partitioned data, we now need to send each request to the thread handling that partition. Chronos uses a classifying function (described in Section 4.2.1) to examine the application header for the partition ID and steering the request to the receive ring buffer of the thread which handles the data partition for the request. While it is possible to add a new field (partition ID) to the application header to steer requests to the appropriate application threads, we choose instead to overload an existing field. In 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.

The partitionable data assumption fits well for classes of applications like key-value stores, search, and OpenFlow. Handling requests for data from multiple partitions is an active area of research [22], and one we hope to study in future work.

4.2.3 Extensible Load Balancing

The endhost should be able to handle large spikes of load, with multiple concurrent requests, while running the underlying system at high levels of utilization. While static request partitioning helps in reducing lock contention, it could still lead to hot-spots where a single thread has to serve a large number of requests. To this end, we present a novel load balancing algorithm that dynamically updates the mapping between threads and partitions such that the incoming requests are equally distributed across the threads.

We now describe the load balancing mechanism. Chronos uses a classifier based on the partition ID field in the application header, and a soft-state table to map the partition ID field to an application thread. To reduce lock contention, the partition-to-thread mapping should ensure that each partition is exclusively mapped to a single thread. The load balancing module periodically updates the table based on the offered load and popular keys. For simplicity, assume that the Chronos load balancer measures the load on a data partition as a function of the number of incoming requests for that partition. This is true for key-value stores when each request is identical in terms of time required for processing the request (table lookup) but not for applications like in-memory databases. In general, the load on a partition is representative of the expected time taken to process the assigned requests. The number of requests served for each partition is maintained in user space for each ring buffer. A counter is updated by the classifying function while handling requests, and the load balancer could optionally be extended to measure the load in other ways as well. The load on a thread is the total load on all partitions assigned to a thread.

The Chronos load-balancing algorithm divides time into epochs, where each epoch is of maximum configurable duration T. The load balancer maintains a mapping of each partition to an application thread in the epoch, *epochMap*, along with per-partition load information. The load balancer also maintains a separate map for measuring thread load, *threadLoadMap* which indicates the number of requests served by an application thread in the current epoch.

Algorithm 1 $Chronos$ Load Balancer updates partitionID to thread mapping based on load offered in last epoch.

```
1: IdealLoad = totalEpochLoad/totalThreads
2: for all k ∈ {totalThreads} do
3:     threadLoadMap[k] = 0
4: end for
5: for all v ∈ partitionID do
6:     t = epochMap.getThread(v)
7:     if threadLoadMap[t] ≤ IdealLoad then
8:         currentEpochMap.assign(v, t)
9:         threadLoadMap[t].add(v.load)
10:    else
11:        for all k ∈ {totalThreads − {t}} do
12:            if threadLoadMap[k] ≤ IdealLoad then
13:                currentEpochMap.assign(v, k)
14:                threadLoadMap[k].add(v.load)
15:                break
16:            end if
17:        end for
18:    end if
19: end for
20: epochMap = currentEpochMap
```

The load balancing algorithm greedily tries to assign partitions to the least loaded thread only if the thread to which partition is already assigned is overloaded with requests. This is to avoid unnecessary movement of partitions across threads. When the application starts, the Chronos load balancer initializes the table with a random mapping of partition IDs to threads. Algorithm 1 shows psuedocode for the $Chronos$ load-balancer module. A new epoch is triggered when the duration T elapses. At the start of a new epoch, the load balancer computes the new mapping as described in Algorithm 1. The load balancer computes the total load in the last epoch and divides that by the number of threads to obtain the ideal load each thread should serve in the next epoch, under the assumption that load distribution will remain the same. In each epoch, it initializes the load for each thread to be zero. It then iterates through all partitions, checking if the thread it is currently assigned to can accommodate the partition load or not. If not, the algorithm assigns the partition to the first lightly loaded thread.

For the proposed algorithm to work effectively, the number of partitions should be at least the number of cores available across all of the application instances. Note that Chronos load balancing does not add to cache pollution that might happen due to sharing of partitions among threads. In fact, the baseline application will have lower cache locality given that all of its threads access a centralized hash table. While the proposed load balancing algorithm tries to distribute the load uniformly on all threads, Chronos can also be used with other load balancing algorithms which optimize for different objectives.

Note that concurrent access to the partitioned data is still protected by a mutex to ensure program correctness, however the partitioning function ensures that there is a serialized set of operations for 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 99^{th} percentile of latency.

4.3 Application Case Studies

Chronos does not require rewriting the application to take full advantage of its framework. Chronos requires only minor modifications to the application code for using the user-level networking API. To demonstrate the ease of deploying Chronos, we port the following three data center applications to use Chronos and evaluate the improvement in their performance.

Memcached: Rather than building a new key-value store, we base Chronos-Memcached (Chronos-MC) on the original Memcached codebase. Chronos-MC is a drop-in implementation of Memcached that modifies only 48 lines of the original Memcached code base, and adds 350 lines. These modifications include support for user-level network APIs, for the in-NIC load balancer, and for adding support for multiple partitions.

Web Search: Another application we consider is Web search, a well-studied problem with numerous scalable implementations [15, 20]. We choose Web search since it is a good example of a horizontally-scalable data center application. 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 term. For Chronos-WebSearch (Chronos-WS), we implement term-based partitioning. 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 data structures. We implemented an index server using read/write locks and UNet APIs. 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 the index server with an RCU mechanism from an open-source code base provided by the RCU authors [37]. We modified it to work with the UNet APIs. Chronos-WS further divides the index server table into several partitions based on terms for efficient load balancing.

OpenFlow Controller: We also implemented an OpenFlow controller application on Chronos (Chronos-OF) using code provided by [29]. 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. This application receives requests from multiple switches and responds with forwarding rules to be inserted in the switch table.

5. EVALUATION

In this section we evaluate the Chronos-based Memcached, Web server and OpenFlow controller using micro and macro-benchmarks. Overall, our results show that:

- Even with Memcached running on the MOSBENCH [13] kernel with an efficient network stack, the tail latency is still high. This justifies the use of kernel bypass networking APIs to deliver predictable low latency.
- Chronos-MC exhibits up to 20x lower mean latency compared to stock Memcached for a uniform request arrival rate of 200,000 requests/sec. For bursty workloads, it reduces the tail latency by 50x for a request rate of 120,000 requests/sec. Reduced tail latency improves the latency predictability and application performance.
- Chronos-MC can effectively scale up to 1 million requests/sec taking advantage of load balancing across concurrent threads.
- Chronos-WS achieves an improvement of 2.5x in mean latency as compared to baseline Web Server application that uses Read/Write locks.
- Chronos-OF achieves an improvement of 12x in mean latency as compared to baseline OpenFlow application.

We now describe our experiment setup, the workloads we use, and performance metrics we measure.

Testbed: We deployed Chronos on 50 HP DL380G6 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 use UDP as the transport layer protocol, along with support for binary protocol for efficient request parsing and virtual buckets for enabling load balancing.

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 the 1) number of requests per second served, 2) mean latency distribution, and 3) 99^{th} percentile latency distributions. To evaluate baseline Memcached and Chronos under realistic conditions, we use two load generators. The first, Memslap [1], is a closed-loop benchmark tool distributed with 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 since these values are representative of read-heavy data center workloads [19]. 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 (i.e client generates requests at a fixed rate irrespective of pending previous requests) we built in-house using low-latency, user-level network APIs to

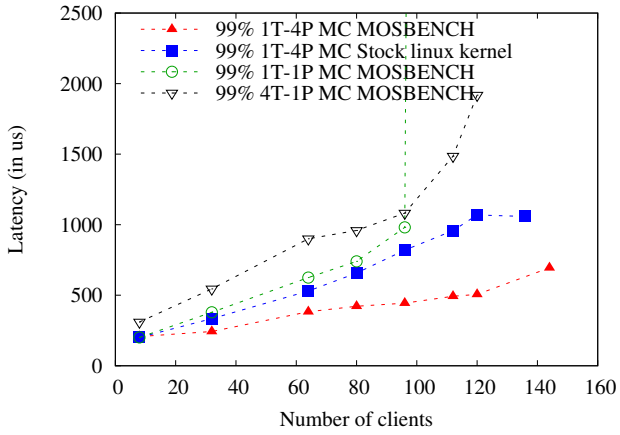


Figure 6: Legend: nT-mP stands for n thread m processes of Memcached(MC). Shown is the tail latency for one and four threads (1T and 4T) running in either one process or four processes (1P or 4P).

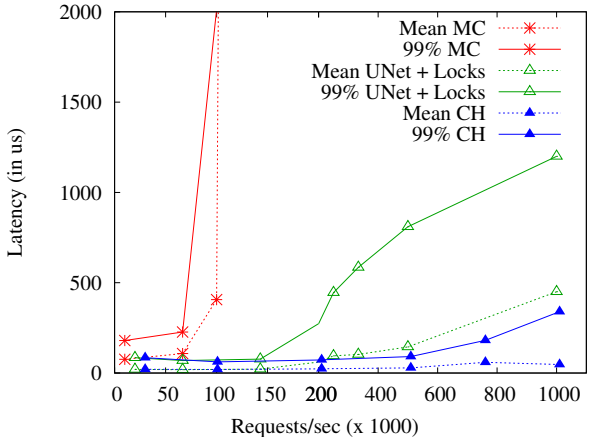


Figure 7: Latency of baseline Memcached (MC), Memcached with user-level network APIs (UNet locks), and Chronos (CH) with 10 open loop clients.

reduce measurement variability. Each instance of this second load generator issues requests at a configurable rate, up to 10Gbps per instance, with either uniform or exponential inter-arrival times. The KV-pair distribution used by the tool is patterned on YCSB [19]. Note that the latency numbers reported in figures generated by the closed-loop clients are higher by 50–70 μs compared to open loop clients since closed loop clients also report the kernel and network stack latency. For Chronos, we run the load-balancer every $50\mu s$, unless specified otherwise.

5.1 Memcached on an Optimized Kernel

We examine the latency of different configurations of Memcached instances – i) one single threaded, ii) one multi threaded (with four threads) and iii) multiple single threaded processes (four processes each running on its own core) – using the MOSBENCH kernel (pk branch) with an efficient network stack. The multi threaded Memcached incurs intra-thread lock contention, while the single threaded and multi-process configurations are free of intra thread lock contention. However, multiple single threaded Memcached processes can support more clients as compared to single threaded instances.

To measure the performance of these different configurations we use a configurable number of Memslap clients, each deployed on its own core to lower the measurement variability. A Memslap client opens a socket connection to one of the four Memcached process. While running in single threaded mode, and thus free of intra-thread resource contention, we expect the single threaded, multiple process Memcached latency and variance to be lower than multi-threaded instance on MOSBENCH. Figure 6 shows our results. For comparison, we also plot the performance of Memcached with the stock linux kernel. Our results show that even with the optimized MOSBENCH kernel, the 99^{th} percentile latency for four single threaded multi-process configuration is as high as 810 μs with 140 clients (35 clients/process), indicating that the kernel's contribution to the tail latency is significant despite kernel optimizations and a lack of application lock contention.

5.2 Uniform Request Workload

In this subsection we show that Chronos-MC reduces the mean application latency by a factor of 20x as compared to baseline Memcached for a workload with uniform inter-arrival time and access pattern for requests. Chronos-MC also outperformed a Memcached implementation that only leveraged user-level networking but no other Chronos feature (request partition or load balancing). We started instances of the three different Memcached implementations with four threads each. We also 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 1KB values and 64 byte keys in a 9:1 ratio of gets to sets. To avoid overloading the server beyond its capacity, each client terminates when the observed request drop rate exceeds 1%.

Figure 7 shows the results for this experiment. While 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 Memcached instance 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 99^{th} percentile, indicating that reducing variability in the network stack, operating system, and application are all important to reduce tail latency.

We also evaluate the performance of Chronos-MC with a larger number of closed loop clients. We instantiated eight client Memslap processes on each physical client machine, and scaled up to 50 client machines. As shown in Figure 8, we see that Chronos-MC supports over 1 million transactions per second (TPS), limited only by the NIC's throughput limit of 10Gbps. With 120 clients, the number of requests served levels out, causing a small amount of additional latency as requests wait to be transmitted at the client. In contrast, baseline Memcached serves fewer request/sec with high latency.

5.3 Skew In Request Inter-Arrival Times

In this subsection, we show that the techniques used in Chronos deliver predictable low latency even with skewed request inter arrival times. With the skewed workload Chronos achieves 50x im-

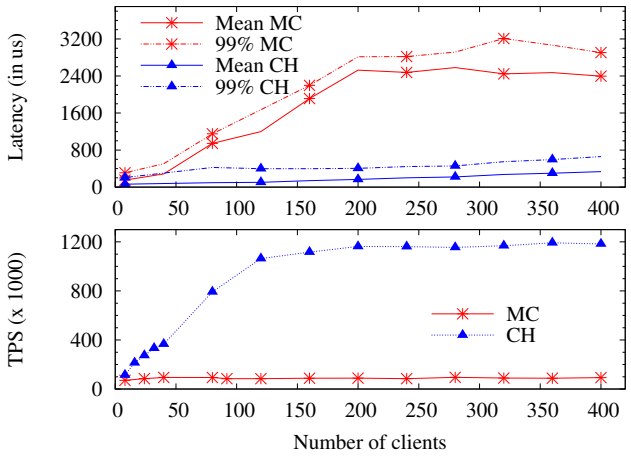


Figure 8: Latency as a function of the number of clients with the Memslap benchmark (closed loop).

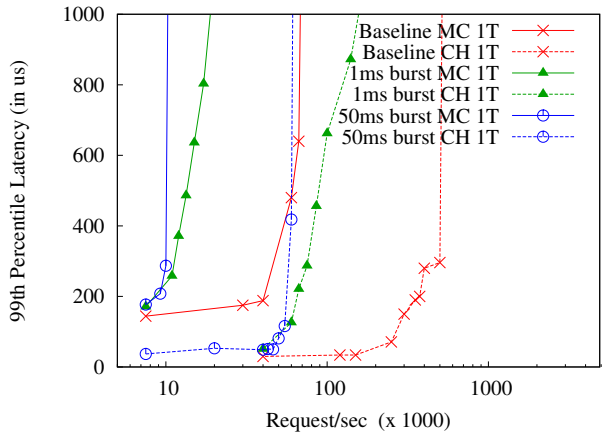


Figure 9: The effect of skewed request inter-arrival times on tail latency. X-axis in logscale.

provement relative to baseline Memcached while serving 10,000 requests per second.

The presence of skewed request inter-arrival times means that, although the average request load might be manageable, there are short periods of request 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 server point of view, skewed workload induces a momentary state of overload, which results in application-layer queueing. To study this behavior, we use the methodology described by Banga and Druschel [9], originally presented in the context of Web server evaluation. Here, multiple clients generate traffic at a fixed rate, punctuated with synchronized short bursty periods. These bursty periods are characterized by two parameters: 1) the ratio of the maximum request rate in the burst and the overall 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 such that each burst has 10% of the total requests sent. Lastly, we ensure that the number of requests in a burst are fixed across the experiments.

Figure 9 shows the 99^{th} percentile of latency for baseline Memcached (MC) as well as Chronos-MC (CH) across a range of burst periods. We see that in the baseline even short burst durations of

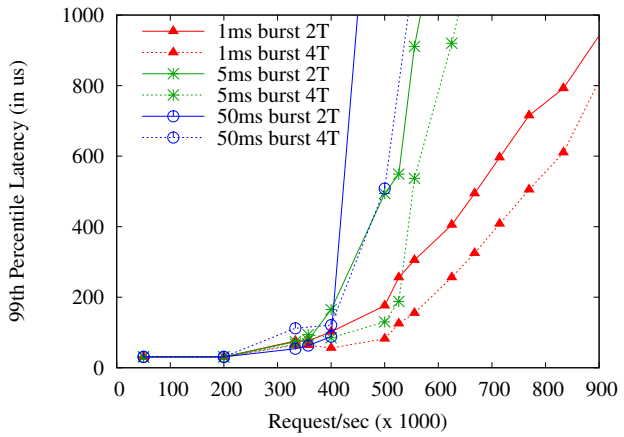


Figure 10: The latency with two threaded (2T) and four threaded (4T) instances of Chronos-MC under skewed request arrivals.

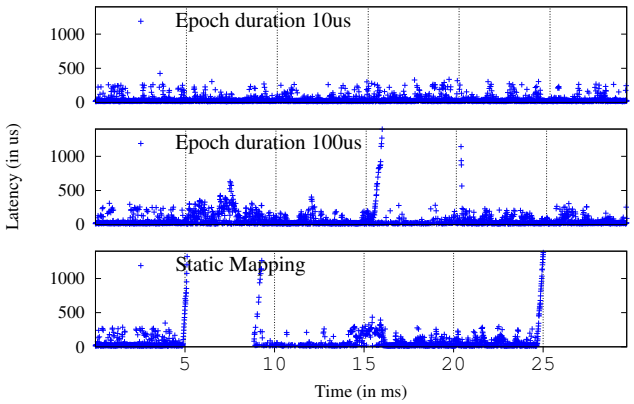


Figure 11: An evaluation of the responsiveness of the Chronos load balancer module across two time epochs (10μs on top and 100μs in the middle) and the static mapping strategy (on the bottom).

1ms impose significant levels of application queueing at 10,000 requests per second, driving latency up to over a millisecond. Note that without request inter-arrival time skew, baseline Memcached supported up to 120,000 requests per second with sub 500μs latency. (Figure 7). For Chronos-MC under a uniform request inter-arrival rate, latency stays largely flat up through 500,000 requests per second (Figure 7). However, just as in the baseline Memcached case, inducing request bursts drives up latency significantly while reducing the throughput of the system. For 1ms bursts, the request rate is reduced to 40,000 requests per second for keeping the latency under 30μs, with an observed latency of up to 1ms at over 150,000 requests per second. For longer burst durations, this effect is more pronounced.

Figure 10 shows how load balancing with more threads improves the performance of Chronos-MC. We consider 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 queueing, 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 queueing will still build up with any fixed number of CPU cores.

5.4 Skew in Request Access Pattern

In this section, we show how loadbalancing with Chronos at fine grained time scales significantly reduces the latency variation with skewed request access patterns. Results are shown in Figure 11. The Chronos load balancing module periodically reapportions requests across application threads to evenly balance the load. As described in Section 4.2.3, 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 as to minimize or eliminate lock contention. Thus, with Chronos-MC, it is expected that the load balancer assigns requests across application threads such that each thread sees a strict partition of vBuckets.

We run the following experiment, to evaluate the responsiveness of Chronos-MC to request access skew. We set up a Chronos-MC instance with four threads and configure the load balancing module with an epoch time of $10\mu s$ and $100\mu s$. 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 client 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 have the remaining core receive all of the cold/unpopular keys. We know by construction that without an adaptive load balancing module, each time the client 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 as to induce overload in that case. Note the client and the server epochs are not synchronized. We repeat the same experiment for a Chronos instance with static mapping of keys to threads. Figure 11 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. Unlike previous figures which show only 99^{th} percentile latency number, Figure 11 shows all data-points including few outliers.

Discussion: Due to our reliance on partitioning to spread load across cores, there are certain cases that will cause the load balancing element in Chronos to perform poorly. When a single key in a partition, or the partition itself, becomes hugely popular, the rate of requests to that partition can overload a single 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.

5.5 Chronos 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-WS, 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 [37]. 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 keys and 1400-byte values in the inverted index list, with a get/set requests ratio equal to 9:1. Figure 4 shows the results of our evaluation. Even with an implementation based on read/write locks and RCU, we see higher latency compared to Chronos-WS with large number of clients. The performance improvement of Chronos-WS would be higher if the workload shifted towards a more write-heavy mixture. The reason for this is that these primitives are optimized for read-heavy workloads and Chronos makes no such assumption about workload type. The RCU implementation based on user-level APIs scales up to 550K requests/sec, while the Chronos implementation scales up to 1M requests/sec. At low request rates and low levels of concurrency, the RCU implementation has similar performance as Chronos-WS. But as we increase the number of clients, and thus load on the server, the application latency increases from 2-3 microseconds to 6-11 microseconds for RCU-WS. This small variation in application latency results in a large end-to-end latency at high loads due to increased queuing delay.

Component	# Switches	Mean latency (μs)	99 %ile latency (μs)
OpenFlow	1	65	140
OpenFlow	16	120	250
Chronos-OF	1	8	50
Chronos-OF	16	10	51

Table 2: Latency of the OpenFlow Controller.

5.6 Chronos OpenFlow Controller

Finally, we show that the Chronos based implementation of the OpenFlow controller (Chronos-OF), which uses TCP for handling requests, reduces the mean latency for request processing by a factor of 12x as compared to baseline.

For this experiment, we replaced the default kernel TCP network implementation in the controller with the user-level TCP implementation provided by our NIC vendor in our evaluation testbed. The controller software itself is single-threaded. For generating load, we used the *Cbench* benchmark [18]. Cbench emulates switches that send *packet-in* messages to the controller, and waits for flow modification rules to be inserted in the switch forwarding tables in response. The controller implements a learning switch application, which generates appropriate forwarding rules in response to packet-in events. We simulated 16 switches supporting 1M MAC entries as suggested in [18]. To measure the controller latency, we installed a packet mirroring rule described in Section 3.

Table 2 shows the results of this experiment. We see that removing the kernel has the predictable effect of reducing average latency. However, the effect on the 99^{th} 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. We expect Chronos-OF controller performance to improve further by enabling load balancing for a multi-threaded implementation.

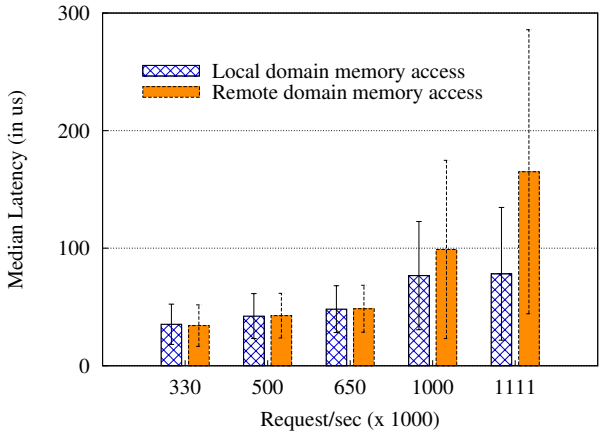


Figure 12: The effect of NUMA-awareness on the Chronos-Memcached load balancer. There is little difference at lower levels of utilization, and an approximate doubling of latency (and latency variation) at the highest levels of utilization.

6. DISCUSSION

To achieve high efficiency, data center networks often rely on multi-tenancy and server virtualization to maximize resource usage. The feasibility of Chronos depends on being able to support these techniques in in a variety of different data center environments.

In a large, multi-tenancy data center, latency sensitive applications share the same endhost with other jobs. A key question for Chronos is what impact this sharing has on latency, and in particular tail latency. To gain some insight into this question, we setup an experiment to test this condition. We first set up a Memcached server, and started a background job that receives traffic from six clients in parallel. Each client sends traffic at rate of 440Mb/s to this background job. We instantiated 21 Memslap clients, and measured the latency of both a stock Memcached server, and Chronos, with and without the presence of the background traffic. These particular rates and numbers of clients were chosen to induce sufficient load on the system to evaluate this question. In the case of baseline Memcached, the presence of background traffic resulted in more than a 60% increase in tail latency, while Chronos-MC's performance was not affected by the presence of the background traffic. This initial result indicates that Chronos can provide low latency in the presence of multi-tenancy, and we seek to further evaluate this in more depth in future work.

Supporting virtualization in the data center and consolidating multiple VMs on a single endhost have become common place today. NIC hardware has been augmented to support SR-IOV, or Single Root I/O Virtualization. SR-IOV allows a guest OS to directly configure access to virtualized instances of the NIC without going through the hypervisor. Although not implemented in this work, we expect Chronos to leverage these features to provide predictable latency in a virtualized setting.

6.1 Effect of NUMA-awareness on latency

Modern processor architectures employ non-uniform memory access (NUMA) architectures, in which memory is partitioned across two or more banks, or domains. The access time to a core-local domain is lower than that of a remote domain, and so it is advantageous to organize memory to be as domain-local as possible. To evaluate the effect of NUMA on Chronos, we setup an experiment as follows. We choose a Chronos-based Memcached instance with four threads, of which two are in one NUMA domain, and two are in the other. We then adjust the memory allocator to allocate domain-local memory for each thread. We compared the observed latency of this with a second Chronos-based Memcached instance in which the allocator selects entirely domain-remote memory for each thread.

Figure 12 show the latency in these two cases. At low to medium rates of requests, there is little difference between the two policies. As the request rate exceed 1 million requests per second, there is a divergence in which the NUMA-remote instance imposes almost double the latency of the NUMA-local instance, with significantly high latency variation.

In our testbed, each NUMA domain contained four cores, which alone were enough to saturate the 10 Gbps NIC. Thus, it is not necessary to load balance requests across NUMA domains to meet throughput requirements. So ensuring that the load balancer restricts requests to NUMA-local cores is adequate for current link speeds. Furthermore, when running the server in low or moderate request loads, the effect is minimal in either case. Thus NUMA effects are not significant to the efficiency of Chronos, however their effect might become more pronounced in environments utilizing virtual machines. The specific issue arises when cores from different NUMA domains are assigned to the same virtual machine, causing high memory latencies and increasing tail latency.

7. RELATED WORK

Optimized Network/OS interfaces: A key bottleneck that our work addresses is the kernel and network stack overhead. We share this goal with several academic and industrial efforts. User-level networking was developed to support applications which emit packets at a high rate, and to reduce latency in the kernel [16, 11, 39]. Arsenic [30] proposed installing custom filters in NIC for packet classification. While user-level networking APIs are integral to the early partitioning aspect of our design, Chronos also facilitates per-CPU core load balancing and removing application lock contention through deep-packet inspection using these APIs to reduce application tail latency. Myrinet [12] and Infiniband [21] are examples of low-latency, high bandwidth interconnect fabrics that are often used in high-performance computing clusters. While Myrinet and Infiniband address a key bottleneck, Chronos focuses on commodity Ethernet switching and eliminates latency across the entire end-to-end application path, including application lock contention and hotspots.

Operating System Improvements: There have been various proposals on improving the scalability and performance of the Linux kernel. Corey [13] identified numerous instances of in-kernel data structure sharing which reduced potential parallelism across threads, and proposed address ranges, kernel cores, and sharing to improve kernel performance. In [31], the authors conclude that locking and blocking system calls were significant causes of application performance degradation. Boyd-Wickizer et al. [14] study the scalability of seven applications, including Memcached, across a 48-core machine and conclude that by modifying the kernel and applications, it is possible to remove many performance bottlenecks. However, their study focused on throughput, and not latency. With Chronos, we find that even for single-threaded processing the kernel introduces significant additional latency, even after accounting for these recent improvements. An analysis of latency in the endhost network stack was carried out by Larsen et al. [24].

Lock Contention: Lock contention has long been recognized as a key impediment to performance of shared memory and multi-

threaded applications [36]. Replacing mutex locks with read/write locks may have little advantage [17]. Triplett et al. [37] propose a dynamic concurrent hash table with resizing using a *read-copy update* (RCU) mechanism. This mechanism works well in situations where the number of reads is significantly greater than writes. VoltDB [38] and H-Store [22] 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.

Data center Networks & Applications: New transport protocols like DCTCP [5] and QCN [4] reduce in-network queueing and congestion, further reducing network latency. Recent proposals such as DeTail [40] and HULL [6] also focus on reducing latency by performing in-network traffic management. There have been numerous efforts to improve Memcached's throughput [10, 33], though none specifically look at improving predictable tail latency. Previous efforts to improve performance of Memcached over RDMA protocol [8, 23] required redesigning Memcached from the ground up and works only for a single threaded implementation.

8. CONCLUSIONS

The scale of modern data centers 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 Memcached, OpenFlow, and a Web search application implemented on Chronos, we show that we can reduce latency by up to a factor of twenty, while significantly reining in latency outliers. Reducing the tail-latency of data center applications results in improving efficiency of data center applications since more clients can be served from a limited set of resources. The result is a system that can enable more throughput by increasing predictability, a key contribution to improving data center efficiency.

9. ACKNOWLEDGMENTS

We would like to thank Abhijeet Bhorkar and Mohammad Naghshvar for input on our analytical analysis, as well as the anonymous reviewers of this work for their valuable insight and advice. We would also like to thank authors of [37] for sharing RCU implementation. This work was supported in part by NSF Grants CSR-1116079 and MRI CNS-0923523, and a NetApp Faculty Fellowship.

10. REFERENCES

[1] Memslap Benchmark. `http://docs.libmemcached.org/memslap.html`.

[2] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *SC*, 2009.

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

[4] 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 *CCC*, 2008.

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

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

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

[8] J. Appavoo, A. Waterland, D. Da Silva, V. Uhlig, B. Rosenburg, E. Van Hensbergen, J. Stoess, R. Wisniewski, and U. Steinberg. Providing a Cloud Network Infrastructure on a Supercomputer. In *HPDC*, 2010.

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

[10] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-Core Key-Value Store. In *IGCC*, 2011.

[11] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *ISCA*, 1994.

[12] 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.

[13] 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 *OSDI*, 2008.

[14] 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 *OSDI*, 2010.

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

[16] P. Buonadonna, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *SC*, 1998.

[17] B. Cantrill and J. Bonwick. Real-world concurrency. *Queue*, 6(5):16–25, Sept. 2008.

[18] OpenFlow Cbench Controller Benchmark. `http://www.openflow.org/wk/index.php/Oflops#Benchmarks`.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[20] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *SOSP*, 1997.

[21] Infiniband. `http://www.infinibandta.org/`.

[22] E. P. Jones, D. J. Abadi, and S. Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *SIGMOD*, 2010.

[23] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *ICPP*, 2011.

[24] S. Larsen, P. Sarangam, and R. Huggahalli. Architectural breakdown of end-to-end latency in a TCP/IP network. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 195 –202, Oct. 2007.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[26] Memcached. `http://memcached.org/`.

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

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

[29] OpenFlow Controller Source Code. `http://www.openflow.org/wp/downloads/`.

[30] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *INFOCOM*, 2001.

[31] Y. Ruan and V. S. Pai. The Origins of Network Server Latency & the Myth of Connection Scheduling. In *SIGMETRICS*, 2004.

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

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

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

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

[36] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *PPoPP*, 2010.

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

[38] VoltDB. `http://voltdb.com/`.

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

[40] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz. Detail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM*, 2012.